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

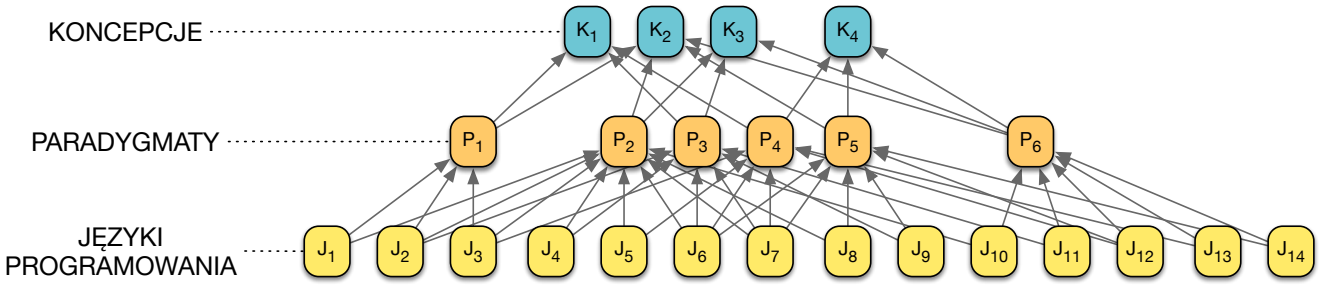
- języki — paradygmaty — koncepcje
- przegląd podstawowych koncepcji
- przegląd podstawowych paradygmatów

Języki — paradygmaty — koncepcje
Paradygmat

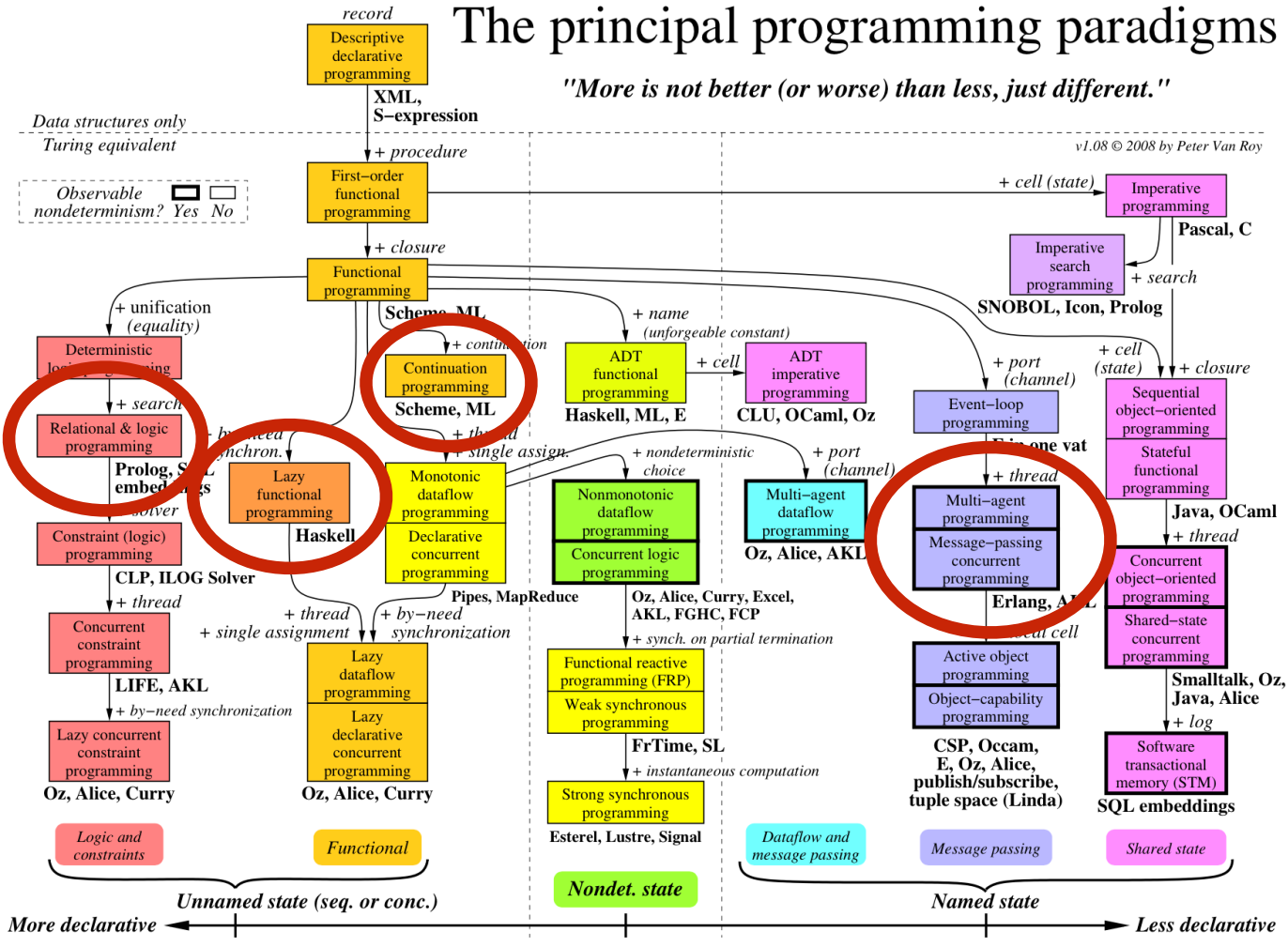
paradygmat [gr. παράδειγμα = przykład, wzór]
przyjęty sposób widzenia w danej dziedzinie, doktrynie itp.

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, …



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

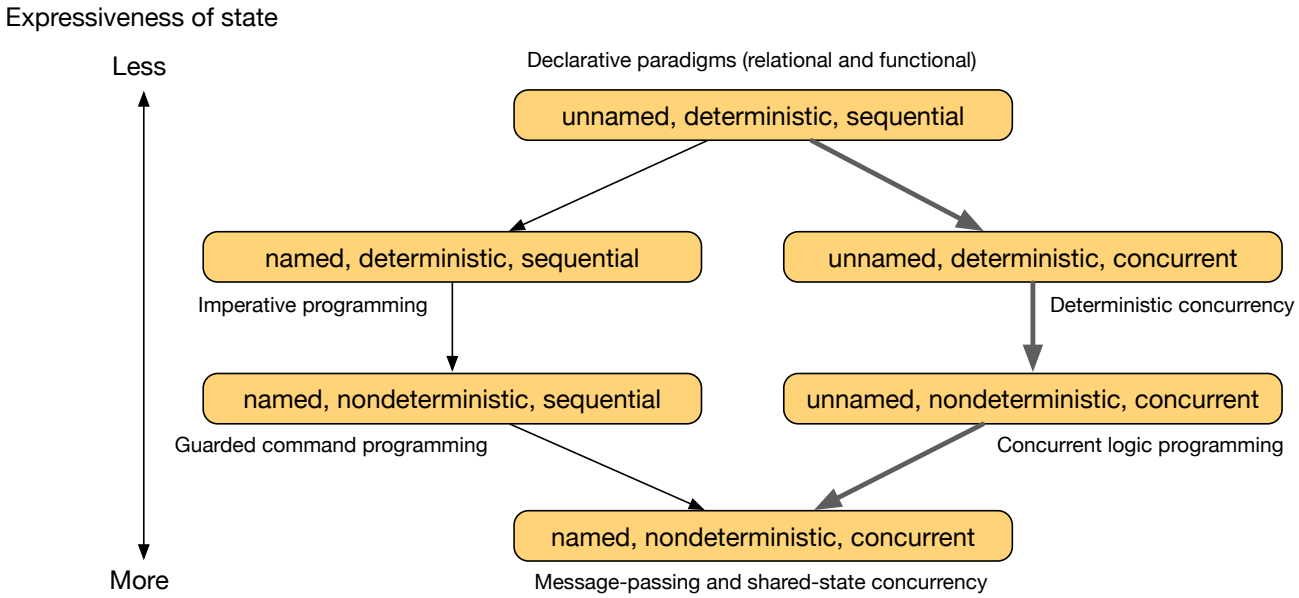
program planujący (scheduler)
podczas wykonywania wybór dalszych obliczeń dokonywany jest przez program planujący będący częścią run-time

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



Koncepcja: **rekord** (record)

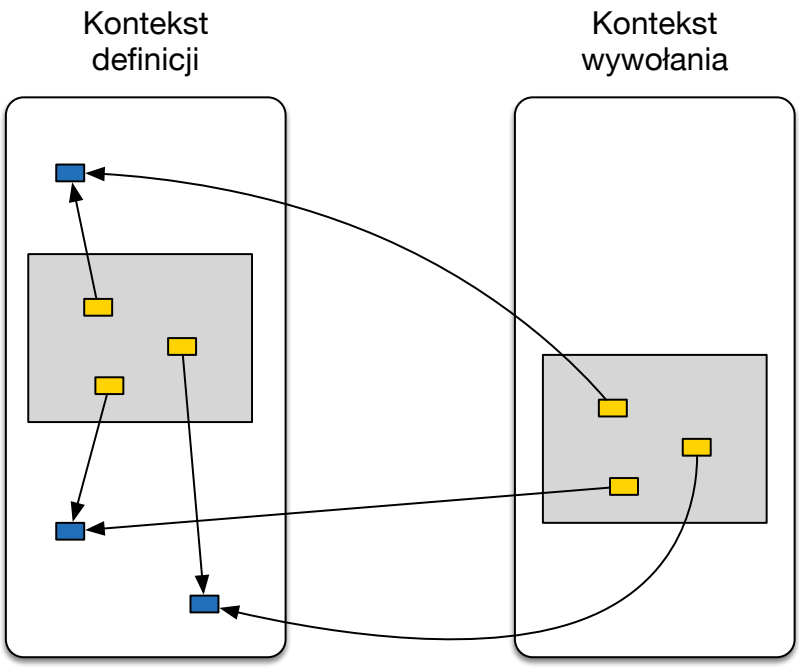
R=nazwa(pole$_1$:wartość$_1$ pole$_2$:wartość$_2$ … pole$_n$:wartość$_n$)
R.pole$_1$ R.pole$_2$ … R.pole$_n$

R=nazwa(1:wartość$_1$ 2:wartość$_2$ … *n*:wartość$_n$)
R.1 R.2 … R.n

R=nazwa(wartość$_1$ wartość$_2$ … wartość$_n$)
R.1 R.2 … R.n

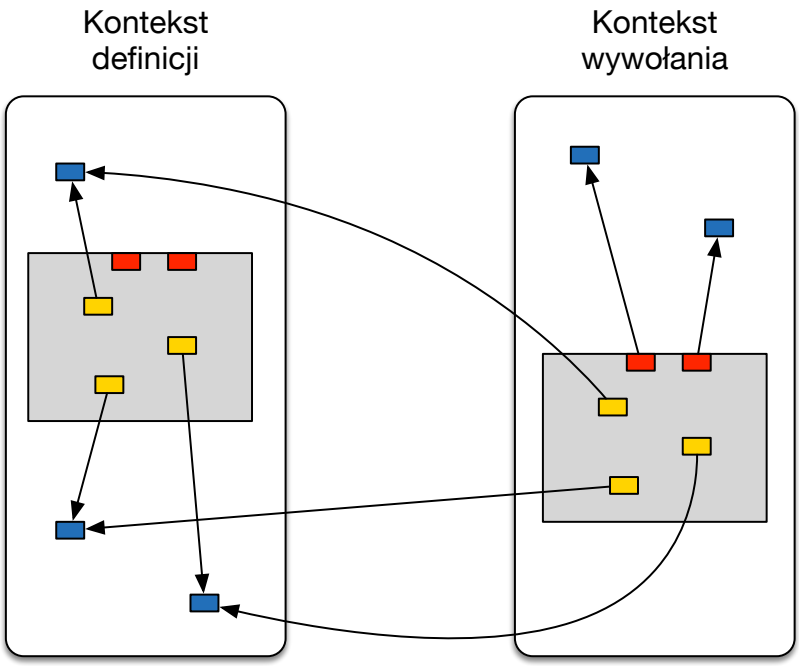
C=circle(x:10 y:20 radius:5)
R=rectangle(x:100 y:150 height:20 width:200)

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 zależne[1)]

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)

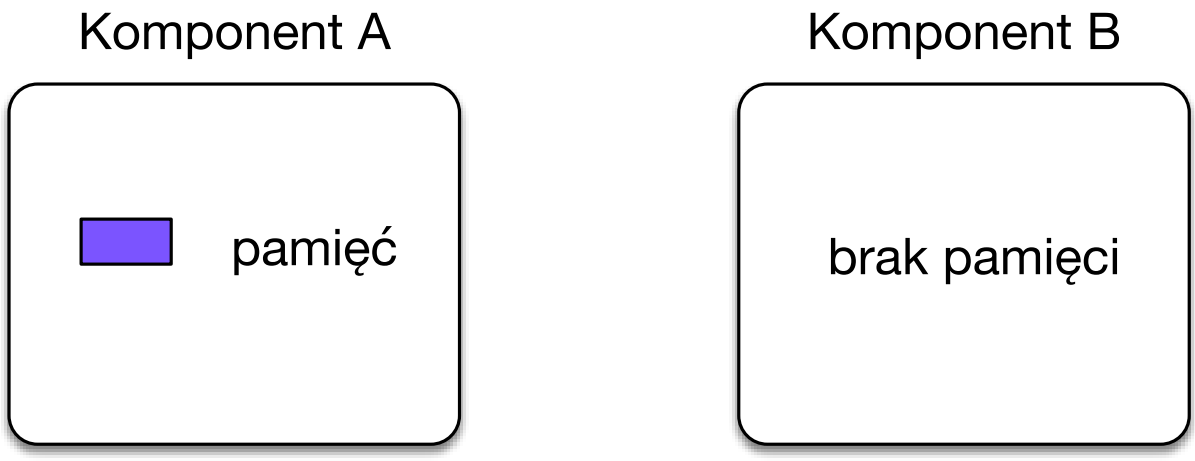
Trzy poziomy współbieżności:

- 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.

nazwany stan (named state)
sekwencja wartości w czasie jaką posiadała pojedyncza nazwa



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 …}
```

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

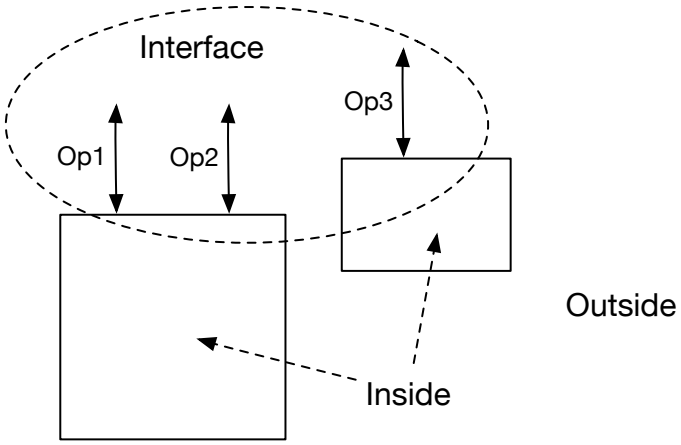
```
fun {F … Fin Fout}
    Fout=Fin+1
    …
end
```

```
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

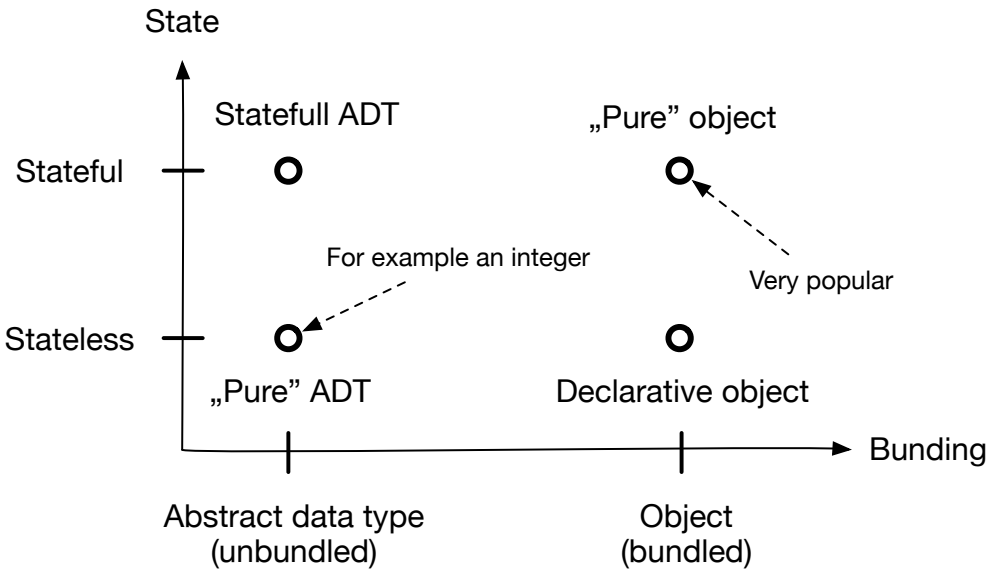
```
fun {NewStack}: <Stack T>
fun {Push <Stack T> T}: <Stack T>
fun {Pop <Stack T> T}: <Stack T>
fun {IsEmpty <Stack T>}: <Bool>
```

```
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)

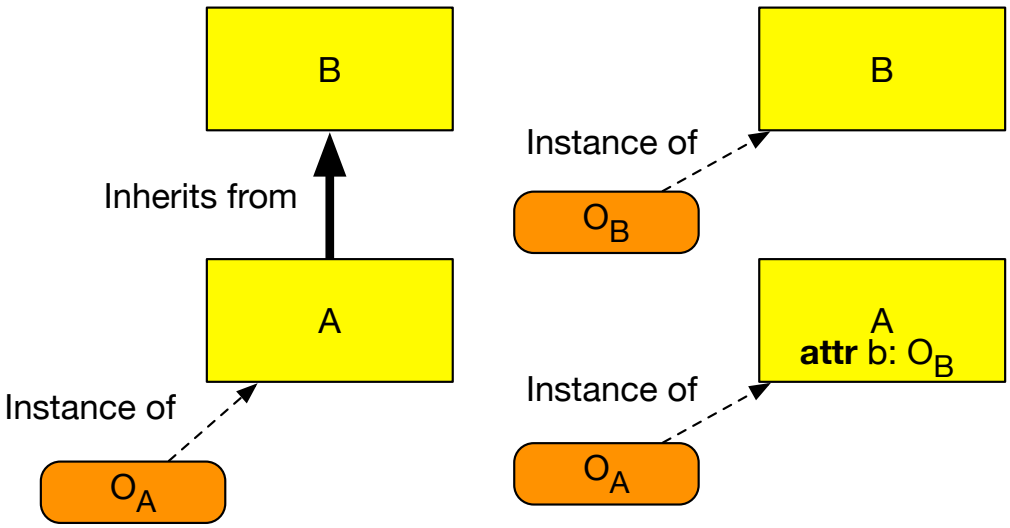


Abstrakcja danych: **polimorfizm**

```
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**



Inheritance Composition

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 posible?	Inputs can be nodeterm.?	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*):

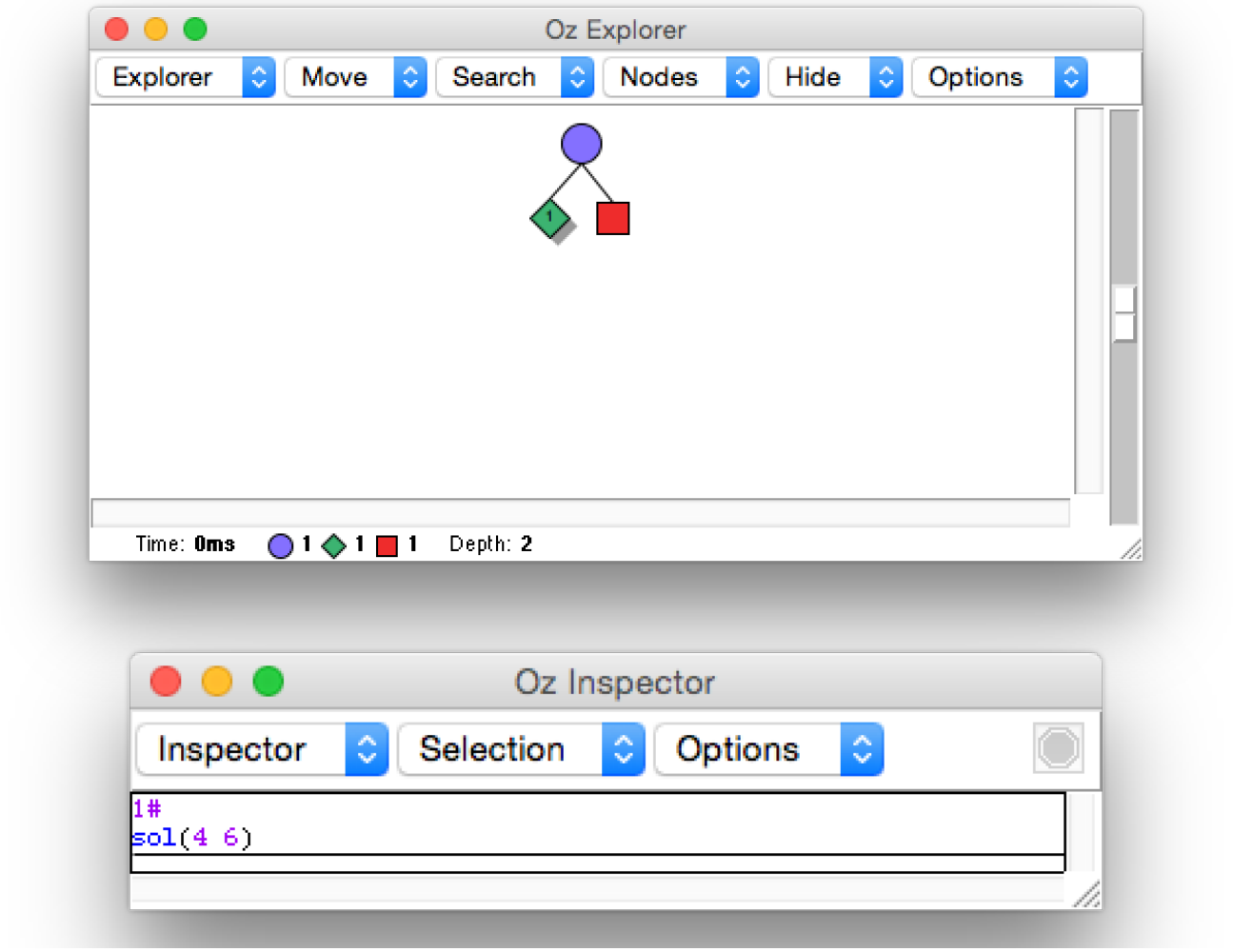
- `{WaitNeeded X}`: bieżący wątek czeka aż któryś z wątków wykona `{Wait X}`.

```
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∧C) oraz (P∧¬C) w konsekwencji decydując p kształcie drzewa poszukiwań. Wspomniany podział dokonywany jest rekurencyjnie w każdym węźle drzewa.

Zagadka kryptoarytmetyczna:

	S	E	N	D
+	M	O	R	E
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*.

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

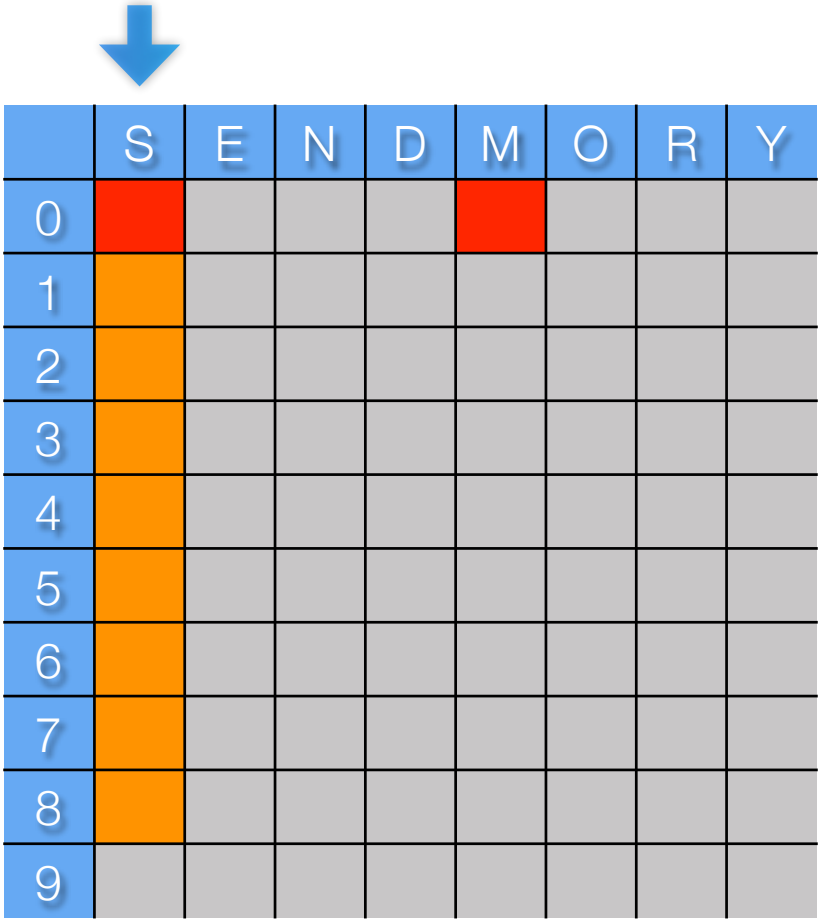
$$1000\cdot s_0 < min \rightarrow s \in \lceil min/1000 \rceil..s_1$$

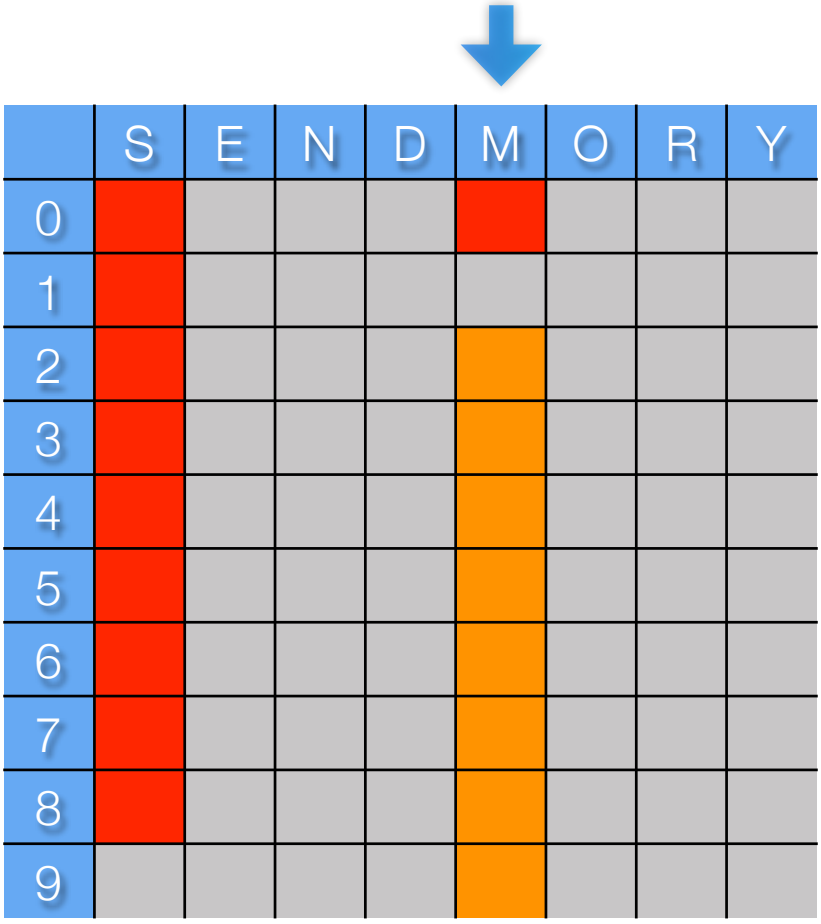
Na tym etapie:

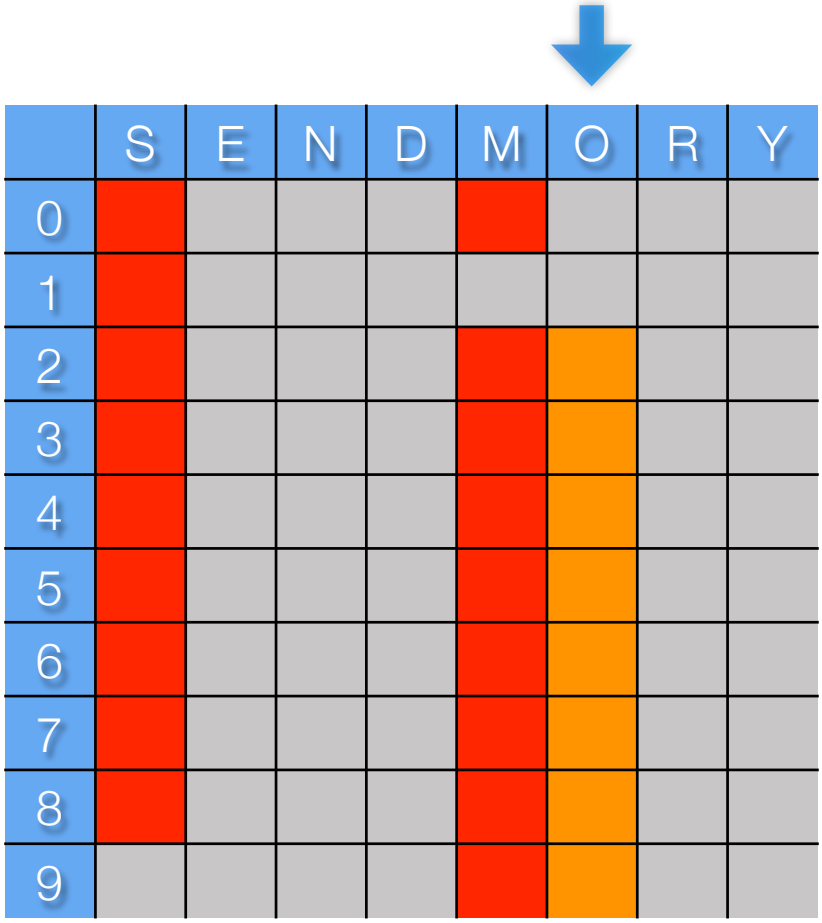
$$min = 8082,\ max = 89919,\ s_0 = 1\ s_1 = 9 \rightarrow s \in 9..9$$

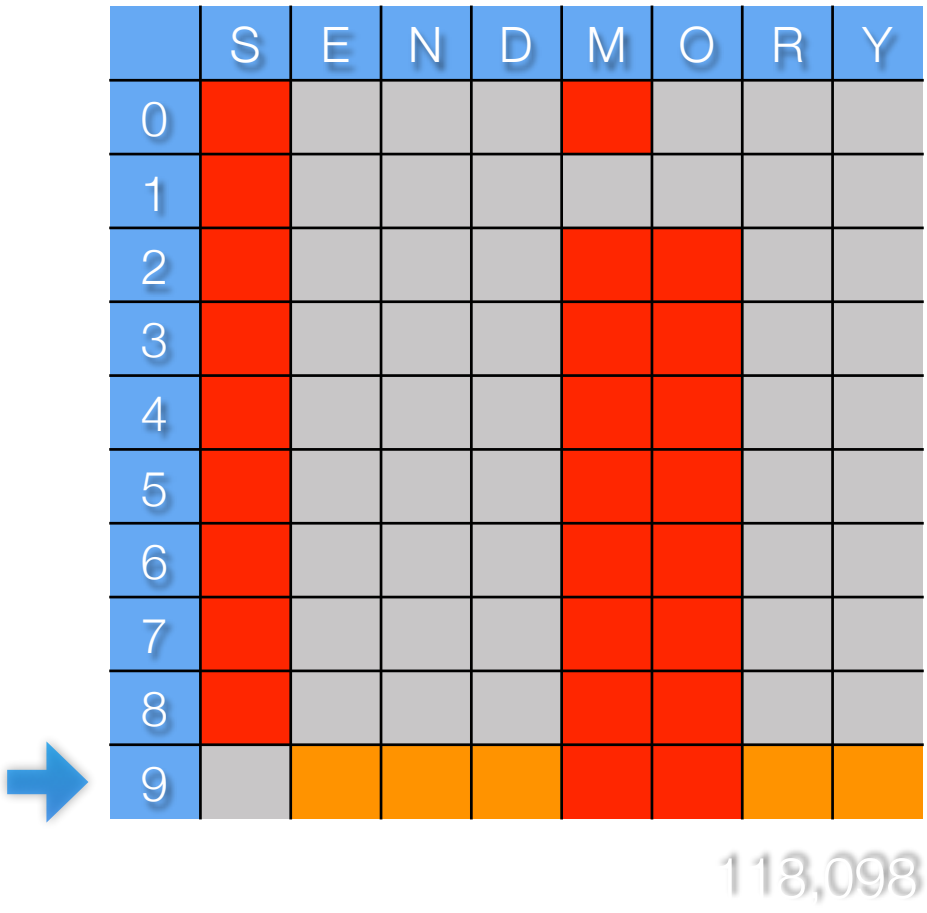
	S	E	N	D	M	O	R	Y
0								
1								
2								
3								
4								
5								
6								
7								
8								
9								

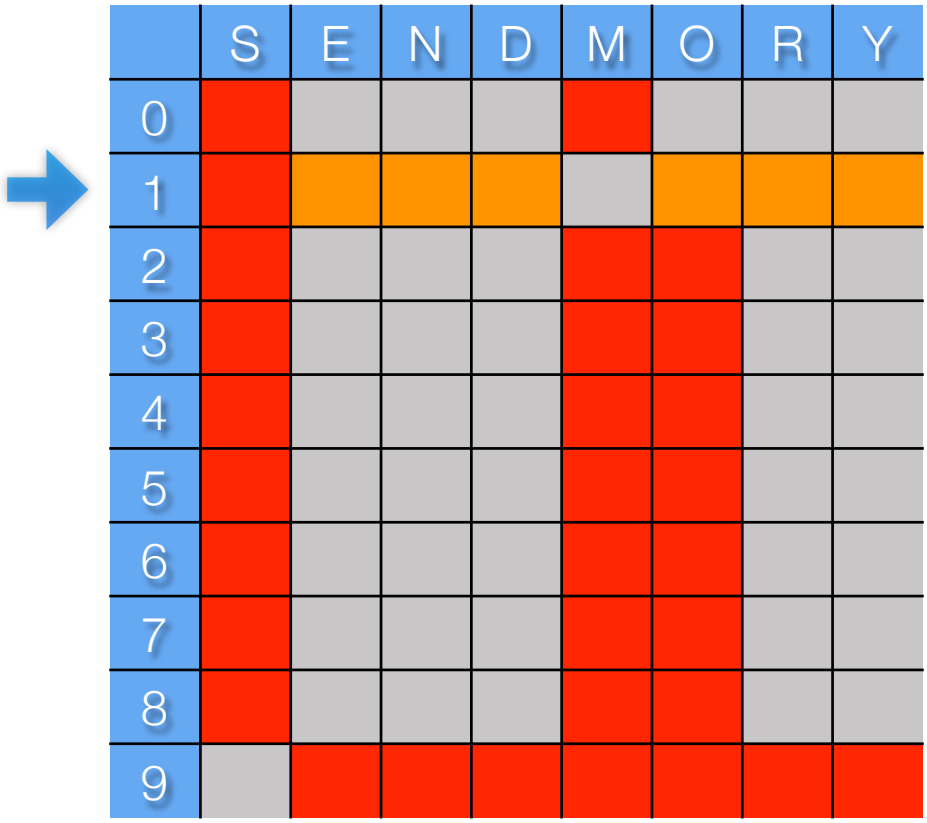
31,000,000


S E N D M O R Y
0
1
2
3
4
5
6
7
8
9
9,000,000

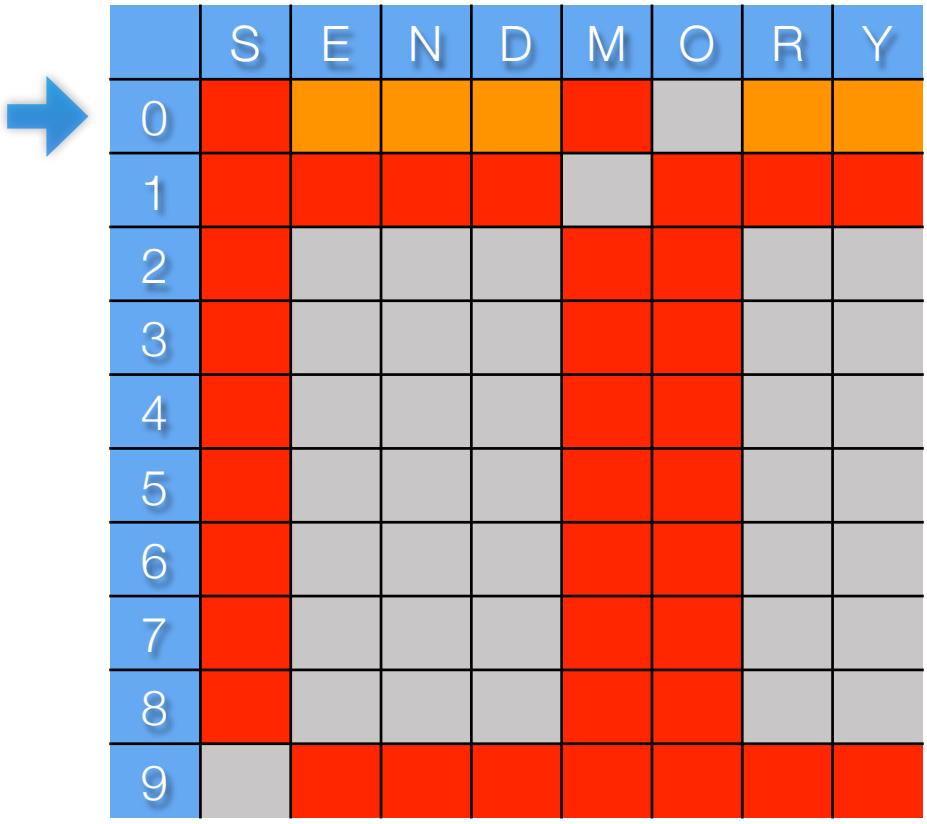

S E N D M O R Y
0
1
2
3
4
5
6
7
8
9
1,000,000


S E N D M O R Y
0
1
2
3
4
5
6
7
8
9
200,000

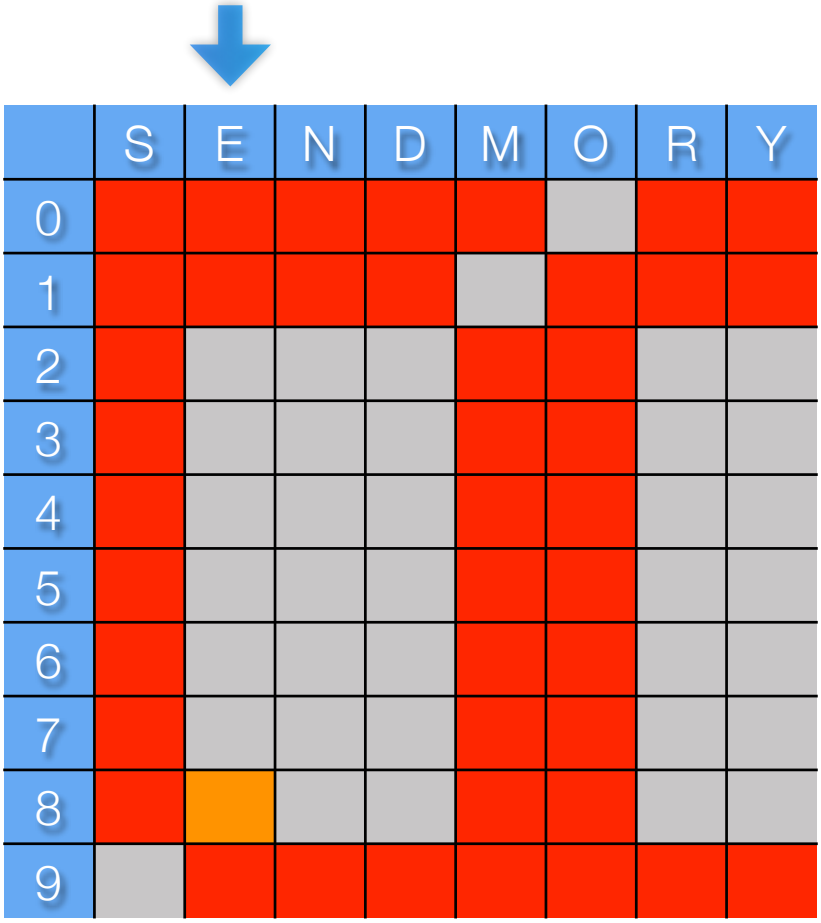

S E N D M O R Y
0
1
2
3
4
5
6
7
8
9
118,098



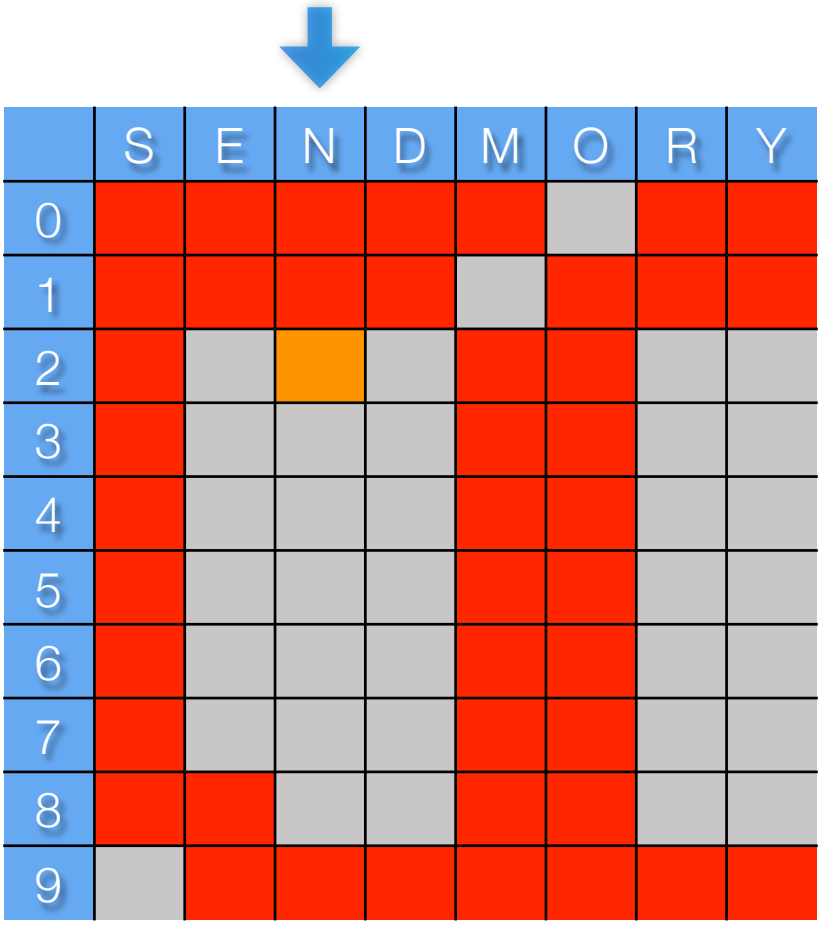
32,768



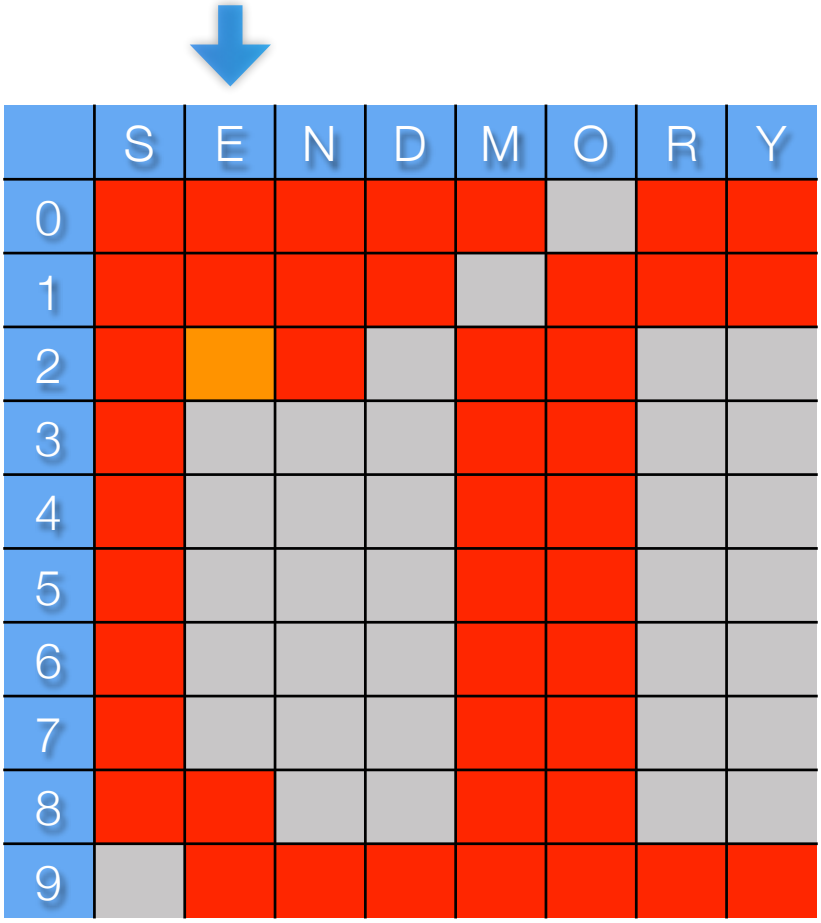
16,807



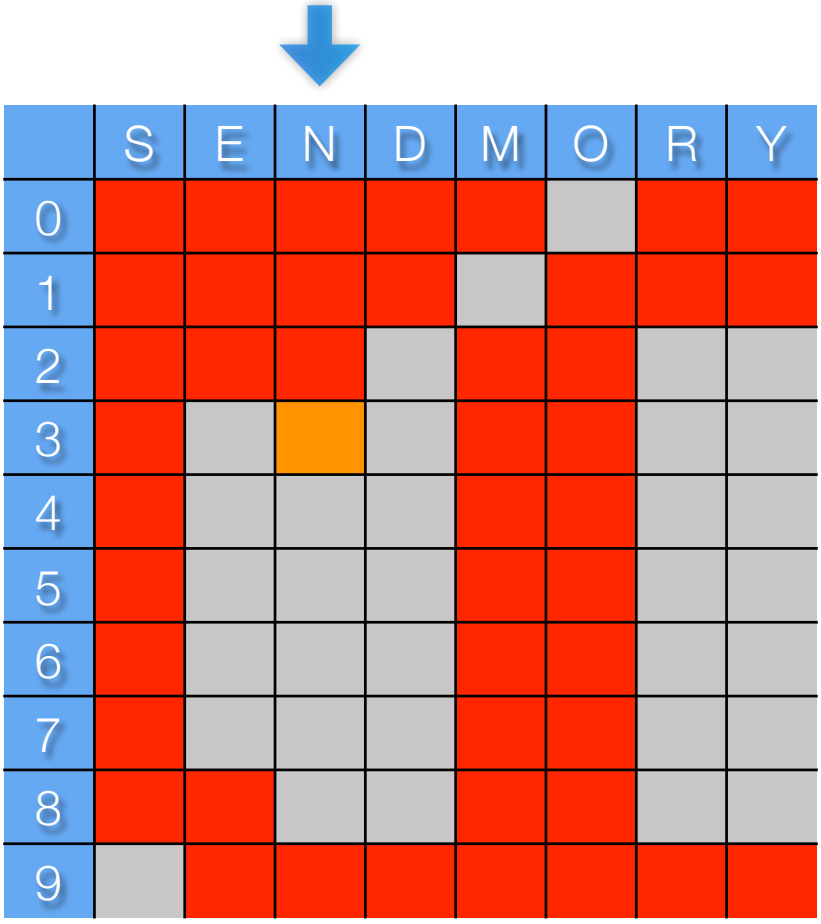
14,406



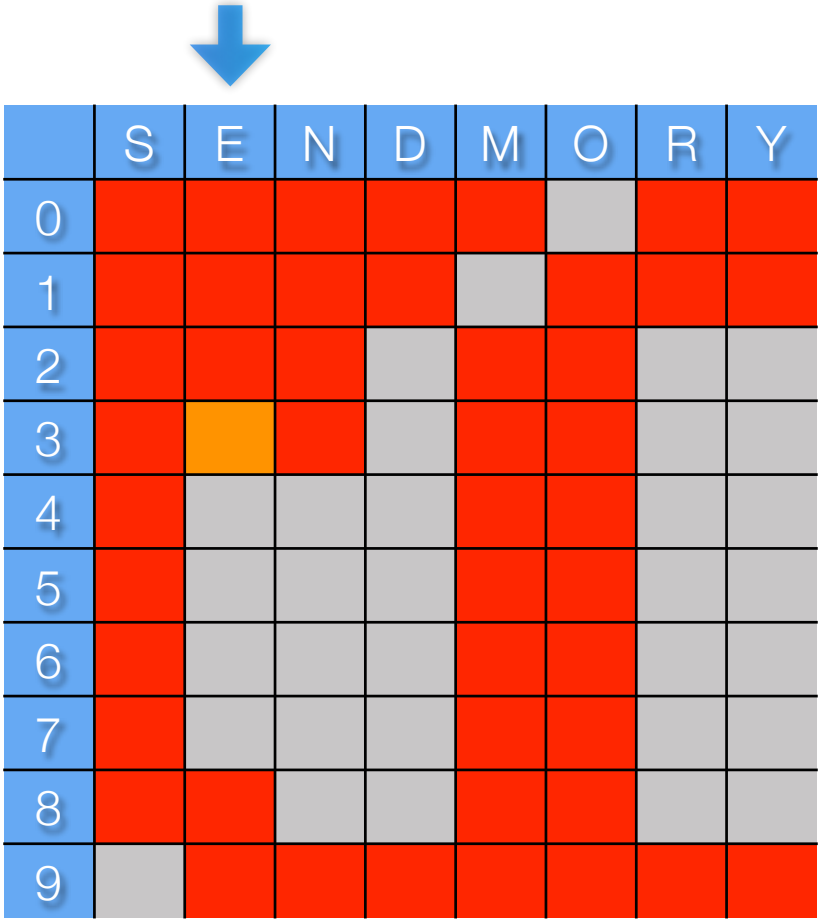
12,348



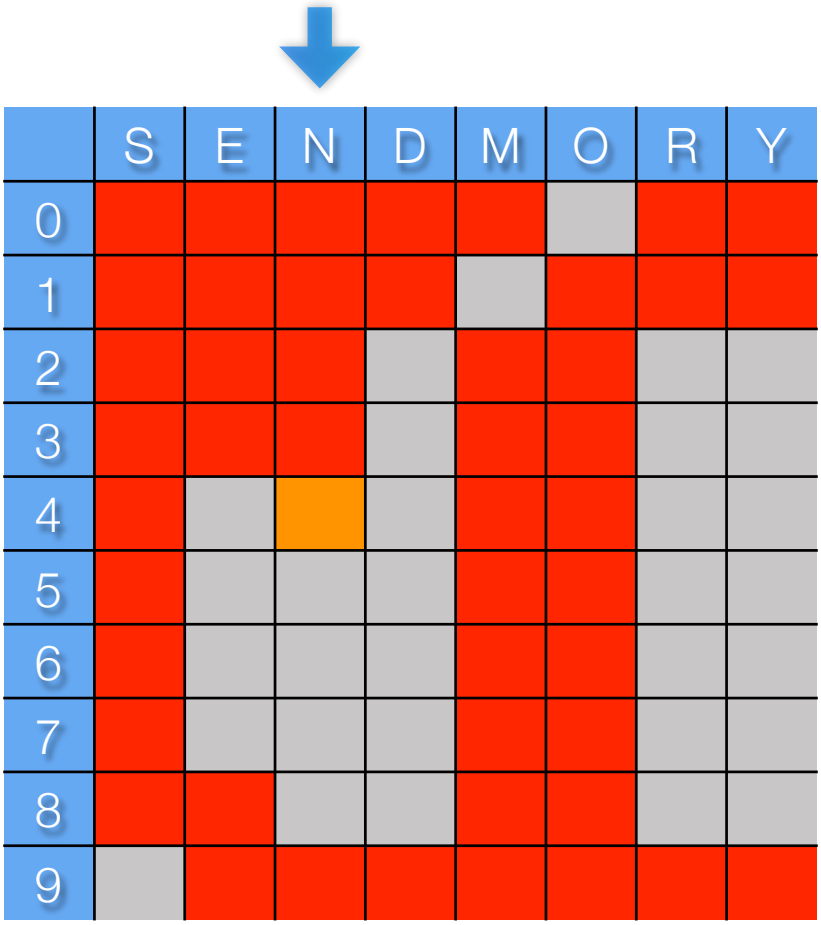
10,290



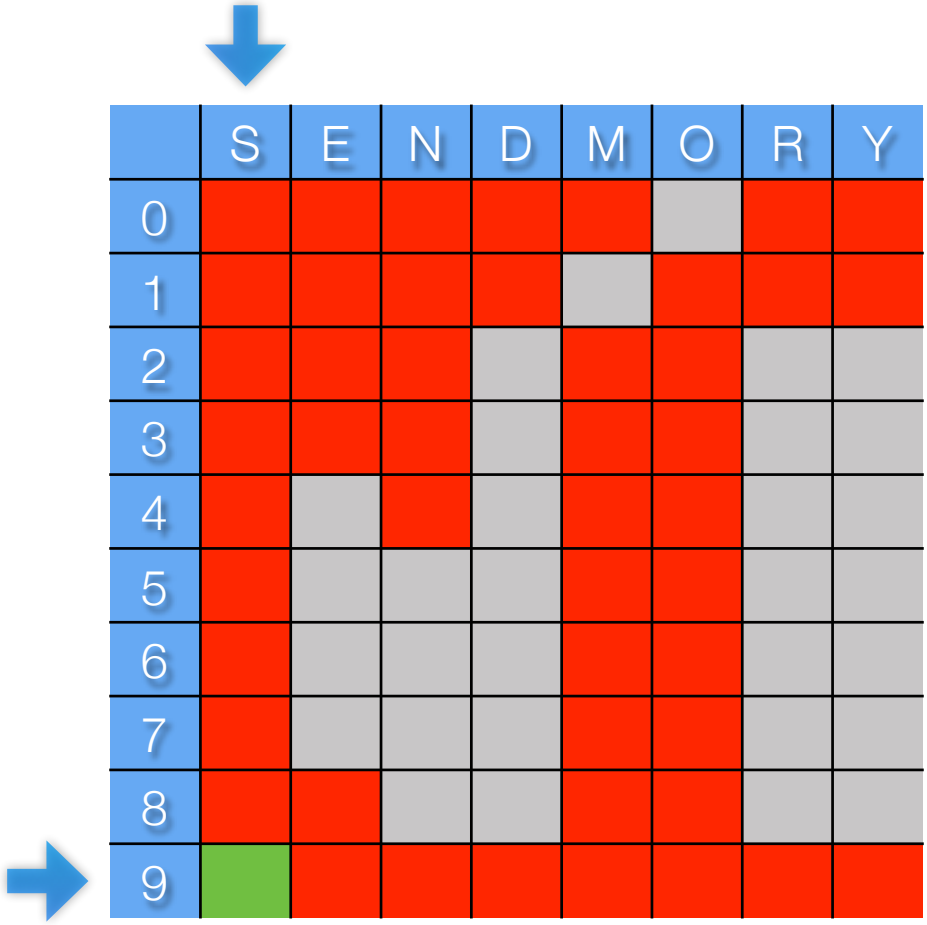
8,575



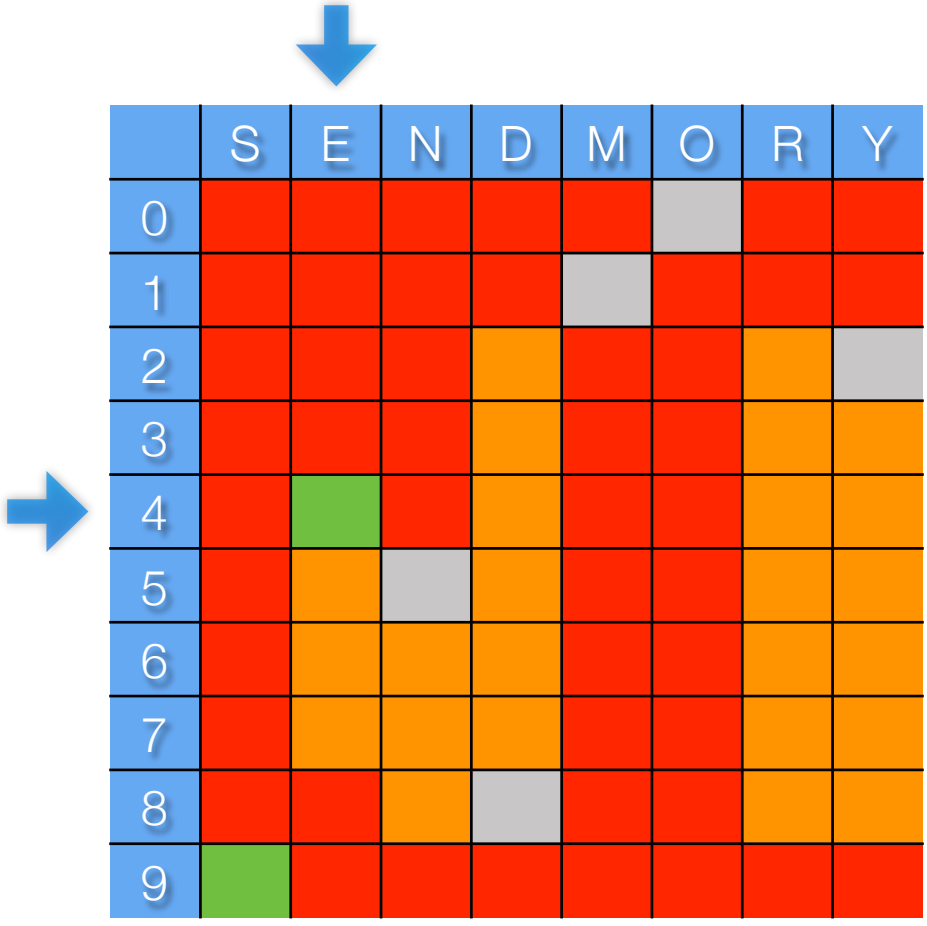
6,360



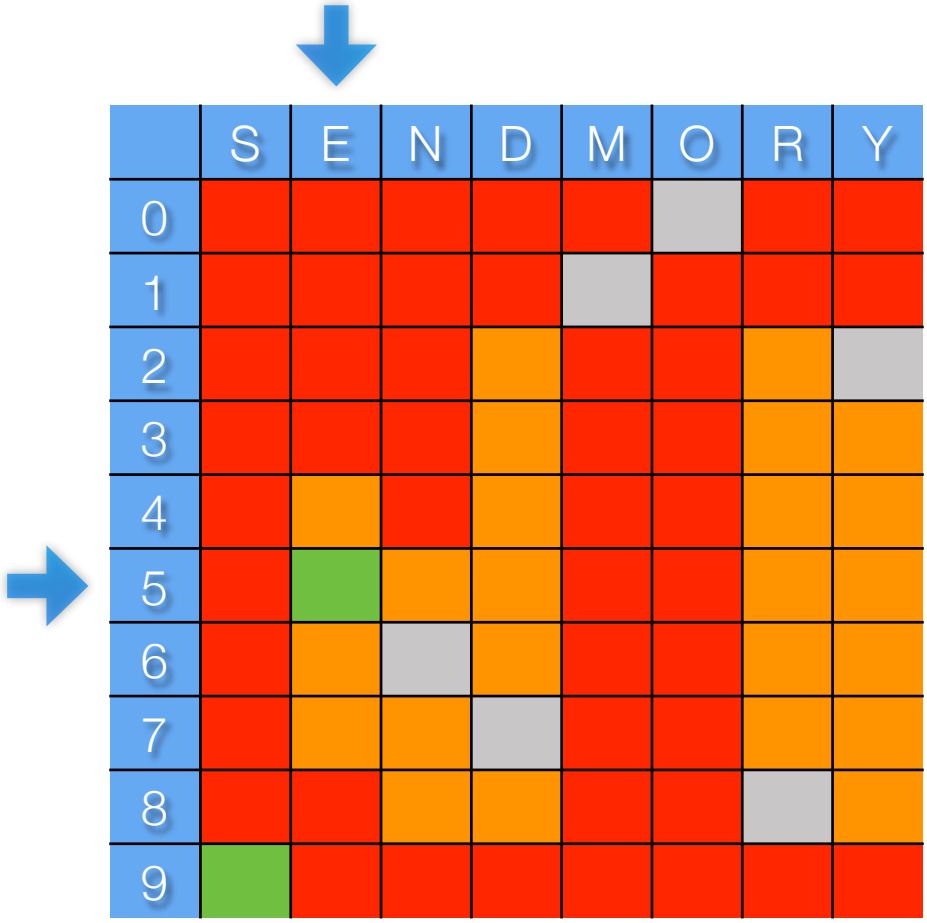
5,488



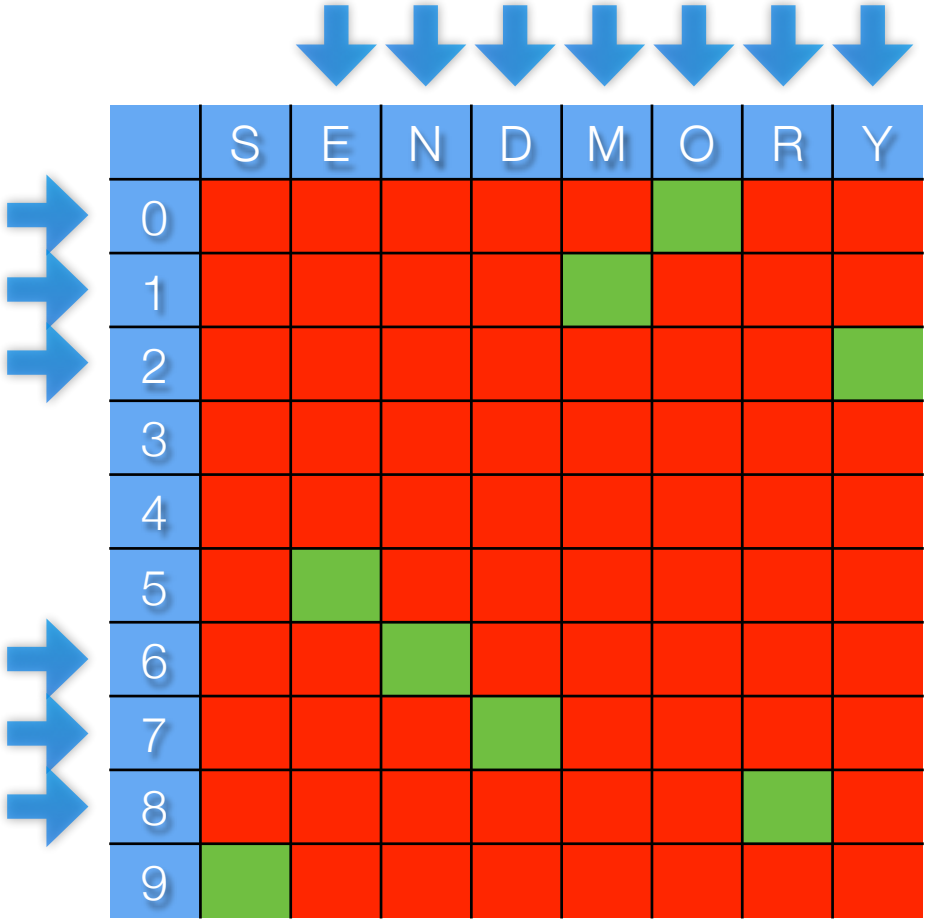
5,488



0



1



1

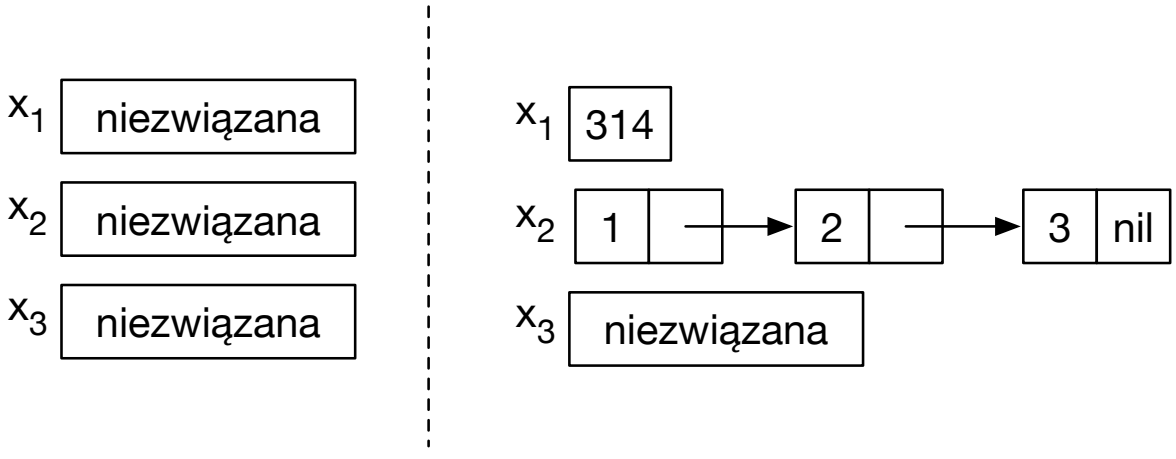
9567
+ 1085
10652

Przegląd paradygmatów na przykładzie języka **Oz**

- programowanie deklaratywne
- współbieżność deklaratywna
- współbieżność z przesyłaniem komunikatów

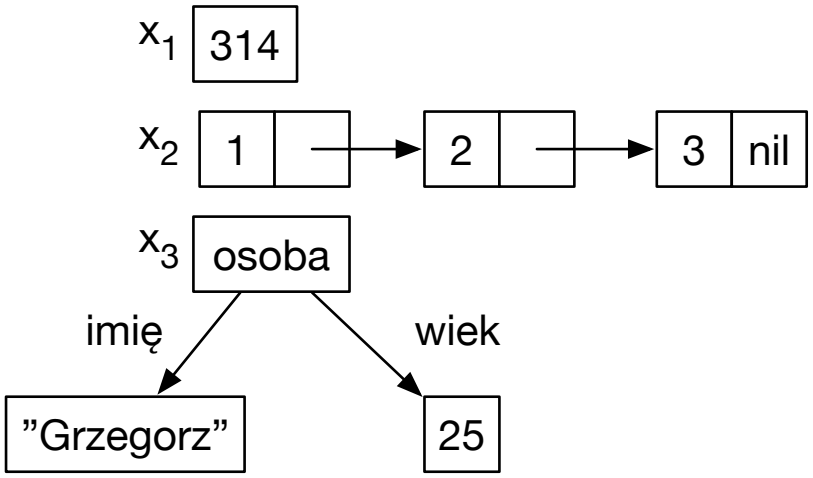
Programowanie deklaratywne

Obszar jednokrotnego przypisania (*single-assignment store*)



$\{x_1=314, x_2=[1\ 2\ 3], x_3\}$

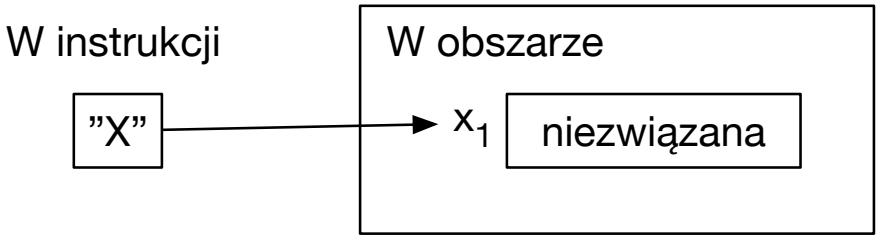
Programowanie deklaratywne



$\{x_1=314, x_2=[1\ 2\ 3], x_3=\text{osoba(imię:”Grzegorz” wiek:24)}\}$

Programowanie deklaratywne

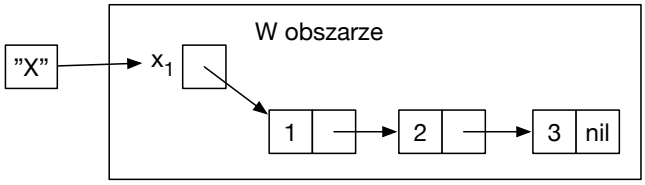
Identyfikator zmiennej umożliwia odwołanie się do wartości z obszaru przypisania.



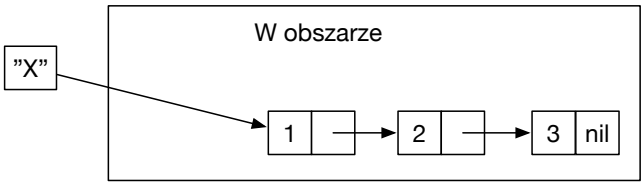
Odwzorowanie identyfikatorów zmiennych na elementy obszaru nazywa się **środowiskiem**.

$\{X \rightarrow x_1\}$

Programowanie deklaratywne



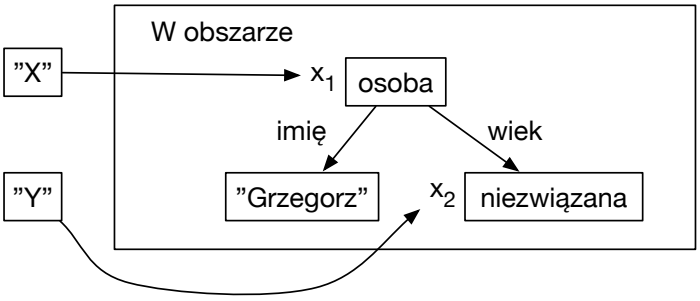
a) Identyfikator zmiennej odwołujący się do zmiennej związanej



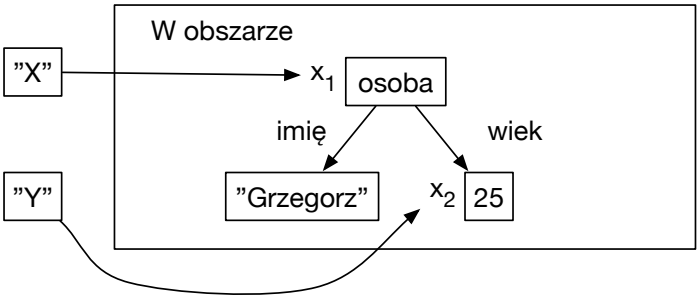
b) Identyfikator zmiennej odwołujący się wartości

Śledzenie łączy zmiennych związanych w celu otrzymania wartości nazywa
się **dereferencją** (wyłuskaniem) i jest niewidoczne dla programisty.

Programowanie deklaratywne

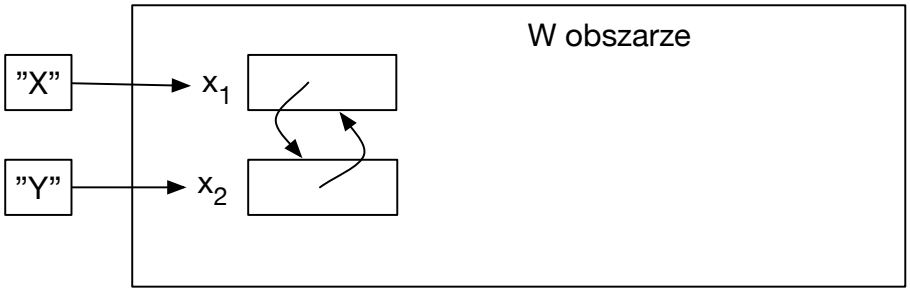


a) Wartość częściowa

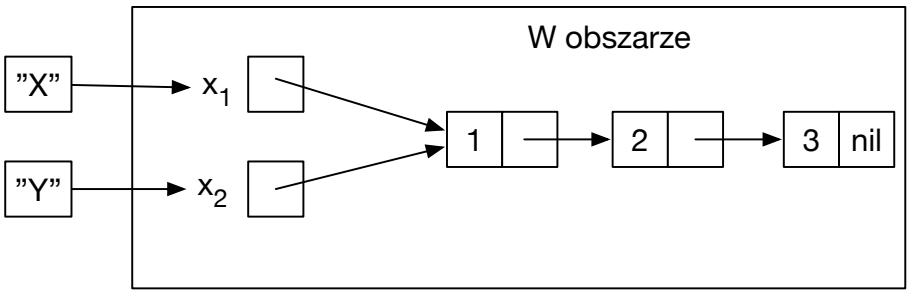


b) Wartość bez zmiennych niezwiązanych czyli wartość pełna

Programowanie deklaratywne



a) Dwie zmienne związane ze sobą



b) Obszar po związaniu jednej ze zmiennych

Programowanie deklaratywne

Po wykonaniu związania **`X=Y`** otrzymujemy dwie zmienne związane.

Mówimy, że $\{x_1, x_2\}$ tworzy zbiór równoważności[1].

Kiedy jedna z równoważnych zmiennych zostaje związana, to wszystkie pozostałe zmienne widzą związanie.

[1] Formalnie: tworzą klasę równoważności ze względu na relację równoważności.

Programowanie deklaratywne

Zmienne przepływu danych

W programowaniu deklaratywnym tworzenie zmiennych i ich wiązanie wykonywane jest osobno. Co kiedy odwołamy się do zmiennej jeszcze nie związanej?

1. Wykonywanie jest kontynuowane bez komunikatu o błędzie. Zawartość zmiennej jest niezdefiniowana, tzn. zawiera „śmieć”. (C++)
2. Wykonywanie jest kontynuowane bez komunikatu o błędzie. Zawartość zmiennej jest inicjowana wartością domyślną. (Java w przypadku pól w obiektach albo tablicach)
3. Wykonywanie jest zatrzymane i pojawia się komunikat o błędzie. (Prolog)
4. Wykonywanie niej jest możliwe ponieważ kompilator wykrył, że istnieje ścieżka wykonania wiodąca do użycia zmiennej bez jej zainicjowania. (Java w przypadku zmiennych lokalnych)
5. Wykonywanie jest wstrzymane do momentu związania zmiennej a później kontynuowane. (Oz)

Programowanie deklaratywne

Język modelowy (*kernel language*)

<s> ::=
skip
| <s>$_1$ <s>$_2$
| **local** <x> **in** <s> **end**
| <x>$_1$=<x>$_2$
| <x>=<v>
| **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
| **case** <x> **of** <pattern> **then** <s>$_1$ **else** <s>$_2$ **end**
| {<x> <y>$_1$ … <y>$_n$}

Programowanie deklaratywne

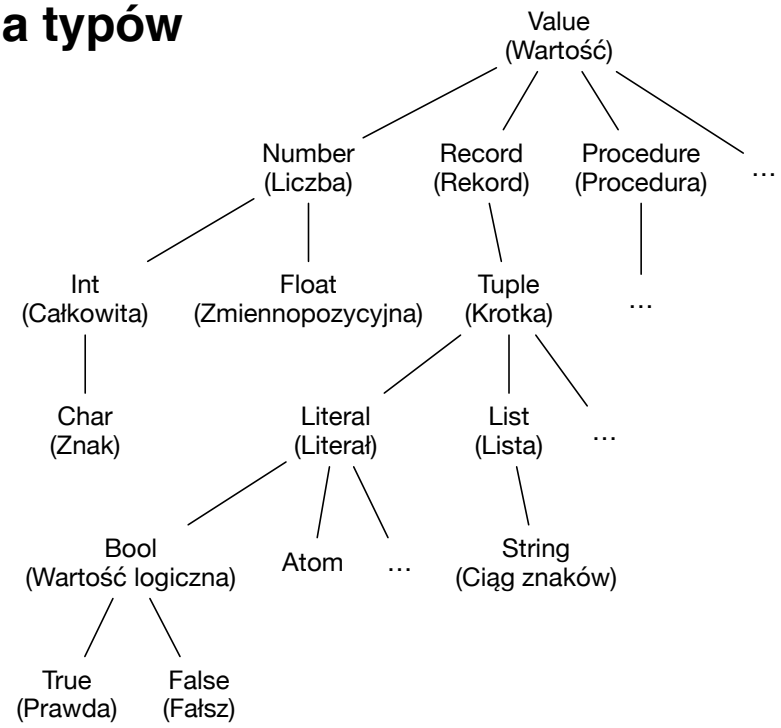
<v> ::= <number> | <record> | <procedure>
<number> ::= <int> | <float>
<record> <pattern> ::= <literal>
  |<literal> (<feature>:<x>$_1$ … <feature>:<x>$_n$)
<procedure> ::= **proc** { $ <x>$_1$ .. <x>$_n$ } <s> **end**
<literal> ::= <atom> | <bool>
<feature> ::= <atom> | <bool> | <int>
<bool> ::= **true** | **false**

Składnia identyfikatora zmiennej:

- Wielka litera, po której występuje zero lub więcej znaków alfanumerycznych (litery, cyfry, znak podkreślenia)
- Dowolny ciąg znaków ujęty w odwrotne apostrofy.

Programowanie deklaratywne

Hierarchia typów



Programowanie deklaratywne

Typy podstawowe

- *Liczby*. Całkowite lub zmiennopozycyjne. Za minus należy używać znak tyldy (~).
- *Atomy*. Niepodzielne wartości będące rodzajem symbolicznych stałych. Ciąg znaków alfanumerycznych rozpoczynających się od małej litery albo dowolny ciąg znaków ujęty w apostrofy.
- *Wartości logiczne*. Symbol **true** albo **false**.
- *Rekordy*. Złożona struktura składająca się z etykiety, po której występuje zbiór par cech i identyfikatorów zmiennych. Cechami mogą być atomy, liczby całkowite lub wartości logiczne.
- *Krotki*. Krotka jest rekordem, którego cechami są liczby całkowite rozpoczynające się od 1 (w tym przypadku cechy nie muszą być podawane).

Programowanie deklaratywne

Typy podstawowe cd.

- *Listy.* Lista jest albo atomem `nil` albo krotką `'|'(H T)`, gdzie `T` jest albo niezwiązane, albo związane z listą. Lukier składniowy:
  - Etykieta `'|'` może być zapisana jako operator wrostkowy, więc `H|T` oznacza to samo co `'|'(H T)`.
  - Operator `'|'` wiąże prawostronnie, więc `1|2|3|nil` oznacza to samo co `1|(2|(3|nil))`.
  - Listy kończące się atomem `nil` mogą być zapisane przy użyciu nawiasów kwadratowych `[…]`, więc zapis `[1 2 3]` oznacza to samo co `1|2|3|nil`.
- *Ciągi znaków*. Ciąg znaków jest listą kodów znaków. Ciągi znaków zapisuje się za pomocą cudzysłowów, więc zapis `"E=mc^2"` oznacza to samo co `[69 61 109 99 94 50]`.

Programowanie deklaratywne

Typy podstawowe cd.

- *Procedury*. Procedura jest wartością typu proceduralnego.

Instrukcja:

<x> = **proc** { $ <y>$_1$ … <y>$_n$ } <s> **end**

wiąże <x> z nową wartością procedury. Oznacza to po prostu zadeklarowanie nowej procedury. Symbol $ określa, że wartość procedury jest anonimowa, tj. tworzona bez związania jej z identyfikatorem. Możliwy jest zapis skrócony:

proc { <x> <y>$_1$ … <y>$_n$ } <s> **end**

Taki zapis skrócony jest czytelniejszy ale zaciemnia rozróżnienie między utworzeniem wartości a związaniem jej z identyfikatorem.

Programowanie deklaratywne

Przykłady operacji podstawowych

Operacja	Opis	Typ argumentu
A==B	Porównanie równości	Value
A\=B	Porównanie nierówności	Value
{IsProcedure P}	Sprawdzenie czy procedura	Value
A<=B	Porównanie mniejszy niż lub równy	Number lub Atom
A<B	Porównanie mniejszy niż	Number lub Atom
A>=B	Porównanie większy niż lub równy	Number lub Atom
A>B	Porównanie większy niż	Number lub Atom
A+B	Dodawanie	Number
A-B	Odejmowanie	Number
A*B	Mnożenie	Number

Programowanie deklaratywne

Przykłady operacji podstawowych cd.

Operacja	Opis	Typ argumentu
A **div** B	Dzielenie	Int
A **mod** B	Modulo	Int
A/B	Dzielenie	Float
{Arity R}	Krotność	Record
{Label R}	Etykieta	Record
R.F	Wybór pola	Record

Programowanie deklaratywne

Proste wykonanie

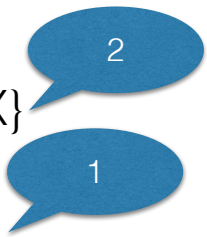
```
local A B C D in
   A=11
   B=2
   C=A+B
   D=C*C
end
```

```
local A in
   local B in
      local C in
         local D in
            A=11
            B=2
            C=A+B
            D=C*C
         end
      end
   end
end
```

Programowanie deklaratywne

Statyczne wyznaczanie zakresu

```
local X in
  X=1
  local X in
    X=2
    {Browse X}
  end
  {Browse X}
end
```



Programowanie deklaratywne

Procedury

```
proc {Max X Y ?Z}
  if X>=Y then Z=X else Z=Y end
end
```

Procedury z odwołaniami zewnętrznymi

```
proc {LB X ?Z}
  if X>=Y then Z=X else Z=Y end
end
```

Programowanie deklaratywne

```
local Y LB in
  Y=10
  proc {LB X ?Z}
    if X>=Y then Z=X else Z=Y end
  end
  local Y=15 Z in
    {LB 5 Z}
  end
end
```

Programowanie deklaratywne

Dynamiczne a statyczne określanie zasięgu

```
local P Q in
  proc {Q X} {Browse stat(X)} end
  proc {P X} {Q X} end
  local Q in
    proc {Q X} {Browse dyn(X)} end
    {P hello}
  end
end
```

stat(hello)

Oryginalna wersja języka Lisp obsługiwała **dynamiczne** wyznaczanie zakresu. Języki Common Lisp i Scheme domyślnie obsługują **statyczne** wyznaczanie zakresu.

Programowanie deklaratywne

Zachowanie w przypadku przepływu danych

```
local X Y Z in
  X=10
  if X>=Y then Z=X else Z=Y end
end
```

Nie jest możliwe określenie wartości porównania X>=Y ale nie jest zgłaszany błąd tylko wykonanie instrukcji warunkowej zostaje wstrzymane do chwili gdy Y zostanie związane.

Programowanie deklaratywne

Przekład na język modelowy

```
local Max C in
  proc {Max X Y ?Z}
    if X>=Y then Z=X else Z=Y end
  end
  {Max 3 5 C}
end
```

Programowanie deklaratywne

Przekład na język modelowy cd.

```
local Max in
  local A in
    local B in
      local C in
        Max = proc {$ X Y Z}
                local T in
                  T=(X>=Y)
                  if T then Z=X else Z=Y end
                end
              end
        A=3
        B=5
        {Max A B C}
      end
    end
  end
end
```

Programowanie deklaratywne

Od języka modelowego do języka praktycznego

Programy w języku modelowym są zbyt rozwlekłe. Można tę wadę wyeliminować dodając **lukier syntaktyczny** i **abstrakcje lingwistyczne**.

Udogodnienia składniowe

Zagnieżdżone wartości częściowe:
zamiast
local A B **in** A=”Grzegorz” B=25 X=osoba(imię:A wiek:B) **end**
wygodniej
X=osoba(imię:”Grzegorz” wiek:25)

Programowanie deklaratywne

Niejawna inicjalizacja zmiennych:
zamiast
local X **in** X=<expression> <statement> **end**
wygodniej
local X=<expression> **in** <statement> end

Przypadek ogólny ma postać
local <pattern>=<expression> **in** <statement> **end**

Przykład:
local tree(key:A left:B right:C value:D)=T **in** <statement> **end**

Programowanie deklaratywne

Zagnieżdżone instrukcje **if**:

<statement> ::= **if** <expression> **then** <inStatement>
{ **elseif** <expression> **then** <inStatement> }
[**else** <inStatement>] **end**
<inStatement> ::= [{ <declarationPart> }+ **in**] <statement>

Programowanie deklaratywne

Zagnieżdżone instrukcje **case**:

<statement> ::= **case** <expression>
of <pattern> [**andthen** <expression>]
then <inStatement>
{ '[]' <pattern> [**andthen** <expression>]
then <inStatement> }
[**else** <inStatement>] **end**
<pattern> ::= <variable> | <atom> | <int> | <float>
| <string> | unit | true | false
| <label> '(' { [<feature> ':'] <pattern> } ['…'] ')'
| <pattern> <consBinOp> <pattern>
| '[' { <pattern> }+ ']'
<consBinOp> ::= '#' | '|'

Programowanie deklaratywne

Przykład:

case Xs#Ys
of nil#Ys **then** <s>$_1$
[] Xs#nil **then** <s>$_2$
[] (X|Xr)#(Y|Yr) **andthen** X<=Y **then** <s>$_3$
else <s>$_4$
end

case Xs **of** nil **then** <s>$_1$
else
case Ys **of** nil **then** <s>$_2$
else
case Xs **of** X|Xr **then**
case Ys **of** Y|Yr **then**
if X<=Y **then** <s>$_3$ **else** <s>$_4$ **end**
else <s>$_4$ **end**
else <s>$_4$ **end**
end
end

Programowanie deklaratywne

Funkcje

fun {F X1 … XN} <statement> <expression> **end**

proc {F X1 … XN ?R} <statement> R=<expression> **end**

Programowanie deklaratywne

Przykład:

```
fun {Max X Y}
  if X>=Y then X else Y end
end
```

jeśli treść funkcji jest instrukcją **if**, każda alternatywa tej instrukcji może kończyć się wyrażeniem

```
proc {Max X Y ?R}
  R = if X>=Y then X else Y end
end
```

Programowanie deklaratywne

Wywołanie funkcji

Wywołanie funkcji {F X1 … XN} jest tłumaczone na wywołanie procedury {F X1 … XN R}.

Przykład:

Wywołanie {Q {F X1 … XN} … } jest tłumaczone na:

```
local R in
  {F X1 … XN R}
  {Q R … }
end
```

Programowanie deklaratywne

Wywołanie funkcji w strukturach danych

Można wywołać funkcję w ramach struktury danych (rekordu, krotki lub listy).

Przykład:
Ys={F X}|{Map Xr F}
jest tłumaczone na:

```
local Y Yr in
  Ys=Y|Yr
  {F X Y}
  {Map Xr F Yr}
end
```

Programowanie deklaratywne

```
fun {Map Xs F}
  case Xs
  of nil then nil
  [] X|Xr then {F X}|{Map Xr F}
  end
end
```


```
proc {Map Xs F ?Ys}
  case Xs of nil then Ys=nil
  else case Xs of X|Xr then
    local Y Yr in
      Ys=Y|Yr {F X Y} {Map Xr F Yr}
    end
  end end
end
```

Programowanie deklaratywne

Interfejs interaktywny (zmienne globalne)

```
<interStatement> ::=
    <statement>
    | declare { <declarationPart> }+ [ <interStatement> ]
    | declare { <declarationPart> }+ in <interStatement>
<declarationPart> ::=
    <variable> | <pattern> '=' <expression> | <statement>
```

Przykład:

```
declare X Y
X=25
declare A
A=osoba(wiek:X)
declare X Y
```

nowa zmienna X ale wartość 25 nadal jest osiągalna przez zmienną globalną A

Programowanie deklaratywne

Wyjątki

<s> ::= …
| **try** <s>$_1$ **catch** <x> **then** <s>$_2$ **end**
| **raise** <x> **end**

- Instrukcja w której wywołano wyjątek zostaje anulowana.
- Wyjątkiem może być każda wartość częściowa.
- Gdy wyjątek jest zmienną niezwiązaną, to jego zgłoszenie może być współbieżne z jego określeniem (może być nawet przechwycony zanim będzie wiadomo, o który wyjątek chodzi!).
- Jest to rozsądne tylko w przypadku języków ze zmiennymi przepływu danych.

Programowanie deklaratywne

Przykład:

```
fun {Eval E}
  if {IsNumber E} then E
  else
    case E
    of plus(X Y) then {Eval X}+{Eval Y}
    [] times(X Y) then {Eval X}*{Eval Y}
    else raise illFormedExpr(E) end
    end
  end
end
```

Programowanie deklaratywne

Przykład cd.

```
try
  {Browse {Eval plus(plus(5 5) 10)}}
  {Browse {Eval times(6 11)}}
  {Browse {Eval minus(7 10)}}
catch illFormedExpr(E) then
  {Browse '*** Błędne wyrażenie '#E#' ***'}
end
```

Programowanie deklaratywne

Instrukcja **try** może określać klauzulę **finally**, która jest zawsze wykonywana bez względu na to, czy instrukcja zgłosi wyjątek czy nie.

Przykład:

```
try
  {ProcessFile F}
finally {CloseFile F} end
```

Programowanie deklaratywne

Algorytm unifikacji, struktury cykliczne i równość

Operacje związania:

- **bind(ES, <v>)** wiąże wszystkie zmienne ze zbiorów równoważnych zmiennych **ES** z wartością **<v>**.
- **bind(ES_1, ES_2)** łączy dwa zbiory równoważnych zmiennych ES_1 i ES_2.

Programowanie deklaratywne

Definicja operacji **unify(x, y)** bez struktur cyklicznych:

1. Jeśli **x** jest w zbiorze ES_x a **y** w zbiorze ES_y, to wykonaj **bind**(ES_x, ES_y).
2. Jeśli **x** jest w zbiorze ES_x a **y** jest określone, to wykonaj **bind**(ES_x, **y**).
3. Jeśli **y** jest w zbiorze ES_y a **x** jest określone, to wykonaj **bind**(ES_y, **x**).
4. Jeśli **x** jest związane z $r(f_1{:}x_1 \ldots f_n{:}x_n)$ a **y** jest związane z $r'(f'_1{:}y_1 \ldots f'_m{:}y_m)$ i $r \neq r'$ lub $\{f_1, \ldots, f_n\} \neq \{f'_1, \ldots, f'_m\}$, to zgłoś wyjątek **failure**.
5. Jeśli **x** jest związane z $r(f_1{:}x_1 \ldots f_n{:}x_n)$ a **y** jest związane z $r(f_1{:}y_1 \ldots f_n{:}y_n)$, to dla **i** od **1** do **n** wykonaj **unify**(x_i, y_i).

Programowanie deklaratywne

Uwzględnienie struktur cyklicznych.

- Niech **M** będzie pustą tabelą.
- Wykonaj **unify”(x, y)**.

Operacja **unify”(x, y)** najpierw sprawdza czy para **(x, y)** występuje w tabeli **M**.

1. Jeśli występuje, to kończy pracę.
2. Jeśli nie występuje, to wywołuje operację **unify(x, y)**, w której rekurencyjne wywołania **unify** zastąpiono wywołaniami **unify”**.

Programowanie deklaratywne

Sprawdzenie równości i nierówności

X==Y zachodzi gdy obie struktury są identyczne.
Możliwe jest wcześniejsze stwierdzenie, że zachodzi X\=Y nawet gdy obie struktury są jeszcze niepełne.

declare L1 L2 X **in**
L1=[1]
L2=[X]
{Browse L1==L2}

nic się nie pojawia bo czeka na związanie X

declare L1 L2 X Y **in**
L1=[X]
L2=[Y]
{Browse L1==L2}
X=Y

po unifikacji pojawia się **true**

declare L1 L2 X **in**
L1=[1 2]
L2=[2 X]
{Browse L1==L2}

false

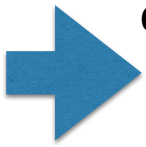
od razu pojawia się **false**

Programowanie deklaratywne

Techniki programowania: obliczenia iteracyjne

$$S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_{final}$$

```
fun {Iterate Si}
  if {IsDone Si} then Si
  else Si+1 in
    Si+1={Transform Si}
    {Iterate Si+1}
  end
end
```



```
fun {Iterate S IsDone Transform}
  if {IsDone S} then S
  else S1 in
    S1={Transform S}
    {Iterate S1 IsDone Transform}
  end
end
```

Programowanie deklaratywne

Techniki programowania: obliczenia iteracyjne

```
fun {Sqrt X}
  {Iterate
    1.0
    fun {$ G} {Abs X-G*G}/X<0.00001 end
    fun {$ G} (G+X/G)/2.0 end}
end
```

Programowanie deklaratywne

Techniki programowania: konwersja rekurencji na iterację

```
fun {Fact N}
  if N==0 then 1
  else N*{Fact N-1}
  end
end
```

```
fun {Fact N}
  fun {IterFact N Acc}
    if N==0 then Acc
    else {IterFact N-1 N*Acc}
    end
  end
in
  {IterFact N 1}
end
```

Programowanie deklaratywne

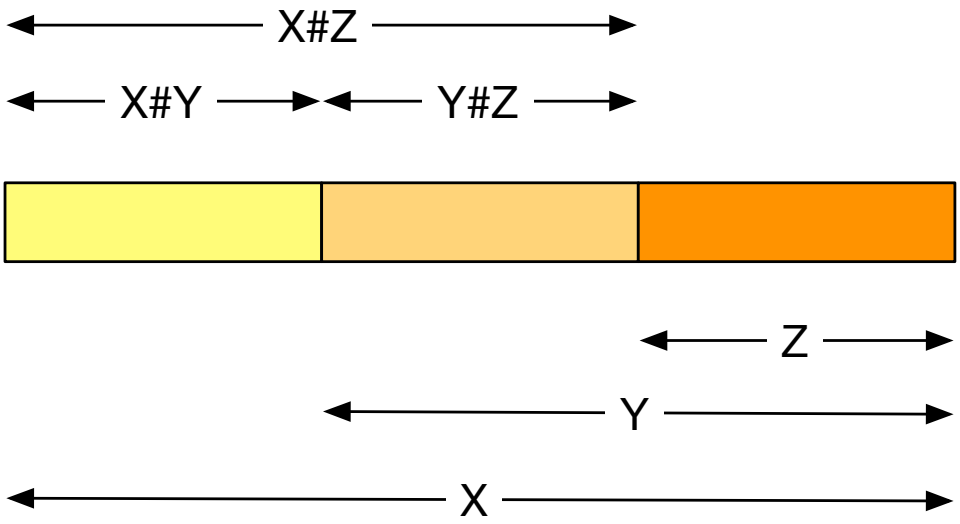
Techniki programowania: operacje na listach

```
fun {Length Xs}
  case Xs of nil then 0
  [] _|Xr then 1+{Length Xr}
  end
end
```

```
fun {Length Xs}
  fun {LengthIter Xs Acc}
    case Xs of nil then Acc
    [] _|Xr then {LengthIter Xr Acc+1}
    end
  end
in
  {LengthIter Xs 0}
end
```

Programowanie deklaratywne

Techniki programowania: listy różnicowe



Programowanie deklaratywne

Techniki programowania: listy różnicowe

```
fun {AppendD D1 D2}
   X#Y=D1
   Y#Z=D2
in
   X#Z
end
```

```
declare D1 D2 X Y in
D1=(1|2|3|X)#X
D2=(4|5|6|7|Y)#Y
{Browse {AppendD D1 D2}}
```

(1|2|3|4|5|6|7|_)#_

Programowanie deklaratywne

Techniki programowania: uporządkowane drzewa binarne

```
fun {Insert K V T}
  case T
  of leaf then tree(K V leaf leaf)
  [] tree(K1 V1 T1 T2) andthen K==K1 then
       tree(K V T1 T2)
  [] tree(K1 V1 T1 T2) andthen K<K1 then
       tree(K1 V1 {Insert K V T1} T2)
  [] tree(K1 V1 T1 T2) andthen K>K1 then
       tree(K1 V1 T1 {Insert K V T2})
  end
end
```

Programowanie deklaratywne

Techniki programowania: uporządkowane drzewa binarne

```
fun {Lookup K T}
  case T
  of leaf then notfound
  [] tree(K1 V T1 T2) andthen K==K1 then found(V)
  [] tree(K1 V T1 T2) andthen K<K1 then {Lookup K T1}
  [] tree(K1 V T1 T2) andthen K>K1 then {Lookup K T2}
 end
end
```

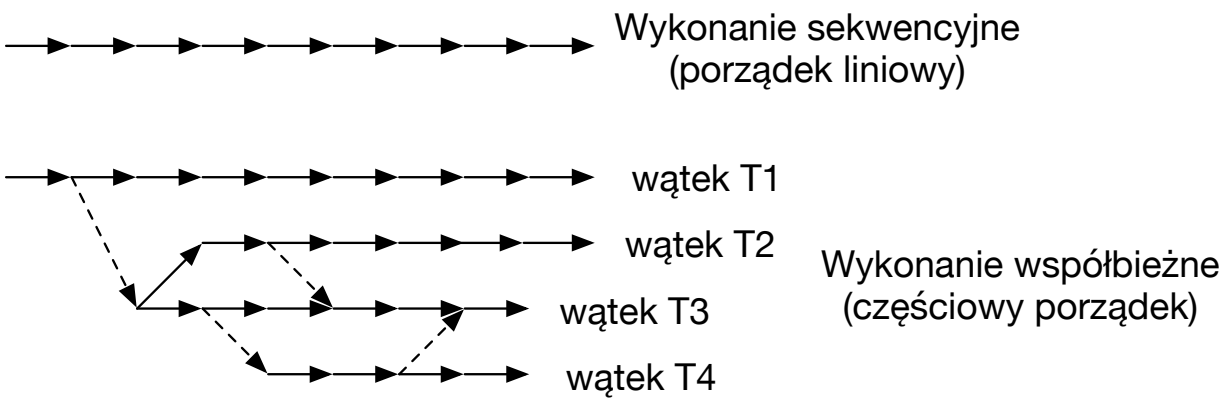
Współbieżność deklaratywna

Język modelowy

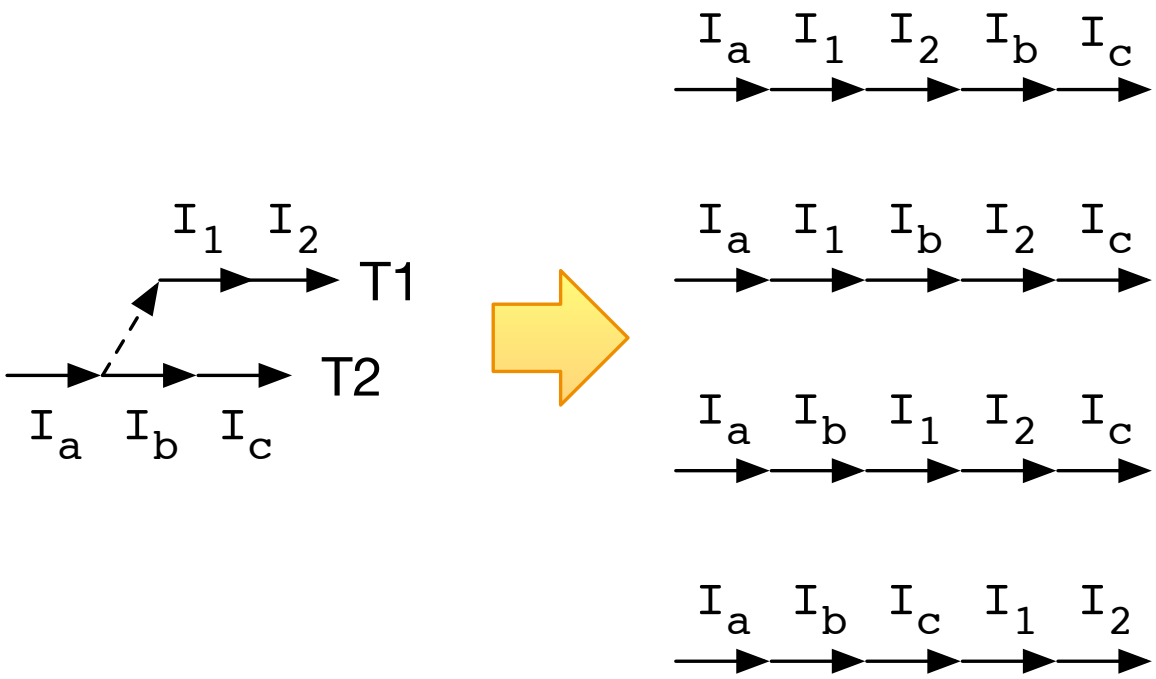
<s> ::=
skip
| <s>$_1$ <s>$_2$
| **local** <x> **in** <s> **end**
| <x>$_1$=<x>$_2$
| <x>=<v>
| **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
| **case** <x> **of** <pattern> **then** <s>$_1$ **else** <s>$_2$ **end**
| {<x> <y>$_1$ … <y>$_n$}
| **thread** <s> **end**

Współbieżność deklaratywna

Porządek przyczynowy



Współbieżność deklaratywna



Współbieżność deklaratywna

Częściowe zakończenie

```
fun {Double Xs}
   case Xs of X|Xr then 2*X|{Double Xr} end
end
```

- Obliczenia powyższej funkcji nigdy się nie zakończą.
- Jeśli strumień wejściowy **Xs** przestanie rosnąć, to obliczenie się zakończy.
- Obliczenie osiągnie **częściowe zakończenie**, ponieważ jeśli znowu strumień wejściowy zacznie znowu rosnąć, to obliczenia zostaną wznowione aż do następnego częściowego zakończenia.

Współbieżność deklaratywna

Logiczna równoważność

X=1 Y=X ≡ Y=X X=1

w obu przypadkach X i Y są związane z wartością 1

X=foo(Y W) Y=Z ≡ X=foo(Z W) Y=Z

Zbiór wiązań w obszarze jednokrotnego przypisania nazywamy ograniczeniem. Niech *values*(*x*, *c*) będzie zbiorem wartości dla zmiennej *x* w ograniczeniu *c*, które nie naruszają ograniczenia *c*.

Ograniczenia c_1 i c_2 są logicznie równoważne, gdy:
1. Zawierają te same zmienne.
2. Dla każdej zmiennej *x*, *values*(*x*, c_1)=*values*(*x*,c_2).

Współbieżność deklaratywna

Deklaratywna współbieżność

Współbieżny program jest **deklaratywny** jeśli dla dowolnych wejść zachodzi poniższy warunek.

Wszystkie wykonania, dla danego zbioru wejść, mają jeden z dwóch rezultatów:
1. wszystkie nie kończą się lub
2. wszystkie kończą się osiągając częściowe zakończenie i dają logicznie równoważne rezultaty.

niezauważalny niedeterminizm

Współbieżność deklaratywna

Awaria (*failure*)

Awaria jest nieprawidłowym zakończeniem deklaratywnego programu osiągnięte przez umieszczenie niezgodnych (konfliktowych) informacji w obszarze przypisania.

thread X=1 **end**
thread Y=2 **end**
thread X=Y **end**

każde wykonanie osiąga niepowodzenie

Współbieżność deklaratywna

Zamknięcie awarii (ukrywanie niedeterminizmu)

```
declare X Y
local X1 Y1 S1 S2 S3 in
   thread
      try X1=1 S1=ok catch _ then S1=error end
   end
   thread
      try Y1=2 S2=ok catch _ then S2=error end
   end
   thread
      try X1=Y1 S3=ok catch _ then S3=error end
   end
   if S1==error orelse S2==error orelse S3=error then
      X=1  Y=1 % wartości domyślne dla X i Y
   else
      X=X1 Y=Y1 end
end
```

Współbieżność deklaratywna

Tworzenie nowego wątku

```
thread
   proc {Count N} if N>0 then {Count N-1} end end
in
   {Count 1000000}
end
```

```
declare X in
local Y in
   thread Y = 10*10 end
   X = Y + 100*100
end
{Browse X}
```

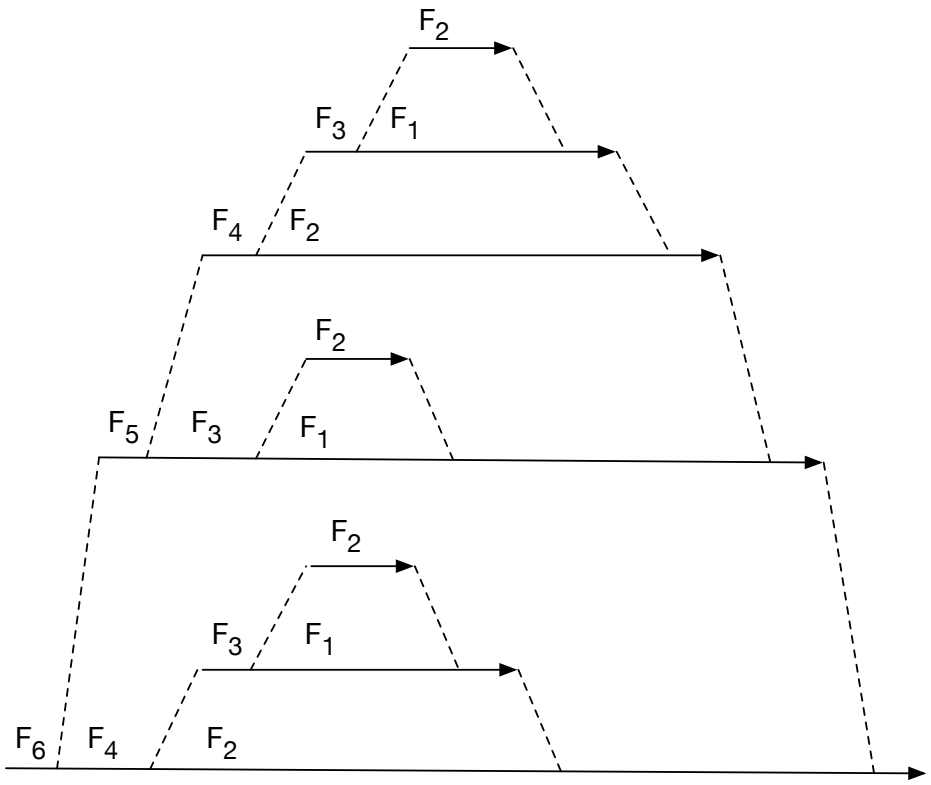
```
declare X in
X = thread 10*10 end + 100*100
{Browse X}
```

Współbieżność deklaratywna

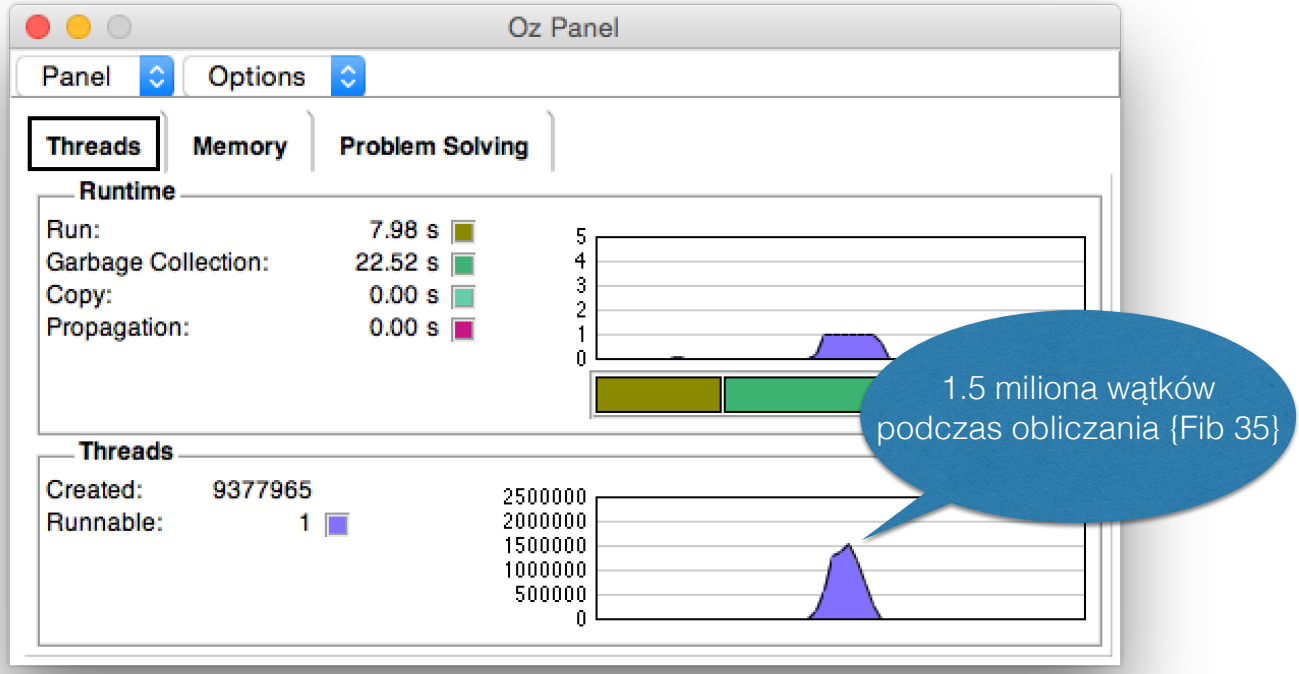
Przykład: liczby Fibonacciego

```
fun {Fib X}
   if X=<2 then 1
   else thread {Fib X-1} end + {Fib X-2} end
end
```

Współbieżność deklaratywna



Współbieżność deklaratywna



Współbieżność deklaratywna

Strumienie w postaci list otwartych

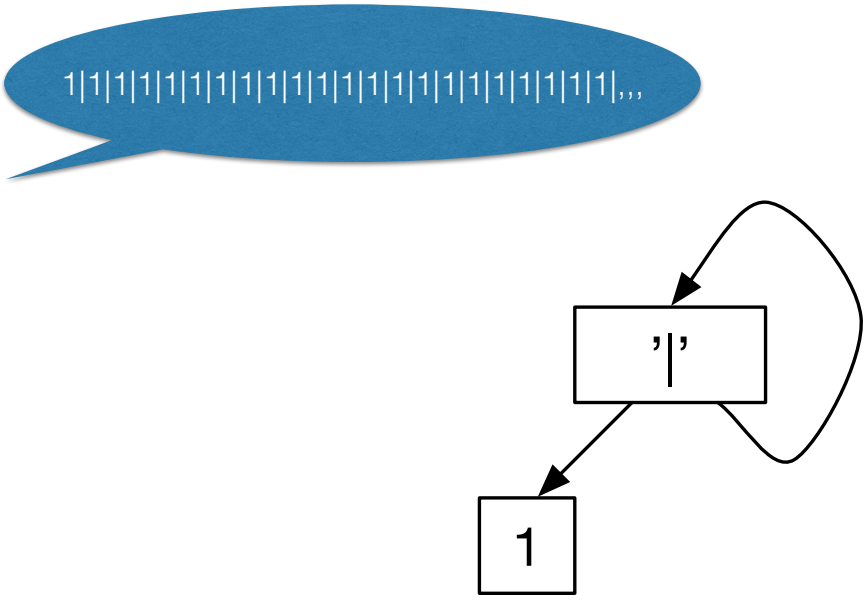
X=1|X1
X1=2|X2
X2=3|X3
X3=4|X4
X4=nil

Zmienna X zostaje związana z listą, na której początkowo jest tylko liczba 1. Później dochodzą jeszcze liczby 2, 3 i 4. Na koniec lista zostaje zamknięta (nie wa nieustalonego ogona).

Współbieżność deklaratywna

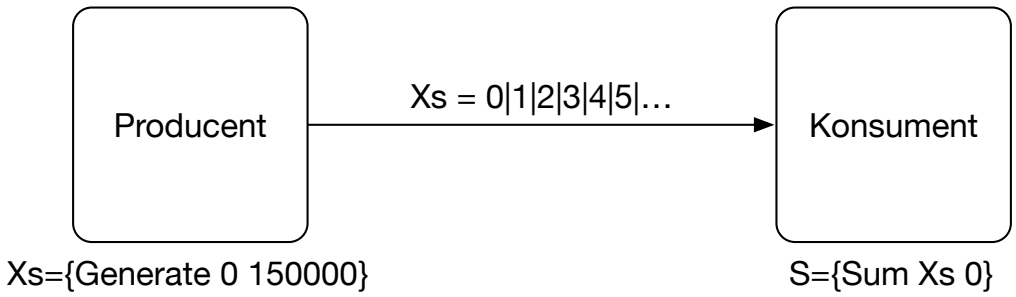
Nieskończony strumień liczb

X=1|X
{Browse X}



Współbieżność deklaratywna

Schemat producenta-konsumenta



Współbieżność deklaratywna

```
fun {Generate N Limit}
   if N<Limit then
      N|{Generate N+1 Limit}
   else nil
end
fun {Sum Xs A}
   case Xs
   of X|Xr then {Sum Xr A+X}
   [] nil then A
   end
end
local Xs S in
   thread Xs={Generate 0 150000} end % wątek producenta
   thread S={Sum Xs 0} end           % wątek konsumenta
   {Browse S}
end
```

Współbieżność deklaratywna

Odczyt wielokrotny

```
local Xs S1 S2 S3 in
   thread Xs={Generate 0 150000} end
   thread S1={Sum Xs 0} end
   thread S2={Sum Xs 0} end
   thread S3={Sum Xs 0} end
end
```

wątek każdego konsumenta pobiera elementy strumienia niezależnie

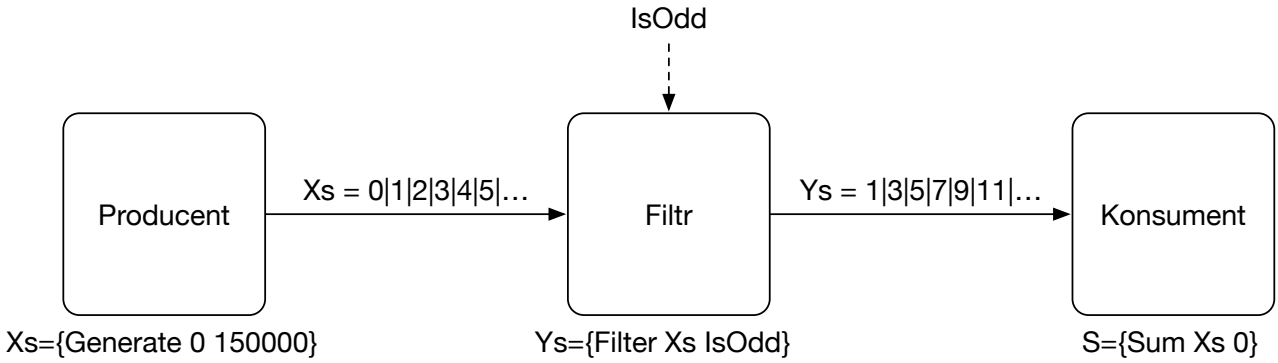
Współbieżność deklaratywna

Przetworniki i filtry

```
fun {Filter Xs F}
   case Xs
   of nil then nil
   [] X|Xr andthen {F X} then X|{Filter Xr F}
   [] X|Xr then {Filter Xr F}
   end
end
```

```
fun {IsOdd X} X mod 2 \= 0 end
```

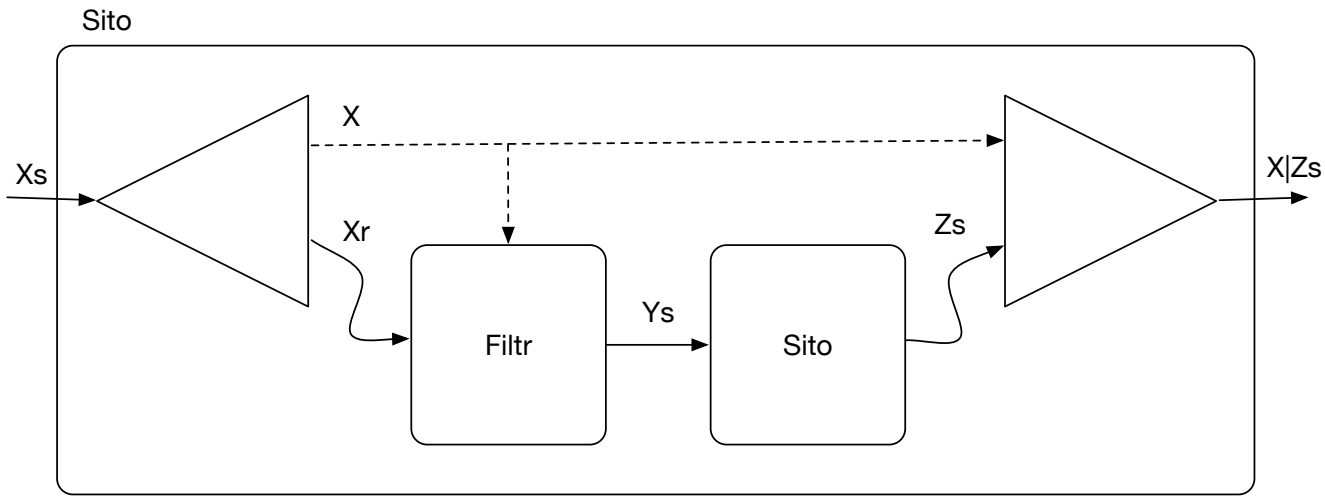
Współbieżność deklaratywna




```
local Xs Ys S in
   thread Xs={Generate 0 150000} end
   thread Ys={Filter Xs IsOdd} end
   thread S={Sum Ys 0} end
   {Browse S}
end
```

Współbieżność deklaratywna

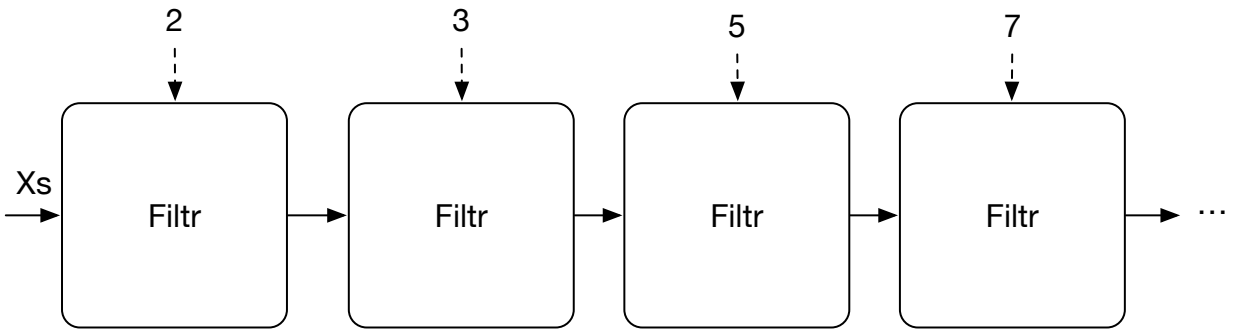
Sito Eratostenesa



Współbieżność deklaratywna

```
fun {Sieve Xs}
   case Xs
   of nil then nil
   [] X|Xr then Ys in
      thread Ys={Filter Xr fun {$ Y} Y mod X \= 0 end} end
      X|{Sieve Ys}
   end
end
```

Współbieżność deklaratywna



Potok filtrów

Współbieżność deklaratywna

Wykonywanie leniwe

- ewaluacja gorliwa *(data-driven evaluation*)
- ewaluacja leniwa (*demand-driven evaluation*)

Współbieżność deklaratywna

Wykonywanie leniwe

```
fun lazy {F1 X} 1+X*(3+X*(3+X)) end
fun lazy {F2 X} Y=X*X in Y*Y end
fun lazy {F3 X} (X+1)*(X+1) end
A={F1 10}
B={F2 20}
C={F3 30}
D=A+B
```

Współbieżność deklaratywna

Wyzwalacze potrzeby

<s> ::=
skip
| <s>$_1$ <s>$_2$
| **local** <x> **in** <s> **end**
| <x>$_1$=<x>$_2$
| <x>=<v>
| **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
| **case** <x> **of** <pattern> **then** <s>$_1$ **else** <s>$_2$ **end**
| {<x> <y>$_1$ … <y>$_n$}
| **thread** <s> **end**
| {ByNeed <x> <y>}

utworzenie wyzwalacza

Współbieżność deklaratywna

Instrukcja **{ByNeed P Y}** ma ten sam efekt, co instrukcja **thread** {P Y} **end** ale wywołanie **{P X}** zostanie wykonane tylko wtedy, gdy potrzebna jest wartość **Y**.

```
declare Y
{ByNeed proc {$ A} A=111*111 end Y}
{Browse Y}
declare Z
Z=Y+1
```

w oknie Browser pojawia się wartość 12321

Współbieżność deklaratywna

Problem Hamminga

Wygenerować pierwszych *n* liczb całkowitych postaci $2^a3^b5^c$, gdzie $a, b, c \geq 0$.

Idea: generować liczby całkowite w porządku rosnącym w potencjalnie nieskończony strumieniu.

```
fun lazy {Times N H}
   case H of X|H2 then N*X|{Times N H2} end
end
```

dla nieskończonego strumienia liczb H, wartością jest nieskończony strumień liczb N razy większych

Współbieżność deklaratywna

```
fun lazy {Merge Xs Ys}
   case Xs#Ys of (X|Xr)#(Y|Yr) then
      if X<Y then X|{Merge Xr Ys}
      elseif X>Y then Y|{Merge Xs Yr}
      else X|{Merge Xr Yr}
      end
   end
end
```

leniwe scalenie dwóch uporządkowanych i nieskończonych strumieni liczb w jeden nieskończony i uporządkowany strumień liczb

Współbieżność deklaratywna

```
H=1|{Merge {Times 2 H}
           {Merge {Times 3 H}
                  {Times 5 H}}}
{Browse H}
```

wyświetli się 1|_<Future>

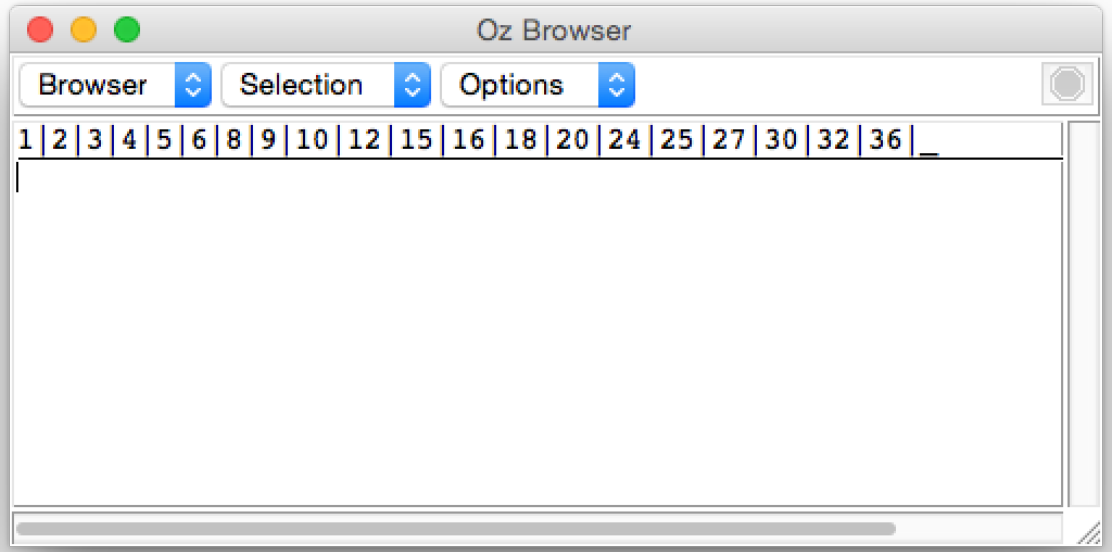
H jest nieskończonym i uporządkowanym strumieniem liczb postaci $2^a3^b5^c$

```
proc {Touch N H}
   if N>0 then {Touch N-1 H.2} else skip end
end
```

ogon listy H

```
{Touch 20 H}
```

Współbieżność deklaratywna



Współbieżność z przesyłaniem komunikatów

- Przesyłanie komunikatów
- Abstrakcja tworzenia portu
- Przykład

Współbieżność z przesyłaniem komunikatów

Język modelowy

<s> ::=
skip
| <s>$_1$ <s>$_2$
| **local** <x> **in** <s> **end**
| <x>$_1$=<x>$_2$
| <x>=<v>
| **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
| **case** <x> **of** <pattern> **then** <s>$_1$ **else** <s>$_2$ **end**
| {<x> <y>$_1$ … <y>$_n$}
| **thread** <s> **end**
| {NewPort <y> <x>}
| {Send <x> <y>}

Współbieżność z przesyłaniem komunikatów

Operacje tworzenia kanału i wysyłania do niego:

- `{NewPort S P}` utworzenie nowego portu z punktem wejścia P i strumieniem S.
- `{Send P X}` asynchroniczne wysłanie X do strumienia odpowiadającego punktowi wejścia P.

```
declare S P in
{NewPort S P}
{Browse S}
{Send P a}
{Send P b}
```


```
S<future>
a|_<future>
a|b|_<future>
```

Współbieżność z przesyłaniem komunikatów

Abstrakcja tworzenia portu

```
fun {NewPortObject Init Fun}
Sin Sout in
   thread {FoldL Sin Fun Init Sout} end
   {NewPort Sin}
end
```

- Init — stan początkowy
- Fun — funkcja zmiany stanu

$$\text{Fun} : \text{STAN} \times \text{KOMUNIKAT} \mapsto \text{STAN}$$

Współbieżność z przesyłaniem komunikatów

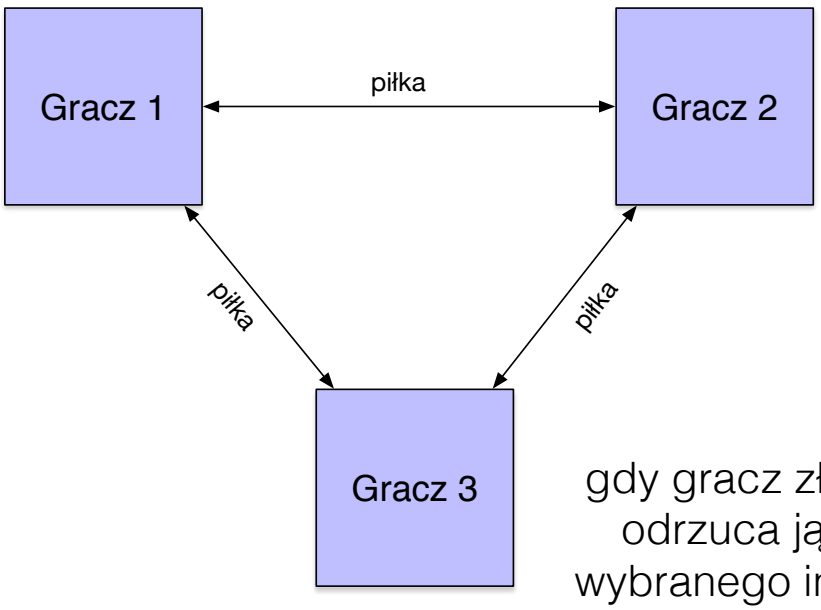
Abstrakcja tworzenia portu

```
fun {NewPortObject2 Proc}
Sin in
   thread for Msg in Sin do {Proc Msg} end end
   {NewPort Sin}
end
```

- Proc — procedura wywoływana dla każdego komunikatu

Współbieżność z przesyłaniem komunikatów

Przykład



gdy gracz złapie piłkę, to odrzuca ją do losowo wybranego innego gracza

Współbieżność z przesyłaniem komunikatów

```
declare NewPortObject2 Player P1 P2 P3 in
fun {NewPortObject2 Proc}
Sin in
   thread for Msg in Sin do {Proc Msg} end end
   {NewPort Sin}
end
fun {Player Others}
   {NewPortObject2
     proc {$ Msg}
        case Msg of ball then
           Ran={OS.rand} mod {Width Others} + 1
        in
           {Send Others.Ran ball}
        end
     end}
end
```

Współbieżność z przesyłaniem komunikatów

```
P1={Player others(P2 P3)}
P2={Player others(P1 P3)}
P3={Player others(P1 P2)}
```

utworzenie graczy

```
{Send P1 ball}
```

zainicjowanie gry

Elementy języka **Scheme**

- Historia języka Lisp
- Wyrażenia i ewaluacja wyrażeń
- Identyfikatory i wyrażenie **let**
- Wyrażenia **lambda**
- Definicje globalne
- Wyrażenia warunkowe
- Przypisanie
- Kontynuacje

Historia języka Lisp

- 1958 **John McCarthy** (MIT) rozpoczęcie prac nad obliczeniami symbolicznymi
- 1960 **Lisp 1**
- 1962 **Lisp 1.5**
- 1975 **Scheme**
- 1984 **Common Lisp**
- 2007 **Clojure**

Wyrażenia i ewaluacja wyrażeń

Interakcja z językiem Scheme

```
42 => 42
22/7 => 22/7
3.1415 => 3.1415
+ => #<procedure>
(+ 76 31) => 107
'halo => halo
halo => Unbound variable: halo
'(a b c d) => (a b c d)
(car '(a b c)) => a
(cdr '(a b c)) => (b c)
(cons 'a '(b c)) => (a b c)
```

```
(cons (car '(a b c))
      (cdr '(a b c))) => (a b c)

(define kwadrat
  (lambda (n)
    (* n n)))
(kwadrat 5) => 25
(kwadrat 1/2) => 1/4
```

Załóżmy, że w pliku **odwrotna.ss** znajduje się następujący kod:

```
(define odwrotna (lambda (n) (if (= n 0)
"ojej!" (/ 1 n))))

(load "odwrotna.ss")
(odwrotna 10) => 1/10
(odwrotna 1/10) => 10
(odwrotna 0) => "ojej!"
```

Wyrażenia proste

```
(+ 1/2 1/2) => 1
(- 1.5 1/2) => 1.0
(* 3 1/2) => 3/2
(/ 1.5 3/4) => 2.0
(+ (+ 2 2) (+ 2 2)) => 8
(- 2 (* 4 1/3)) => 2/3
(quote (1 2 3 4 5)) => (1 2 3 4 5)
(quote (+ 3 4)) => (+ 3 4)
(car '(a b c)) => a
(cdr '(a b c)) => (b c)
(cons 'a '(b c)) => (a b c)
(cons 'a 'b) => (a . b)
(list 'a 'b 'c) => (a b c)
(list 'a) => (a)
(list) => ()
```

Ewaluacja wyrażeń

- Obiekt stały (liczby, napisy) ma wartość identyczną z nim samym.
- `(procedura arg1 arg2 … argN)`
  1. obliczana jest wartość `procedura`
  2. obliczana są wartości `arg1`, `arg2`, …, `argN`
  3. stosuje się wartość procedury do wartości argumentów `arg1`, `arg2`, …, `argN`
- `(quote obiekt)` ma wartość `obiekt` (dokładnie ją „cytuje”)

Przykład:

```
((car (list + - * /)) 2 3) => 5
```

Identyfikatory i wyrażenie **let**

```
(let ((x 2)) (+ x 3)) => 5
```

niech x ma wartość 2 w wyrażeniu (+ x 3)

```
(let ((x1 w1))
  (let ((x2 w2))
    …
      (let ((xn wn))
        …))…)
```

```
(let* ((x1 w1)
       (x2 w2)
       …
       (xn wn))
  …)
```

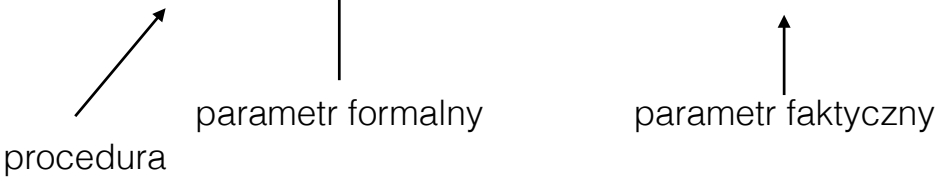
wartości w1, …, wn są obliczane i wiązane ze zmiennymi x1, …, xn jedna po drugiej

```
(let ((x1 w1)
      (x2 w2)
      …
      (xn wn))
  …)
```

wartości w1, …, wn są obliczane i wiązane ze zmiennymi x1, …, xn jednocześnie

Wyrażenia **lambda**

```
(lambda (x) (+ x x)) => #<procedure>

((lambda (x) (+ x x)) (* 3 4)) => 24
```




```
(let ((podwojone (lambda (x) (+ x x))))
  (list (podwojone (* 3 4))
        (podwojone (/ 99 11))
        (podwojone (- 2 7))))  => (24 18 -10)
```

```
(let ((f (lambda x x)))
  (f 1 2 3 4))                    => (1 2 3 4)

(let ((g (lambda (x . y) (list x y))))
  (g 1 2 3 4))                    => (1 (2 3 4))

(let ((h (lambda (x y . z) (list x y z))))
  (h 1 2 3 4))                    => (1 2 (3 4))
```

Definicje globalne

```
(define podwojone-coś (lambda (f x) (f x x)))
(podwojone-coś + 10) => 20
(podwojone-coś cons 'a) => (a . a)

(define kanapka "masło orzechowe z dżemem")
kanapka => "masło orzechowe z dżemem"

(define list (lambda x x))
(define cadr (lambda (x) (car (cdr x))))
(define cddr (lambda (x) (cdr (cdr x))))
(cadr '(a b c)) => b
(cddr '(a b c)) => (c)
```

drugi element listy

ogon ogona listy

```
(define podwajacz
  (lambda (f)
    (lambda (x)
      (f x x))))
(define podwojone (podwajacz +))
(podwojone 13/2) => 13

(define podwojone-cons (podwajacz cons))
(podwojone-cons 'a) => (a . a)

(define podwojone-coś
  (lambda (f x)) ((podwajacz f) x)))
```

Przykład:

```
(define g (lambda (f x) (f x x)))

(g g g) = ((lambda (f x) (f x x)) g g)
        = (g g g)
        = …
```

nieskończone obliczenia

Wyrażenia warunkowe

```
(define abs
  (lambda (n)
    (if (< n 0)
        (- 0 n)
        n)
  )
)
```

forma składniowa
(to nie jest wywołanie procedury)

Wybrane warunki

```
(< -1 0) => #t
(> -1 0) => #f
(not #t) => #f
(not #f) => #t
(not '()) => #t
(not '(a b)) => #f
(or) => #f
(or #f) => #f
(of #f #t) => #t
(or #f 'a #f) => a
(and) => #t
(and #t) => #t
(and #f) => #f
(and #t 'a) => a
(and #t #f 'a) => #f
```

lista pusta jest fałszem

każda wartość różna od listy pustej i #f jest prawdą

wartość pierwszego prawdziwego argumentu lub fałsz

wartość ostatniego argumentu lub fałsz

Przykład:

```
(define odwrotna
  (lambda (n)
    (and (not (= n 0)) (/ 1 n))))

(odwrotna 1/10) => 10
(odwrotna 0) => #f
```

```
(null? '()) => #t
(null? 'abc) => #f
(null? '(a b c)) => #f
(equal? 'a 'a) => #t
(equal? 'a 'b) => #f
(equal? '(a b c) '(a b c)) => #t
(pair? '(a . c)) => #t
(pair? 'a) => #f
(pair? '(a b c)) => #t
```

bo lista jest parą głowy i ogona
(a . (b . (c . ())))

```
(cond (test1 wyrażenie1)
      (test2 wyrażenie2)
      …
      (else domyślne))
```

testy sprawdzane są po kolei i jeśli któryś jest prawdą, to wartością jest wartość odpowiadającego mu wyrażenia, w przeciwnym przypadku wartość wyrażenia domyślnego

Przykład:

```
(define podatek
  (lambda (dochód)
    (cond ((<= dochód 3091)
           0)
          ((<= dochód 85528)
           (- (* 0.18 dochód) 556.20))
          (else
           (+ 14839.02
              (* 0.32 (- dochód 85528)))))))
```

```
(define member
  (lambda (x ls)
    (cond ((null? ls) #f)
          ((equal? (car ls) x) ls)
          (else (member x (cdr ls))))))

(member 'b '(a b b d)) => (b b d)
(member 'c '(a b b d)) => #f

(define remove
  (lambda (x ls)
    (cond ((null? ls) '())
          ((equal? (car ls) x)
           (remove x (cdr ls)))
          (else (cons (car ls)
                      (remove x (cdr ls)))))))

(remove 'b '(a b b d)) => (a d)
```

Przypisanie

```
(define abcde '(a b c d e))
abcde => (a b c d e)
(set! abcde (cdr abcde))
abcde => (b c d e)
```

```
(define licznik1
  (lambda ()
    (let ((x 0))
      (set! x (+ x 1))
      x)))

(licznik1) => 1
(licznik1) => 1
…
```

za każdym razem x jest inicjowane wartością 0 więc za każdym razem zwracana jest wartość 1

```
(define akumulator 0)
(define licznik2
  (lambda ()
    (let ((x akumulator))
      (set! akumulator (+ akumulator 1))
      x)))

(licznik2) => 0
(licznik2) => 1
(licznik2) => 2
…
```

tylko jeden licznik bo
tylko jeden globalny akumulator

```
(define nowy-licznik
  (lambda ()
    (let ((akumulator 0))
      (lambda ()
        (let ((x akumulator))
          (set! akumulator (+ akumulator 1))
          x)))))

(define licz1 (nowy-licznik))
(define licz2 (nowy-licznik))
(licz1) => 0
(licz1) => 1
(licz2) => 0
(licz1) => 2
…
```

każdy licznik ma swój lokalny akumulator

Kontynuacje

- Kontynuacja zawsze jest dla danego podwyrażenia w kontekście szerszego wyrażenia.
- W wyrażeniu `(f (g (h)))` kontynuacją podwyrażenia `(g (h))` jest obliczenie `(f x)`, gdzie `x` jest wartością zwróconą przez `(g (h))`.
- `(call-with-current-continuation procedura)` przekazuje bieżącą kontynuację do jednoargumentowej procedury.
- Będziemy korzystać z następującej definicji skracającej zapis:

```
(define call/cc call-with-current-continuation)
```

Przykłady:

```
(call/cc (lambda (k) (* 5 4))) => 20
```

procedura nie skorzystała z kontynuacji i wyliczyła iloczyn

```
(call/cc (lambda (k) (* 5 (k 4)))) => 4
```

procedura przerwała obliczenie iloczynu i przeniosła obliczenia do kontynuacji oddając wynik 4 (parametr faktyczny kontynuacji)

```
(+ 2 (call/cc (lambda (k) (* 5 (k 4))))) => 6
```

procedura przerwała obliczenie iloczynu i przeniosła obliczenia do kontynuacji oddając wynik 4, który został powiększony o 2

Styl przekazywania kontynuacji (CPS - continuation passing style)

(define (f x…) …) (define (kf x… k) (k (f x…)))

```
(define (return x)
  x)

(define (k+ a b k)
  (k (+ a b)))

(define (k* a b k)
  (k (* a b)))

(k+ 1 2 (lambda (x) (k* x 3 return)))
```

Funkcje rekurencyjne w CPS

```
;;; normal factorial
(define (fact n)
  (if (= n 1)
    1
    (* n (fact (- n 1)))))

;;; CPS factorial
(define (kfact n k)
  (if (= n 1)
    (k 1)
    (kfact (- n 1) (lambda (x) (k (* n x))))))

;;; normal
(+ 3 (fact 4))

;;; CPS
(kfact 4 (lambda (x) (k+ x 3 return)))
```

Pętla do

```
(do ((variable init update) … )
    (test expr)
    body)
```

Połączenie pętli **for** (inicjowanie zmiennych i uaktualnienie ich wartości po każdym przebiegu) z pętlą **while** sprawdzającą warunek `test` i zwracającą wartość wyrażenia `expr` gdy zachodzi. Przy każdym przebiegu pętli wykonywana jest jej treść `body`.

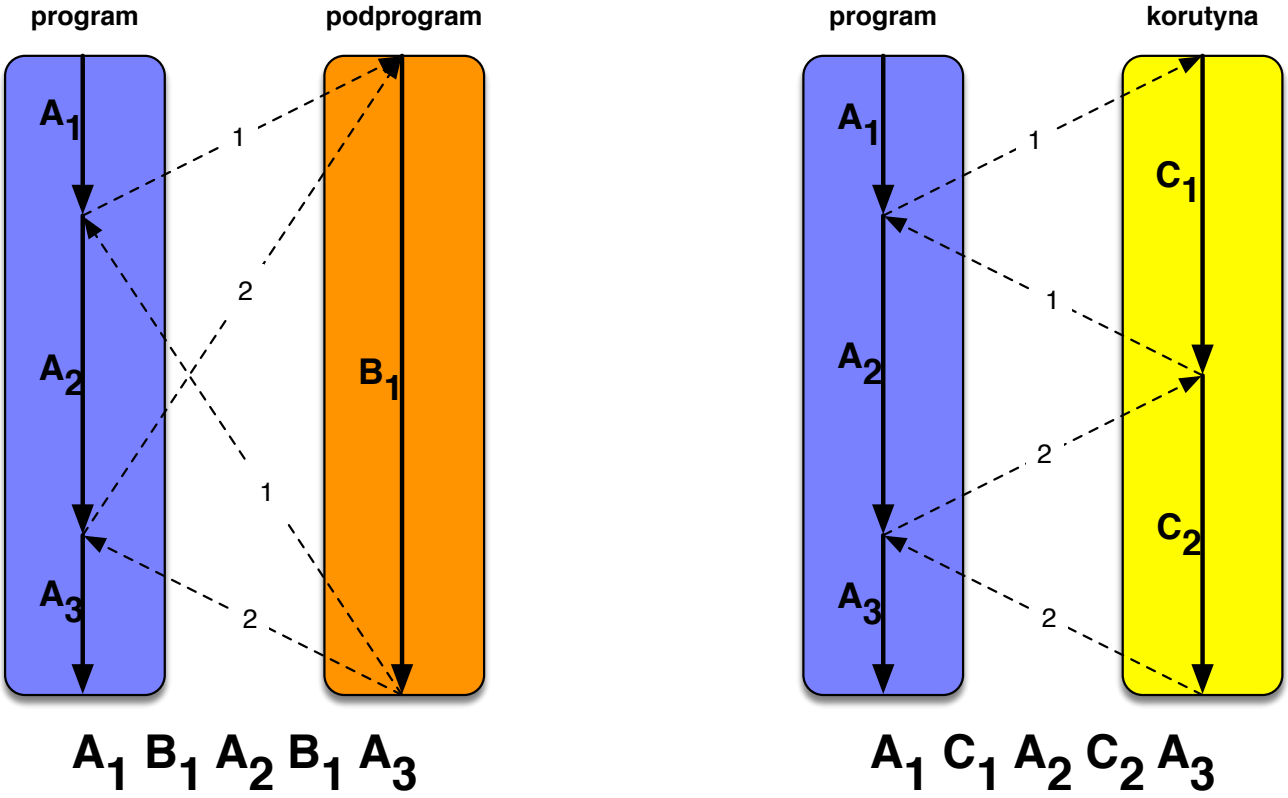
Przykład:

```
(define factorial
  (lambda (n)
    (do ((i n (- i 1)) (a 1 (* a i)))
      ((zero? i) a))))
```

Tworzenie nielokalnego wyjścia z pętli

```
(define member
  (lambda (x ls)
    (call/cc
      (lambda (stop)
        (do ((ls ls (cdr ls)))
          ((null? ls) #f)
          (if (equal? x (car ls))
              (stop ls)))))))
```

Korutyny

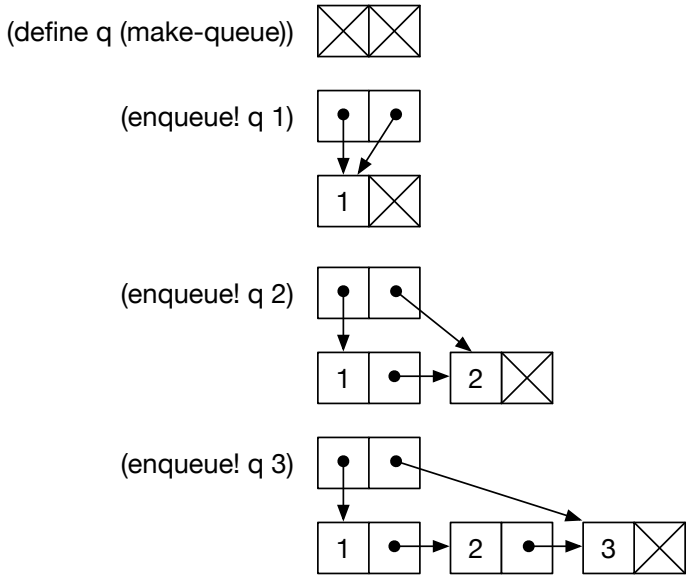


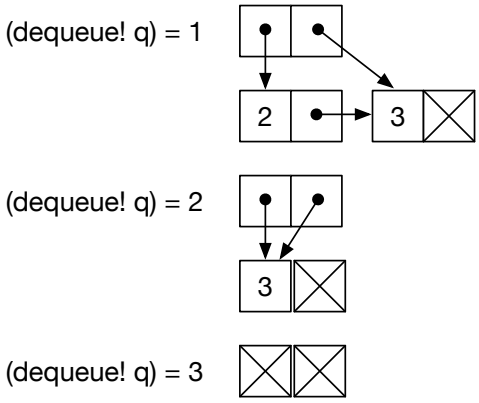
```
;;; coroutines.rkt
;;;
;;; Przykład z "Yet Another Scheme Tutorial"
;;; http://www.shido.info/lisp/idx_scm_e.html

;;; Zmodyfikowano dla języka Racket.

(require scheme/mpair)
```

mutable pair constructors and selectors

(**mcons** A B) (**mpair?** P)
(**mcar** P) (**mcdr** P)
(**set-mcar!** P C) (**set-mcdr!** P C)

```
;;; queue

(define (make-queue)
  (mcons '() '()))

(define (enqueue! queue obj)
  (let ((lobj (mcons  obj '())))
    (if (null? (mcar queue))
        (begin
          (set-mcar! queue lobj)
          (set-mcdr! queue lobj))
        (begin
          (set-mcdr! (mcdr queue) lobj)
          (set-mcdr! queue lobj)))
    (mcar queue)))

(define (dequeue! queue)
  (let ((obj (mcar (mcar queue))))
    (set-mcar! queue (mcdr (mcar queue)))
    obj))
```






```
;;; coroutine

(define process-queue (make-queue))

(define (coroutine thunk)
  (enqueue! process-queue thunk))

(define (start)
    ((dequeue! process-queue)))

(define (pause)
  (call/cc
    (lambda (k)
      (coroutine (lambda () (k #f)))
      (start))))
```

```
;;; example

(coroutine (lambda ()
  (for ([i '(0 1 2 3 4)])
    (display i)
    (pause))))

(coroutine (lambda ()
  (for ([i '(5 6 7 8 9)])
    (display i)
    (pause))))

;;; run example

(start)
```

```
$ racket
Welcome to Racket v6.8.
> (load "coroutines.rkt")
0516273849
> (exit)
$
```

Elementy języka **Haskell**

- Cechy języka
- Historia języka
- Proste przykłady
- Środowisko interakcyjne
- Typy i klasy
- Definiowanie funkcji
- Wyrażenia listowe
- Deklarowanie typów, danych i klas
- Monady

Cechy języka

- zwięzłe programy
- silny system typów
- wyrażenia listowe
- funkcje rekurencyjne
- funkcje wyższego rzędu
- czyste funkcje (bez efektów ubocznych)
- formalny opis efektów ubocznych
- obliczenia leniwe
- wnioskowanie z równań

Historia języka

- lata 30-te Alonzo Church formułuje rachunek lambda
- lata 50-te John McCarthy tworzy język Lisp
- lata 60-te Peter Landin tworzy czysty język funkcyjny ISWIM (bez przypisania zmiennych)
- lata 70-te John Backus tworzy język FP z funkcjami wyższego rzędu
- lata 70-te Robin Milner i inni tworzą pierwszy nowoczesny język funkcyjny ML z polimorfizmem i wnioskowaniem typów
- lata 80-te David Turner tworzy komercyjny język MIRANDA
- 1987 międzynarodowy komitet rozpoczyna prace nad językiem Haskell (logik Haskell Curry)
- lata 90-te Philip Wadler i inni opracowują koncepcję klas dla przeciążenia i monad dla obsługi wyjątków
- 2003 opublikowano Haskell Report
- 2010 poprawiono i zaktualizowano Haskell Report

Proste przykłady

```
sum []     = 0
sum (n:ns) = n + sum ns
```

Funkcja **sum** jest typu Num a => [a] -> a

```
qsort []     = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
  where
    smaller = [a | a <- xs, a <= x]
    larger  = [b | b <- xs, b > x]
```

Funkcja **qsort** jest typu Ord a => [a] -> [a]

Środowisko interakcyjne

- kompilator GHC (Glasgow Haskell Compiler)
- program GHCi
- uruchamianie:

```
$ ghci
> 2+3*4
14
> (2+3)*4
20
> sqrt (3^2+4^2)
5.0
> :quit
```

- wybrane funkcje standardowe (biblioteka *Standard Prelude*):

```
> head [1,2,3,4,5]
1
> tail [1,2,3,4,5]
[2,3,4,5]
> [1,2,3,4,5] !! 2
3
> take 3 [1,2,3,4,5]
[1,2,3]
> drop 3 [1,2,3,4,5]
[4,5]
```

```
> length [1,2,3,4,5]
5
> sum [1,2,3,4,5]
15
> product [1,2,3,4,5]
120
> [1,2,3] ++ [4,5]
[1,2,3,4,5]
> reverse [1,2,3,4,5]
[5,4,3,2,1]
```

- zastosowanie funkcji ma wyższy priorytet niż inne operatory:

matematyka	Haskell
f(a, b) + cd	f a b + c*d
f(x)	f x
f(x, y)	f x y
f(g(x))	f (g x)
f(x, g(y))	f x (g y)
f(x)g(y)	f x * g y

- skrypty w Haskellu

```
-- test.hs
double x = x + x
quadruple x = double (double x)
```

```
$ ghci test.hs
> quadruple 10
40
> take (double 2) [1,2,3,4,5]
[1,2,3,4]
> :quit
```

- w osobnym terminalu można dopisać do pliku **test.hs** następujące definicje:

```
factorial n = product [1..n]
average ns  = sum ns `div` length ns
```

```
> :reload
> factorial 10
3628800
> average [1,2,3,4,5]
3
> :quit
```

Komenda	Działanie
:load name	załadowanie skryptu
:reload	przeładowanie bieżącego skryptu
:set editor name	zdefiniowanie polecenia edytora
:edit name	edycja skryptu
:edit	edycja bieżącego skryptu
:type expr	pokaż typ wyrażenia
:?	pokaż wszystkie komendy
:quit	wyjdź z GHCi

- grupowanie definicji na podstawię wcięć:

```
a = b + c
  where
    b = 1
    c = 3
d = a * 2
```

```
a = b + c
  where
    { b = 1;
      c = 2};
d = a * 2
```

```
a = b + c where {b = 1; c = 2};
d = a * 2
```

Unikaj tabulacji (stosuj spacje).

- komentarze

```
-- ble ble ble

{- ble ble ble
   ble ble ble -}
```

Typy i klasy

```
False :: Bool
True :: Bool
not :: Bool -> Bool

not False :: Bool
not True :: Bool
not (not False) :: Bool
```

$$\frac{f :: A \mapsto B \quad e :: A}{f\ e :: B}$$

Wyrażenie **not 3** nie ma sensu. Podczas kontroli typu występuje *błąd typu* (argument funkcji **not** powinien być typu **Bool**). Kontrola typu odbywa się przed rozpoczęciem obliczeń i w ich trakcji błąd ten na pewno nie wystąpi.

Warunkowe wyrażenie:

```
if True then 1 else False
```

zawiera *type error* gdyż **1** i **False** powinny być tego samego typu.

```
> :type not
not :: Bool -> Bool
> :type False
False :: Bool
> :type not False
not False :: Bool
```

Podstawowe typy

- Bool — wartości logiczne
- Char — znaki
- String — łańcuchy znaków
- Int — liczby całkowite ustalonej precyzji (GHCi $-2^{63}..2^{63}-1$)
- Integer — liczby całkowite dowolnej precyzji
- Float — liczby zmiennopozycyjne pojedynczej precyzji
- Double — liczby zmiennopozycyjne podwójnej precyzji

```
> 2^63::Int
-9223372036854775808
> 2^63
9223372036854775808
> sqrt 2::Float
1.4142135
> sqrt 2::Double
1.4142135623730951
```

Listy

Typ **[T]** oznacza listę obiektów typu **T**.

```
[False,True,False] :: [Bool]
['a','b','c','d'] :: [Char]
["One","Two","Three"] :: [String]

> :type []
[] :: [t]
> :type [[]]
[[]] :: [[t]]
```

Krotki

Typ **(T1, T2, …, Tn)** oznacza krotkę obiektów typów **T1**, **T2**, …, **Tn**.

```
(False,True) :: (Bool,Bool)
(False,'a',True) :: (Bool,Char,Bool)
("Yes",True,'a') :: (String,Bool,Char)

> :type ()
() :: ()
```

specjalny typ oznaczający pustą krotkę

Funkcje

Typ **T1 -> T2** oznacza typ wszystkich funkcji odwzorowujących argument typu **T1** w wynik typu **T2**.

```
not :: Bool -> Bool

> :type even
even :: Integral a => a -> Bool

add :: (Int, Int) -> Int
add (x,y) = x + y

zeroto :: Int -> [Int]
zeroto n = [0..n]
```

Curried functions

```
add' :: Int -> (Int -> Int)
add' x y = x + y

> :type add'
add' :: Int -> Int -> Int
> :type add' 1
add' 1 :: Int -> Int
> :type add' 1 2
add' 1 2 :: Int

mult :: Int -> (Int -> (Int -> Int))
mult x y z = x * y * z
```

Wyrażenie **mult x y z** oznacza to samo co **((mult x) y) z**.

Typy polimorficzne

```
> length [1,2,3,4,5]
5
> length [”yes”, ”no”]
2

length :: [a] -> Int
```

Zmienna typu a oznacza listę dowolnego typu.
Typ polimorficzny zawiera co najmniej jedną zmienną typu:

```
fst :: (a,b) -> a
head :: [a] -> a
tail :: [a] -> [a]
take :: Int -> [a] -> [a]
zip :: [a] -> [b] -> [(a,b)]
id :: a -> a
```

Przeciążanie

```
> 1 + 2
3
> 1.0 + 2.0
3.0

(+) :: Num a => a -> a -> a
```

Num a wskazuje, że typ **a** powinien być typem numerycznym.

```
(*) :: Num a => a -> a -> a
negate :: Num a => a -> a
abs :: Num a => a -> a
```

Podstawowe klasy typów

- Eq — typy z równością
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  Bool, Char, String, Int, Integer, Float, Double, listy i krotki są klasy Eq.
- Ord — typy porządkowe
  (<), (<=), (>), (>=), min, max
- Show — typy prezentowalne
- Read — typy odczytywalne
- Num — typy numeryczne
  (+), (*), negate, abs
- Integral — typy całkowite
  div, mod
  Int i Integer są klasy Integral.
- Fractional — typy ułamkowe
  (/), recip

```
show :: a -> String

> show False
"False"
> show 'a'
"'a'"
> show 123
"123"
> show [1,2,3]
"[1,2,3]"
> show ('a',False)
"('a',False)"
```

```
read :: String -> a

> read "False" :: Bool
False
> read "'a'" :: Char
'a'
> read "123" :: Int
123
> read "[1,2,3]" :: [Int]
[1,2,3]
```

Definiowanie funkcji

```
even :: Integral a => a -> Bool
even n = n `mod` 2 == 0

splitAt :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs, drop n xs)

recip :: Fractional a => a -> a
recip n = 1 / n
```

Wyrażenie warunkowe

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n

signum :: Int -> Int
signum n = if n < 0 then -1
  else
    if n == 0 then 0
      else 1
```

Równania z wartownikiem

```
abs n | n >= 0    = n
      | otherwise = -n

signum n | n < 0     = -1
         | n == 0    = 0
         | otherwise = 1
```

Dopasowanie wzorca

```
not :: Bool -> Bool
not True = False
not False = True

(&&) :: Bool -> Bool - Bool
True && True   = True
True && False  = False
False && True  = False
False && False = False

True && True = True
_ && _ = False

True && b = b
False && _ = False
```

```
b && b = b
_ && _ = False
```

to nie jest poprawne
(zmienna **b** dwa razy
po lewej stronie równania)

```
b && c | b == c = b
       | otherwise = False
```

to jest poprawne

Wzorce na krotkach

```
fst :: (a,b) -> a
fst (x,_) = x
snd :: (a,b) -> b
snd (_,y) = y
```

Wzorce na listach

```
head :: [a] -> a
head (x:_) = x
tail :: [a] -> [a]
tail (_:xs) = xs
```

Lambda wyrażenia

```
\x -> x + x

> (\x -> x + x) 2
4

add :: Int -> Int -> Int
add x y = x + y

add :: Int -> (Int -> Int)
add = \x -> (\y -> x + y)

const :: a -> b -> a
const x _ = x

const :: a -> (b -> a)
const x = \_ -> x
```

```
odds :: Int -> [Int]
odds n = map f [0..n-1]
    where f x = x*2+1

odds :: Int -> [Int]
odds n = map (\x -> x*2+1) [0..n-1]
```

Wyrażenia listowe

Podstawy

```
> [x^2 | x <- [1..5]]
[1,4,9,16,25]

> [(x,y) | x <- [1,2,3], y <- [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]

> [(x,y) | y <- [4,5], x <- [1,2,3]]
[(1,4),(2,4),(3,4),(1,5),(2,4),(3,5)]

> [(x,y) | x <- [1..3], y <- [x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

```
concat :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]

firsts :: [(a,b)] -> [a]
firsts ps = [x | (x,_) <- ps]

length :: [a] -> Int
length xs = sum [1 | _ <- xs]
```

Wartownicy

```
factors :: Int -> [Int]
factors n = [x | x <- [1..n], n `mod` x == 0]

> factors 15
[1,3,5,15]

> factors 7
[1,7]

prime :: Int -> Bool
prime n = factors n == [1,n]

> prime 15
False

>prime 7
True
```

Funkcja zip

```
> zip ['a','b','c'] [1,2,3,4]
[('a',1),('b',2),('c',3)]

pairs :: [a] -> [(a,a)]
pairs xs = zip xs (tail xs)

> pairs [1,2,3,4]
[(1,2),(2,3),(3,4)]

sorted :: Ord a => [a] -> Bool
sorted xs = and [x <= y | (x,y) <- pairs xs]

> sorted [1,2,3,4]
True

> sorted [1,3,2,4]
False
```

```
positions :: Eq a => a -> [a] -> [Int]
positions x xs = [i | (x',i) <- zip xs [0..], x == x']

> positions False [True,False,True,False]
[1,3]
```

Deklarowanie typów

Wprowadzenie nowej nazwy na istniejący typ

```
type String = [Char]

type Pos = (Int, Int)
type Trans = Pos -> Pos
```

`type Tree = (Int, [Tree])`

to nie jest poprawne
(deklaracja typu nie może być rekurencyjna)

Deklaracja parametryzowana

```
type Pair a = (a, a)

type Assoc k v = [(k, v)]

find :: Eq k => k -> Assoc k v -> v
find k t = head [v | (k',v) <- t, k == k']
```

Deklarowanie danych

```
data Move = North | South | East | West

move :: Move -> Pos -> Pos
move North (x,y) = (x,y+1)
move South (x,y) = (x,y-1)
move East  (x,y) = (x+1,y)
move West  (x,y) = (x-1,y)

moves :: [Move] -> Pos -> Pos
moves [] p = p
moves (m:ms) p = moves ms (move m p)
```

```
data Shape = Circle Float | Rect Float Float
```

Circle i **Rect** są konstruktorami

```
square :: Float -> Shape
square n = Rect n n

area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y

> :type Circle
Circle :: Float -> Shape
> :type Rect
Rect :: Float -> Float -> Shape
```

Deklaracja parametryzowana

```
data Maybe a = Nothing | Just a

savediv :: Int -> Int -> Maybe Int
savediv _ 0 = Nothing
savediv m n = Just (m `div` n)

savehead :: [a] -> Maybe a
savehead [] = Nothing
safehead xs = Just (head xs)
```

Typy rekursywne

```
data Nat = Zero | Succ Nat

nat2int :: Nat -> Int
nat2int Zero = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n - 1))

add :: Nat -> Nat -> Nat
add m n = int2nat (nat2int m + nat2int n)

add :: Nat -> Nat -> Nat
add Zero n = n
add (Succ m) n = Such (add m n)
```

```
data List a = Nil | Cons a (List a)

len :: List a -> Int
len Nil = 0
len (Cons _ xs) = 1 + len xs

data Tree a = Leaf a | Node (Tree a) a (Tree a)

occurs :: Eq a => a -> Tree a -> Bool
occurs x (Leaf y) = x == y
occurs x (Node l y r) = x == y ||
    occurs x l ||
    occurs x r

flatten :: Tree a -> [a]
flatten (Leaf x) = [x]
flatten (Node l x r) = flatten l ++ [x] ++ flatten [r]

occurs :: Ord a => a -> Tree a -> Bool
occurs x (Leaf y) = x == y
occurs x (Node l y r) | x == y    = True
                      | x < y     = occurs x l
                      | otherwise = occurs x r
```

Deklarowanie klas

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y = not (x == y)
```

- Aby typ **a** był przykładem klasy **Eq** musi obsługiwać równość i nierówność.
- Domyślna definicja dla **/=** została zawarta, zatem deklaracja przykładu wymaga tylko zdefiniowania **==**.

```
instance Eq Bool where
    False == False = True
    True  == True  = True
    _     == _     = False
```

Klasy można rozszerzać do nowych klas:

```
class Eq a => Ord a where
    (<), (<=), (>), (>=) :: a -> a -> Bool
    min, max             :: a -> a -> a

    min x y | x <= y = x
            | otherwise = y

    max x y | x <= y = y
            | otherwise = x

instance Ord Bool where
    False < True = True
    _     < _    = False
    b <= c = (b < c) || (b == c)
    b > c  = c < b
    b >= c = c <= b
```

Monady

Funktory

```
inc :: [Int] -> [Int]
inc [] = []
inc (n:ns) = n+1 : inc ns

sqr :: [Int] -> [Int]
sqr [] = []
sqr (n:ns) = n^2 : sqr ns

map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs

inc = map (+1)
sqr = map (^2)
```

Ogólnie: mapować funkcję po każdym elemencie struktury danych (nie tylko po listach).

Klasa typów, które umożliwiają mapowanie po strukturze nazywa się **funktorami**.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

typ parametryczny

Przykład:

```
instance Functor [] where
    -- fmap :: (a -> b) -> [a] -> [b]
    fmap = map
```

```
data Maybe a = Nothing | Just a

instance Functor Maybe where
    -- fmap :: (a -> b) -> Maybe a -> Maybe b
    fmap _ Nothing = Nothing
    fmap g (Just x) = Just (g x)

> fmap (+1) Nothing
Nothing
> fmap (*2) (Just 3)
Just 6
> fmap not (Just False)
Just True
```

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
    deriving Show

instance Functor Tree where
  -- fmap :: (a -> b) -> Tree a -> Tree b
  fmap g (Leaf x) = Leaf (g x)
  fmap g (Node l r) = Node (fmap g l) (fmap g r)

> fmap length (Leaf ”abc”)
Leaf 3
> fmap even (Node (Leaf 1) (Leaf 2))
Node (Leaf False) (Leaf True)
```

```
inc :: Functor f => f Int -> f Int
inc = fmap (+1)

> inc (Just 1)
Just 2
> inc [1,2,3,4,5]
[2,3,4,5,6]
> inc (Node (Leaf 1) (Leaf 2))
Node (Leaf 2) (Leaf 3)
```

Aplikacje

```
fmap0 :: a -> f a
fmap1 :: (a -> b) -> fa -> f b
fmap2 :: (a -> b -> c) -> f a -> f b -> fc
fmap3 :: (a -> b -> c -> d) -> f a -> f b -> f c -> f d
…
```

nieskończenie wiele przypadków!!!

```
> fmap2 (+) (Just 1) (Just 2)
Just 3
```

Skorzystamy z **curring**:

```
pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b

g <*> x <*> y <*> z ≡ ((g <*> x) <*> y) <*> z
```

Typowe użycie **pure** i **<*>** (*styl aplikacyjny*):

```
pure g <*> x1 <*> x2 … <*> xn
```

```
fmap0 = pure
fmap1 g x = pure g <*> x
fmap2 g x y = pure g <*> x <*> y
fmap3 g x y z = pure g <*> x <*> y <*> z
…
```

Klasa aplikacji:

```
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just g) <*> mx = fmap g mx

> pure (+1) <*> Just 1
Just 2
> pure (+) <*> Just 1 <*> Just 2
Just 3
> pure (+) Nothing <*> Just 2
Nothing
```

```
instance Applicative [] where
    -- pure :: a -> [a]
    pure x = [x]
    -- (<*>) :: [a -> b] -> [a] -> [b]
    gs <*> xs = [g x | g <- gs, x <- xs]

> pure (+1) <*> [1,2,3]
[2,3,4]
> pure (+) <*> [1] <*> [2]
[3]
> pure (*) <*> [1,2] <*> [3,4]
[3,4,6,8]
```

W powyższych przykładach, typ **[a]** rozumiemy jako uogólnienie typu **Maybe a** umożliwiające wielokrotne wyniki w przypadku powodzenia.

Lista pusta może wówczas oznaczać niepowodzenie.

```
prods :: [Int] -> [Int] -> [Int]
prods xs ys = [x*y | x <- xs, y <- ys]

prods :: [Int] -> [Int] -> [Int]
prods xs ys = pure (*) <*> xs <*> ys
```

Dzięki stylowi aplikacyjnemu dla list możliwe jest pisanie programów **niedeterministycznych**, w których możemy stosować czyste funkcje do wielowartościowych argumentów bez potrzeby obsługi wyboru wartości albo propagowania niepowodzenia.

```
instance Applicative IO where
    -- pure :: a -> IO a
    pure = return
    -- (<*>) :: IO (a -> b) -> IO a -> IO b
    mg <*> mx = do { g <- mg; x <- mx;
                     return (g x)}
```

Przykład: czytanie n znaków

```
getChars :: Int -> IO String
getChars 0 = return []
getChars n = pure (:) <*> getChar <*> getChars (n-1)
```

Monady

```
data Expr = Val Int | Div Expr Expr

eval :: Expr -> Int
eval (Val n) = n
eval (Div x y) eval x `div` eval y

> eval (Div (Val 1) (Val 0))
*** Exception : divide by zero

safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv n m = Just (n `div` m)
```

```
eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) = case eval x of
    Nothing -> Nothing
    Just n -> case eval y of
        Nothing -> Nothing
        Just m -> safediv n m
```

wiele przypadków aby propagować niepowodzenie

to nie jest poprawne!!!
safediv jest typu Int -> Int -> Maybe Int
potrzebny jest typ Int -> Int -> Int

```
eval :: Expr -> Maybe Int
eval (Val n) = pure n
eval (Div x y) = pure safediv <*> eval x <*> eval y
```

Jak zapisać **eval** by było poprawne?

Użyjemy operatora >>= (*bind*):

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
mx >>= f = case mx of
    Nothing -> Nothing
    Just x -> f x

eval :: Expr -> Maybe Int
eval (Val n) = Just n
eval (Div x y) =
    eval x >>= \n ->
        eval y >>= \m ->
            safediv n m
```

Wyrażenie:

```
m1 >>= \x1 ->
m2 >>= \x2 ->
…
mn >>= \mn ->
f x1 x2 … xn
```

można w Haskellu zapisać prościej:

```
do x1 <- m1
   x2 <- m2
   …
   xn <- mn
   f x1 x2 … xn

eval :: Expr -> Mayby Int
eval (Val n) = Just n
eval (Div x y) do n <- eval x
                  m <- eval y
                  safediv n m
```

Klasa monad:

```
class Applicative m => Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    return = pure
```

Przykłady:

```
instance Monad Maybe where
    -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
    Nothing >>= _ = Nothing
    (Just x) >>= f = f x

instance Monad [] where
    -- (>>=) :: [a] -> (a -> [b]) -> [b]
    xs >>= f = [y | x <- xs, y <- f x]

pairs :: [a] -> [b] -> [(a,b)]
pairs xs ys = do x <- xs
                 y <- ys
                 return (x,y)
> pairs [1,2] [3,4]
[(1,3),(1,4),(2,3),(2,4)]
```

Elementy języka **Erlang**

- Cechy języka
- Podstawy
- Programowanie sekwencyjne
- Programowanie współbieżne

Cechy języka

- równoległość na wielu rdzeniach
- odporność na awarie
- programowanie funkcyjne
- obliczenia czasu rzeczywistego
- obliczenia rozproszone
- hot swap

Podstawy

```
$ erl
Erlang/OTP…

Eshell… (abort with ^G)
1> 20+30.
50
2> 2+3*4.
14
3> (2+3)*4.
20
4> X = 123456789.
123456789
5> X.
123456789
6> X = 1234.
** exception error: no match of right side value 1234
```

zmienne nie są zmienne a = nie jest przypisaniem

Atomy

Ciąg znaków alfanumerycznych oraz _ i @ zaczynający się od małej litery.

```
> hello.
hello
```

Liczby całkowite

Dowolna precyzja.

```
> 16#cafe + 32#sugar.
30411865
```

Zmienne

Ciąg znaków alfanumerycznych zaczynający się od wielkiej litery.

Liczby zmiennopozycyjne

```
> 4/2.
2.0
> 5 div 3.
1
> Pi = 3.14159.
3.14159
> R = 5.
5
> Pi * R * R.
78.53975
```

Krotki

```
> Person = {person,
              {name, joe},
              {height, 1.82},
              {footsize, 42},
              {eyecolour, brown}}
{person, {name, joe}, {height, 1.82}, {footsize, 42},
 {eyecolour, brown}}
> F = {firstName, joe}
{firstName, joe}
> L = {lastName, armstrong}
{lastName, armstrong}
> P = {person, F, L}
{person, {firstName, joe}, {lastName, armstrong}}
> {true, Q, 23}.
* 1: variable 'Q' is unbound
```

Listy

```
> [1+7, hello, 2-2, {cost, apple, 30-20}, 3].
[8, hello, 0, {cost, apple, 10}, 3]

[]
[H | T]
```

Łańcuchy znaków

```
> Name = ”Hello”.
”Hello”
> [83, 117, 114, 112, 114, 105, 115, 101].
"Surprise"
```

Programowanie sekwencyjne

Moduły

```
%% geometry.erl
-module(geometry).
-export([area/1]).
area({rectangle, Width, Ht}) -> Width*Ht;
area({circle, R}) -> 3.14159*R*R.

> c(geometry)
{ok, geometry}
> geometry:area({rectangle, 10, 5}).
50
> geometry:area({circle, 1.4}).
6.15752
```

```
Pattern1 ->
    Expressions1;
Pattern2 ->
    Expressions2;
…
PatternN ->
    ExpressionsN.


%% lib_misc.erl
sum(L) -> sum(L, 0).
sum([], N) -> N;
sum([H | T], N) -> sum(T, H + N).
```

Lambda wyrażenia

```
> Z = fun(X) -> 2*X end.
#Fun<…>
> Z(2).
4
> Hypot = fun(X, Y) -> math:sqrt(X*X+Y*Y) end.
#Fun<…>
> Hypot(3, 4).
5.0
> Hypot(3).
** exception error: interpreted function with arity 2
called with one argument
> TempConvert = fun({c, C}) -> {f, 32+C*9/5};
                   ({f,F}) -> {c, (F-32)*5/9}
                end.
#Fun<…>
> L = [1,2,3,4].
[1,2,3,4]
> lists:map(Z, [1,2,3,4]).
[2,4,6,8]
```

```
> Even = fun(X) -> (X rem 2) =:= 0 end.
#Fun<…>
> lists:map(Even, [1,2,3,4,5,6,7,8]).
[false, true, false, true, false, true, false, true]
> lists:filter(Even, [1,2,3,4,5,6,7,8]).
[2,4,6,8]

> Mult = fun(Times) -> (fun(X) -> X*Times end) end.
#Fun<…>
> Triple = Mult(3).
#Fun<…>
> Triple(5).
15

%% lib_misc.erl
for(Max, Max, F) -> [F(Max)];
for(I, Max, F) -> [F(I) | for(I+1, Max, F)].

> lib_misc:for(1, 10, fun(I) -> I end).
[1,2,3,4,5,6,7,8,9,10]
> lib_misc:for(1, 10, fun(I) -> I*I end).
[1,4,9,16,25,36,49,64,81,100]


%% mylists.erl
sum([H | T]) -> H + sum(T);
sum([]) -> 0.

map(_, []) -> [];
map(F, [H | T]) -> [F(H) | map(F, T)].
```

Wyrażenia listowe

```
> [I*I || I <- [1,2,3]].
[1,4,9]    \__ generator __/

map(F, L) -> [F(X) || X <- L].

> [I || I <- [1,2,3,4,5], I rem 2 =:= 0].
[2,4]     \___ generator ___/      \_____ filtr _____/
```

Quicksort

```
qsort([]) -> [];
qsort([Pivot | T]) ->
    qsort([X || X <- T, X < Pivot])
    ++ [Pivot] ++
    qsort([X || X <- T, X >= Pivot]).
```

Trójki pitagorejskie

```
pythag(N) ->
    [{A, B, C} ||
        A <- lists:seq(1, N),
        B <- lists:seq(1, N),
        C <- lists:seq(1, N),
        A+B+C =< N,
        A*A + B*B =:= C*C
    ].
```

wyrażenie arytmetyczne	priorytet
+X	1
-X	1
X * Y	2
X / Y	2
bnot X	2
X div Y	2
X rem Y	2
X band Y	2
X + Y	3
X - Y	3
X bor Y	3
X bxor Y	3
X bsl N	3
X bsr N	3

Wartownicy

```
max(X, Y) when X > Y -> X;
max(X, Y) - Y.
```

`G1; G2; …; Gn` przynajmniej jeden spełniony
`G1, G2, …, Gn` wszystkie spełnione

```
f(X, Y) when is_integer(X), X > Y, Y < 6 -> …
```

```
f(X) when (X == 0) or (1/X > 2) ->
    …
```
wartownik zawodzi gdy X jest 0

```
g(X) when (X == 0) orelse (1/X > 2) ->
    …
```
wartownik spełniony gdy X jest 0

predykaty	funkcje
`is_atom(X)`	`abs(X)`
`is_binary(X)`	`element(N, X)`
`is_constant(X)`	`float(X)`
`is_float(X)`	`hd(X)`
`is_function(X)`	`length(X)`
`is_function(X, N)`	`node()`
`is_integer(X)`	`node(X)`
`is_list(X)`	`round(X)`
`is_number(X)`	`self()`
`is_pid(X)`	`size(X)`
`is_port(X)`	`trunc(X)`
`is_reference(X)`	`tl(X)`
`is_tuple(X)`	
`is_record(X, Tag)`	
`is_record(X, Tag, N)`	

Wyrażenia case i if

```
case Expression of
    Pattern1 [when Guard1] -> Expr_seq1;
    Pattern2 [when Guard2] -> Expr_seq2;
    …
end

filter(P, [H | T]) -> filter1(P(H), H, P, T);
filter(P, []) -> [].
filter1(true, H, P, T) -> [H | filter(P, T)];
filter1(false, H, P, T) -> filter(P, T).

filter(P, [H | T]) ->
    case P(H) of
        true -> [H | filter(P, T)];
        false -> filter(P, T)
    end;
filter(P, []) ->
    [].



if
    Guard1 -> Expr_seq1;
    Guard2 -> Expr_seq2;
    …
    true -> Expr_seqN
end
```

Aplikacja

```
apply(Mod, Func, [Arg1, Arg2, …, ArgN])
≡
Mod:Func(Arg1, Arg2, …, ArgN)
```

Operatory ++ i --

```
> [1,2,3] ++ [4,5,6].
[1,2,3,4,5,6]
> [1,2,3,1,2,3] -- [1].
[2,3,1,2,3]
> [1,2,3,1,2,3] -- [1,1].
[2,3,2,3]
```

Programowanie współbieżne

- **Pid = spawn(Func)**
  tworzy nowy proces obliczający funkcję **Func**
- **Pid ! Message**
  wysyła komunikat do procesu o **Pid**
- **receive … end**
  odbiera i przetwarza komunikaty

```
receive
    Pattern1 [when Guard1] ->
        Expressions1;
    Pattern2 [when Guard2] ->
        Expressions2;
    …
end
```

```
%% area_server0.erl
-module(area_server0).
-export([loop/0]).

loop() ->
    receive
        {rectangle, Width, Ht} ->
          io:format("Area of rectangle is ~p~n",
                    [Width*Ht]),
          loop();
        {circle, R} ->
          io:format("Area of circle is ~p~n",
                    [3.14159*R*R]),
          loop();
        Other ->
          io:format("I dont know area of ~p~n",
                    [Other]),
          loop()
    end.
```

```
> c(area_server0).
> Pid = spawn(fun area_server0:loop/0).
> Pid ! {rectangle, 6, 10}.
Area of rectangle is 60
> Pid ! {circle, 1.2}.
Area of circle is 6.15752
> Pid ! {triangle, 4, 5, 6}.
I dont know area of {triangle, 4, 5, 6}
```

```
%% area_server1.erl
-module(area_server1).
-export([loop/0, rpc/2]).

loop() ->
    receive
        {From, {rectangle, Width, Ht}} ->
            From ! Wdith*Ht,
            loop();
        {From, {circle, R}} ->
            From ! 3.14159*R*R,
            loop();
        {From, Other} ->
            From ! {error, Other},
            loop()
    end.

rpc(Pid, Request) ->
    Pid ! {self(), Request},
    receive
        Response -> Response
    end.
```

Remote **P**rocedure **C**all

```
> c(area_server1).
> Pid = spawn(fun area_server1:loop/0).
> area_server1:rpc(Pid, {rectangle, 6, 10}).
60
> area_server1:rpc(Pid, {circle, 1.2}).
6.15752
> area_server1:rpc(Pid, {triangle, 4, 5, 6}).
{error, {triangle, 4, 5, 6}}
```

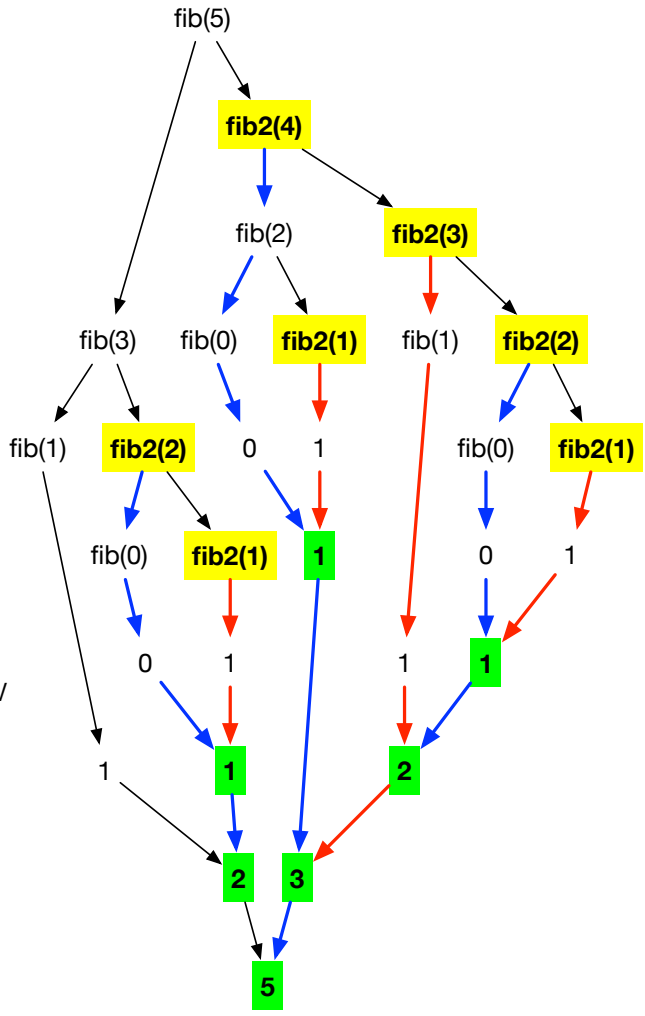
```
%% fib2.erl
-module(fib2).
-export([fib/1]).

fib(0) -> 0;
fib(1) -> 1;
fib(N) ->
    fib2(N - 1) + fib(N - 2).

fib2(N) ->
    Pid = spawn(fun worker/0),
    Pid ! {self(), N},
    receive
        F -> F
    end.

worker() ->
    receive
        {From, N} -> From ! fib(N)
    end.
```

```
> c(fib2).
{ok, fib}
> fib2:fib(5).
5
> fib2:fib(30).
832040
> fib2:fib(40).
102334155
```

7 procesów

1,346,268 procesów

165,580,140 procesów



Serwer par klucz-wartość

put(Key, Value) -> OldValue | undefined

zapisuje w słowniku parę klucz-wartość

get(Key) -> Value | undefined

pobiera ze słownika wartość związaną z kluczem

register(AnAtom, Pid)

rejestruje Pid procesu pod nazwą AnAtom

unregister(AnAtom)

usuwa rejestrację związaną z AnAtom

whereis(AnAtom) -> Pid | undefined

znajduje gdzie jest zarejestrowany AnAtom

rpc:call(Node, Mod, Fun, [Arg1, Arg2, …, ArgN])

```
%% kvs.erl
-module(kvs).
-export([start/0, store/2, lookup/1]).

start() -> register(kvs, spawn(fun() -> loop() end)).

store(Key, Value) -> rpc({store, Key, Value}).

lookup(Key) -> rpc({lookup, Key}).

rpc(Q) ->
   kvs ! {self(), Q},
   receive
      {kvs, Reply} -> Reply
   end.

loop() ->
   receive
      {From, {store, Key, Value}} ->
         put(Key, {ok, Value}),
         From ! {kvs, true},
         loop();
      {From, {lookup, Key}} ->
         From ! {kvs, get(Key)},
         loop()
   end.
```

co gdy nie można doczekać się na odpowiedź?

Przykład 1: prosty serwer nazw

```
$ erl
> kvs:start().
true
> kvs:store({location, joe}, ”Stockholm”).
true
> kvs:store(weather, raining).
true
> kvs:lookup(weather).
{ok, raining}
> kvs:lookup({location, joe}).
{ok, ”Stockholm”}
> kvs:lookup({location, jane}).
undefined
```

Przykład 2: klient i serwer na tym samym serwerze ale różnych węzłach

```
$ erl -sname gandalf
(gandalf@localhost)> kvs:start().
true
```

```
$ erl -sname bilbo
(bilbo@localhost)> rpc:call(gandalf@localhost, kvs,
                            store, [weather, fine]).
true
(bilbo@localhost)> rpc:call(gandalf@localhost, kvs,
                            lookup, [weather]).
{ok,fine}
```

Przykład 3: klient i serwer na różnych serwerach ale w tej samej podsieci

```
doris $ erl -name gandalf -setcookie abc
(gandalf@doris.domain)> kvs:start().
true
```

```
george $ erl -name bilbo -setcookie abc
(bilbo@george.domain)> rpc:call(gandalf@doris.domain,
                                kvs,
                                store, [weather, fine]).
true
(bilbo@george.domain)> rpc:call(gandalf@doris.domain,
                                kvs,
                                lookup, [weather]).
{ok,fine}
```

Timeout

```
receive
  …
after Time ->
  …
end
```

czas w milisekundach

```
sleep(T) ->
    receive
    after T -> ok
    end.

flush() ->
    receive
        _ -> flush()
    after 0 -> ok
    end.
```

Obliczenia na wielu rdzeniach

```
przemko@otryt:~$ erl
Erlang/OTP 18 [erts-7.3] [source] [64-bit] [smp:80:80]
[async-threads:10] [kernel-poll:false]

Eshell V7.3  (abort with ^G)
1>
```

timer:tc(Mod, Func, Args) -> {Time, Value}

czas w mikrosekundach

catch(Expr)

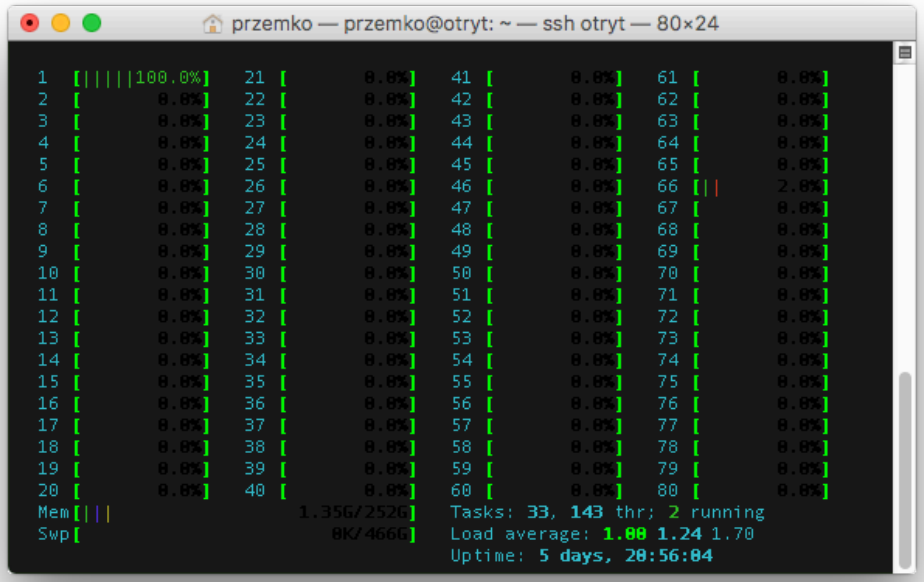
w przypadku błędu ma wartość **{'EXIT', opis_wyjątku}**

```
> A = 1/2.
0.5
5> B = 1/0.
** exception error: an error occurred when evaluating an arithmetic
expression
     in operator  '/'/2
        called as 1 / 0
6> catch(C = 1/2).
0.5
7> catch(D = 1/0).
{'EXIT',{badarith,[{erlang,'/',[1,0],[]},
                   {erl_eval,do_apply,6,[{file,"erl_eval.erl"},{line,674}]},
                   {erl_eval,expr,5,[{file,"erl_eval.erl"},{line,438}]},
                   {erl_eval,expr,5,[{file,"erl_eval.erl"},{line,431}]},
                   {shell,exprs,7,[{file,"shell.erl"},{line,686}]},
                   {shell,eval_exprs,7,[{file,"shell.erl"},{line,641}]},
                   {shell,eval_loop,3,[{file,"shell.erl"},{line,626}]}]}}
```

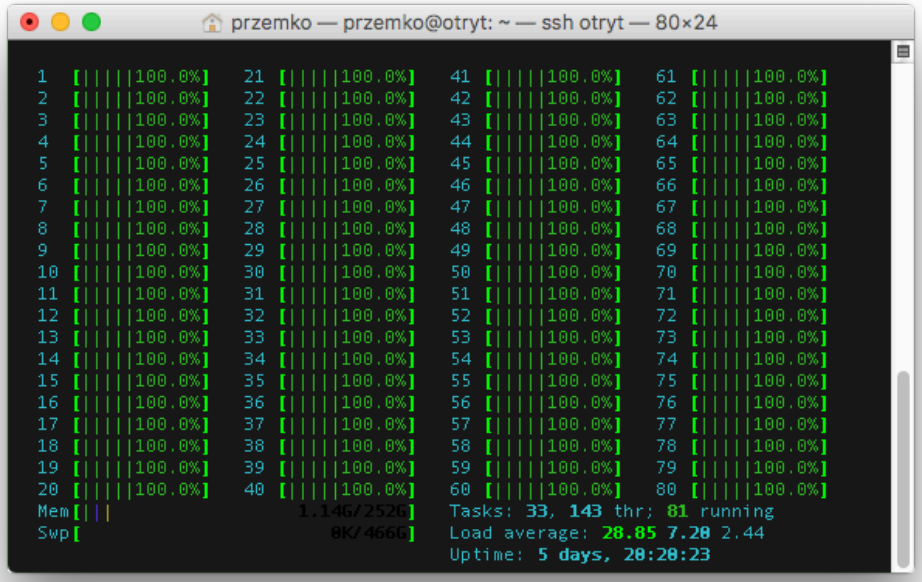
Przykład: równoległy map

```
-module(lib_misc).
-export([map/2, pmap/2]).

map(_, []) -> [];
map(F, [H|T]) -> [F(H)|map(F, T)].

pmap(F, L) ->
   S = self(),
   Ref = erlang:make_ref(), % świeża wartość
   Pids = map( fun(I) ->
            spawn(fun() -> do_f(S, Ref, F, I) end)
         end, L),
   gather(Pids, Ref).

do_f(Parent, Ref, F, I) ->
   Parent ! {self(), Ref, (catch F(I))}.

gather([Pid | T], Ref) ->
   receive
      {Pid, Ref, Ret} -> [Ret|gather(T, Ref)]
   end;
gather([], _) ->
   [].
```

```
> timer:tc(lib_misc, map, [fun fib:fib/1,
                           [40||_<-lists:seq(1,80)]]).
{1609146911,                                                    26,8 minut
 [102334155,102334155,102334155,102334155,102334155,
  102334155,102334155,102334155,102334155,102334155,102334155,
  102334155,102334155,102334155,102334155,102334155,102334155,
  102334155,102334155,102334155,102334155,102334155,102334155,
  102334155,102334155,102334155,102334155|...]}
```




```
> timer:tc(lib_misc, pmap, [fun fib:fib/1,
                            [40||_<-lists:seq(1,80)]]).
{30516393,                                                      30.5 sekundy
 [102334155,102334155,102334155,102334155,102334155,
  102334155,102334155,102334155,102334155,102334155,102334155,
  102334155,102334155,102334155,102334155,102334155,102334155,
  102334155,102334155,102334155,102334155,102334155,102334155,
  102334155,102334155,102334155,102334155|...]}
```



Elementy języka **Prolog**

- Cechy języka
- Podstawy
- Zamrażanie celów (korutyny)
- Programowanie ograniczeń

Cechy języka

- Deklaratywne programowanie w logice
- Formuły rachunku predykatów
- Rezolucja liniowa
- Atrybuty zmiennych

Podstawy

Termy

- zmienne: X, _, _N
- stałe: janek, 'Janek', []
- termy złożone: f(t1, t2, …, tn)

Stałe interpretujemy jako obiekty (indywidua, rzeczy, dane, itp.).

Funktory n-argumentowe interpretujemy jako n-argumentowe funkcje.

janek — chłopiec o imieniu Janek

ojciec/1 — jednoargumentowy funktor interpretowany jako funkcja przyporządkowująca osobie jej ojca

matka/1 — jednoargumentowy funktor interpretowany jako funkcja przyporządkowująca osobie jej matkę

term	interpretacja
janek	chłopiec Janek
matka(janek)	matka Janka
ojciec(janek)	ojciec Janka
matka(matka(janek))	babcia Janka ze strony matki
matka(ojciec(janek))	babcia Janka ze strony ojca
ojciec(matka(janek))	dziadek Janka ze strony matki
ojciec(ojciec(janek))	dziadek Janka ze strony ojca

Listy

Funktor ./2 łączy głowę H z ogonem listy T w listę .(H, T)

[] stała reprezentująca listę pustą

Zapis uproszczony listy: [e1, e2, …, en] [e1, e2 | T]

```
.(1, .(2, .(3, .(4, [])))) = [1,2,3,4]
                           = [1,2,3,4 | []]
                           = [1,2,3 | [4]]
                           = [1,2 | [3,4]]
                           = [1 | [2,3,4]]
```

Unifikacja

Podstawienie $\sigma = \{X_1/t_1, X_2/t_2, \ldots, X_m/t_m\}$.

Zastosowanie t σ podstawienia σ do termu t:

$$t\sigma = \begin{cases} X & \text{gdy } t = X \text{ i nie jest podstawiane w } \sigma \\ t_i & \text{gdy } t = X_i \\ c & \text{gdy } t = c \\ f(s_1\sigma, \ldots, s_n\sigma) & \text{gdy } t = f(s_1, \ldots, s_n) \end{cases}$$

Przykład:

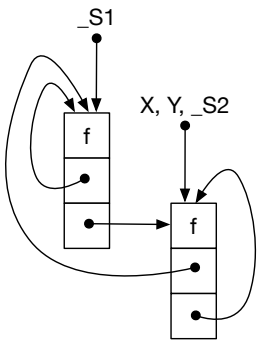
$$f(X, g(Y, Z))\{X/a, Y/b\} = f(a, g(b, Z))$$

Unifikatorem dwóch termów *t* i *s* jest takie podstawienie σ, że $t\sigma$ i $s\sigma$ są identyczne.

Algorytm Martelli-Montanari (złożoność liniowa, średnia stała):

```
S←{s=t}  // układ równań
while S zmienia się do
  if f(s1,…,sn)=f(t1,…,tn)∈S then
    zastąp f(s1,…,sn)=f(t1,…,tn)
    przez s1=t1, …, s1=tn
  if f(s1,…,sm)=g(t1,…,tn)∈S then
    zakończ z niepowodzeniem
  if X=X∈S then usuń X=X z S
  if t=X∈S i t nie jest zmienną then
    zastąp t=X przez X=t
  if X=t∈S i X występuje w t then    // occurs check
    zakończ z niepowodzeniem
  if X=t∈S i X występuje gdzieś w S then
    zastosuj do innych równań {X/t}
end while
```

```
?- f(X) = f(g(a)).
X = g(a)
?- f(X, Y, Z) = f(a, g(X, X), g(Y, Y)).
X = a
Y = g(a, a)
Z = g(g(a, a), g(a, a))
?- X = f(X).
X = f(X) % term cykliczny gdyż Prolog nie wykonuje occurs check
?- X = f(X, Y), Y = f(X, Y).
X = Y, Y = _S2, % where
    _S1 = f(_S1, _S2),
    _S2 = f(_S1, _S2).
```




```
?- X = f(X, Y), Y = f(X, Y), Z = Z, Z).
X = Y, Y = Z, Z = f(Z, Z).
```

Programy

Program składa się z faktów i reguł.

```
lubi(przemko, programowanie).
```
Przemko lubi programowanie.

```
lubi(przemko, X) :-
    lubi(X, programowanie).
```
Przemko lubi tych, którzy lubią programowanie.

```
?- lubi(przemko, X).
```
Co lubi Przemko?

```
X = programowanie ;
X = przemko ;
false.
?- lubi(X, przemko).
```
Kto lubi Przemka?

```
X = przemko ;
false.
```

```
member(X, [X | _]).
member(X, [_ | Y]) :-
    member(X, Y).

?- member(2, [1, 2, 3]).
true.
?- member(X, [1, 2, 3]).
X = 1 ;
X = 2 ;
X = 3 ;
false.
```

Konkatenacja w Haskellu:

```
append :: [a] -> [a] -> [a]
append [] xs = xs
append (x:xs) ys = x:append xs ys
```

Konkatenacja w Prologu:

```
append([], Xs, Xs).
append([X | Xs], Ys, [X | Zs]) :-
    append(Xs, Ys, Zs).

?- append([1, 2, 3], [4, 5, 6], X).
X = [1, 2, 3, 4, 5, 6].
?- append(X, Y, [1, 2, 3]).
X = [], Y = [1, 2, 3] ;
X = [1], Y = [2, 3] ;
X = [1, 2], Y = [3] ;
X = [1, 2, 3], Y = [] ;
false.
```

```
select(X, [X | Xs], Xs).
select(X, [Y | Ys], [Y | Zs]) :-
    select(X, Ys, Zs).

?- select(X, [1, 2, 3], L).
X = 1, L = [2, 3] ;
X = 2, L = [1, 3] ;
X = 3, L = [1, 2] ;
false.
?- select(0, X, [1, 2, 3]).
X = [0, 1, 2, 3] ;
X = [1, 0, 2, 3] ;
X = [1, 2, 0, 3] ;
X = [1, 2, 3, 0] ;
false.
```

Film Woody Allena „Seks nocy letniej”.

Maszynka do wyciągania ości z ryby.

Ta sama maszynka może również wkładać ości w rybę.

```
permutacja([], []).
permutacja(Xs, [X | Zs]) :-
    select(X, Xs, Ys),
    permutacja(Ys, Zs).
```

Wyjmowanie głowy permutacji z danej listy.

```
?- permutacja([1, 2, 3], X).
X = [1, 2, 3] ;
X = [1, 3, 2] ;
X = [2, 1, 3] ;
X = [2, 3, 1] ;
X = [3, 1, 2] ;
X = [3, 2, 1] ;
false.
```

Kolejność leksykograficzna.

```
permutacja([], []).
permutacja([X | Xs], Zs) :-
    permutacja(Xs, Ys),
    select(X, Zs, Ys).
```

Wkładanie głowy danej listy do permutacji.

```
?- permutacja([1, 2, 3], X).
X = [1, 2, 3] ;
X = [2, 1, 3] ;
X = [2, 3, 1] ;
X = [1, 3, 2] ;
X = [3, 1, 2] ;
X = [3, 2, 1] ;
false.
```

Inna kolejność.

Negacja

\+ Warunek

Uwaga: koniunkcja warunków zawierających negację nie jest przemienna.

```
?- member(X, [1, 2]), \+ member(X, [2, 3]).
X = 1 ;
false.
?- \+ member(X, [2, 3]), member(X, [1, 2]).
false.
```

Zagadka z książki Smullyana pod tytułem „Jaki jest tytuł tej książki?”:
Na wyspie żyją rycerze, którzy zawsze mówią prawdę, i łotrzy, którzy zawsze kłamią. Spotykamy trzech jej mieszkańców **A**, **B** i **C**. Pytamy się **A** kim on jest. Ten odpowiada ale nie zrozumieliśmy odpowiedzi. Pytamy się więc pozostałych co powiedział **A**. **B** odpowiada, że **A** powiedział, że jest łotrem. Na co **C** mówi by nie wierzyć **B**, bo **B** jest łotrem.
Kim są **B** i **C**?

```
rycerz(rycerz).
łotr(łotr).
powiedział(rycerz, X) :- X.
powiedział(łotr, X) :- \+ X.

?- powiedział(B, powiedział(A, łotr(A))),
   powiedział(C, łotr(B)).
B = łotr, C = rycerz
```

Implikacja

```
(G1 -> G2; G3)
```

Jeśli cel G1 jest spełniony, to sprawdź cel G2. W przeciwnym przypadku sprawdź cel G3.

```
max(X, Y, Z) :-
    (   X > Y
    ->  Z = X
    ;   Z = Y).
```

Arytmetyka

```
?- X = 2 + 2.
X = 2 + 2.    % !?!
?- X is 2 + 2.
X = 4.
?- 2 +2 =:= 2 * 2.
true.
```

operator	znaczenie
+	suma
-	różnica
*	iloczyn
/	iloraz
mod	modulo
div	dzielenie całkowite
rem	reszta z dzielenia

relacja	znaczenie
=:=	równe
=\=	różne
<	mniejsze
>	większe
=<	mniejsze lub równe
>=	większe lub równe

Aksjomatyka Peano

```
nat(0).
nat(X) :- nat(Y), X is Y+1.
```

Zero jest liczbą naturalną.

Następnik liczby naturalnej jest liczbą naturalną.

```
?- nat(X).
X = 0 ;
X = 1 ;
X = 2 ;
X = 3 ;
… % nieskończenie wiele odpowiedzi
```

Ciąg Fibonacciego

```
fib(0, 0).
fib(1, 1).
fib(N, F) :-
    N > 1, N1 is N-1, N2 is N-2,
    fib(N1, F1), fib(N2, F2),
    F is F1+F2.
```

```
?- time(fib(29, X)).
% 3,328,157 inferences, 71.688 CPU in 76.628 seconds (94% CPU, 46425 Lips)
X = 514229
```

Tablicowanie wyników

```
:- dynamic fib/2.

fib(0, 0).
fib(1, 1).
fib(N, F) :-
    N > 1, N1 is N-1, N2 is N-2,
    fib(N1, F1), fib(N2, F2),
    F is F1+F2,
    asserta(fib(N, F)).
```

Dopisanie faktu na początku definicji.

```
?- time(fib(29, X)).
% 141 inferences, 0.000 CPU in 0.000 seconds (83% CPU, 2350000 Lips)
X = 514229 .

?- time(fib(29, X)).
% 0 inferences, 0.000 CPU in 0.000 seconds (55% CPU, 0 Lips)
X = 514229 .
```

Generowanie wyrazów ciągu Fibonacciego

```
fib(0).
fib(1).
fib(X) :-
    fib(0, 1, X).

fib(A, B, X) :-
    S is A + B,
    (   X = S
    ;   fib(B, S, X)).

?- fib(X).
X = 0 ;
X = 1 ;
X = 1 ;
X = 2 ;
X = 3 ; % nieskończenie wiele odpowiedzi
```

Zamrażanie celów (korutyny)

```
?- X is 2, X > 1.
X = 2.
?- X > 1, X is 2.
ERROR…
```

Koniunkcja warunków zawierających operacje lub relacje arytmetyczne nie jest przemienna.

Należy odroczyć (zamrozić) sprawdzenie warunku do momentu kiedy zmienna będzie miała ustaloną wartość.

```
?- freeze(X, X > 1), X is 2.
X = 2.
```

freeze(Zmienna, Cel) odracza sprawdzenie celu aż zmienna przyjmie wartość.

Zamrożenie ze względu na wszystkie zmienne X_1, …, X_n:

```
freeze(X1, freeze(X2, freeze(X3, …)))
```

Aby zamrozić ze względu na co najmniej jedną ze zmiennych X_1, …, X_n trzeba skorzystać z predykatu **when**:

```
when((nonvar(X1); nonvar(X2); …; nonvar(Xn)), Cel)
```

when(Warunek, Cel) zamraża sprawdzenie celu aż warunek zostanie spełniony.

Strumień danych organizujemy, podobnie jak w języku Oz, w postaci list otwartych:

_ pusty strumień, w którym nie pojawiła się jeszcze żadna wartość;

[1 | _] strumień, w którym pojawiła się liczba 1 ale nie ma na razie kolejnych danych;

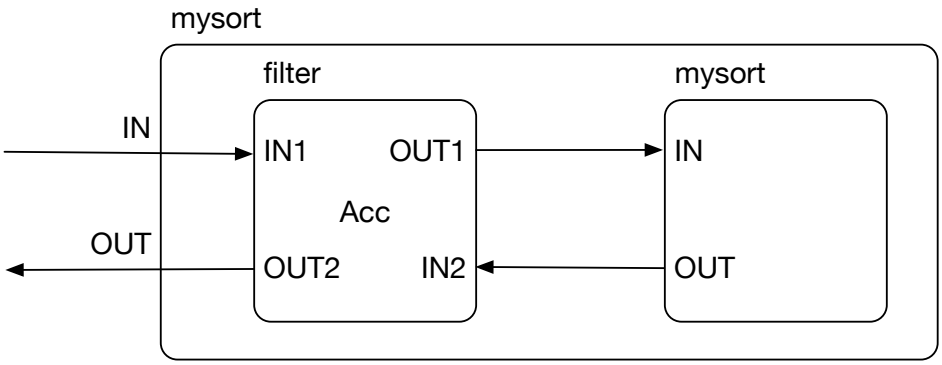
[] strumień zamknięty, w którym nie pojawi się już żadna dana;

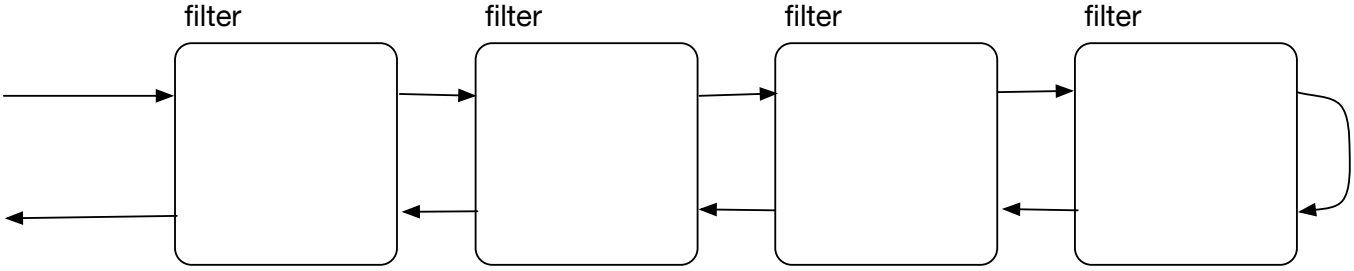
[1, 2, 3] strumień zamknięty, w którym pojawiły się trzy liczby.

```
% strumienie.pl

main(N) :-
    drukowanie(S1),
    podwajanie(S2, S1),
    numlist(1, N, S2). % generowanie liczb od 1 do N

podwajanie(IN, OUT) :-
    freeze(IN, % czekaj na liczbę w strumieniu IN
           (   IN = [H | IN_] % jeśli nowa liczba H
           ->  H2 is 2 * H,
               OUT = [H2 | OUT_], % wyślij podwojoną
               podwajanie(IN_, OUT_)
           ;   OUT = [])).

drukowanie(IN) :-
    freeze(IN,
           (   IN = [H | IN_]
           ->  writeln(H),
               drukowanie(IN_)
           ;   true)).
```

```
?- main(10).
2
4
6
8
10
12
14
16
18
20
?-
```

ZERO-TIME SORTING NETWORK





```
% mysort.pl

mysort(IN, OUT) :-
    freeze(IN,
           (   IN = [H | IN_]
           ->  filter(H, IN_, OUT1, IN2, OUT),
               mysort(OUT1, IN2)
           ;   OUT = [])).
```

```
filter(Acc, IN1, OUT1, IN2, OUT2) :-
    freeze(IN1,
           (   IN1 = [H | IN1_]
           ->  (   H >= Acc
               ->  OUT1 = [H | OUT1_],
                   filter(Acc, IN1_, OUT1_, IN2, OUT2)
               ;   OUT1 = [Acc | OUT1_],
                   filter(H, IN1_, OUT1_, IN2, OUT2))
           ;   OUT1 = [],
               OUT2 = [Acc | IN2])).
```

```
?- mysort(X, Y).
freeze(X, …).

?- mysort(X, Y), X = [2 | A].
X = [2|A],
freeze(A,…),
freeze(_1796, …).

?- mysort(X, Y), X = [2 | A], A = [4 | B].
X = [2, 4|B],
A = [4|B],
freeze(B, …),
freeze(_542, …),
freeze(_668, …).



?- mysort(X, Y), X = [2 | A], A = [4 | B],
   B = [1 | C].
X = [2, 4, 1|C],
A = [4, 1|C],
B = [1|C],
freeze(C,),
freeze(_1292, …),
freeze(_1436, …),
freeze(_1580, …).

?- mysort(X, Y), X = [2 | A], A = [4 | B],
   B = [1 | C], C = [].
X = [2, 4, 1],
Y = [1, 2, 4],
A = [4, 1],
B = [1],
C = [].
```

Programowanie ograniczeń

W Prologu jest możliwość narzucania ograniczeń na zmienne nie związane jeszcze z wartością. W tym celu można użyć pakietu **clpfd** (ang. *Constraint Logic Programming over Finite Domains*).

```
?- use_module(library(clpfd)).
true.

?- X #> 1, X #= 2.
X = 2.

?- X #>2, X #< 6, X #\= 4.
X in 3\/5.
```

X in 1..10	zdefiniowanie dziedziny
[X1, X2, …, Xn] ins 1..10	zdefiniowanie dziedzin
#=, **#\=**, **#<**, **#>**, **#=<**, **#>=**	relacje między wartościami

indomain(X)
label([X1, X2, …, Xn])
labeling([Opcja, …], [X1 X2, …, Xn])
Możliwe opcje:

- wybór zmiennej
  leftmost (domyślna), **ff**, **ffc**, **min**, **max**
- wybór wartości
  up (domyślna), **down**
- strategie podziału
  step (domyślna), **enum**, **bisect**
- kolejność rozwiązań
  min(Wyrażenie), **max(Wyrażenie)**

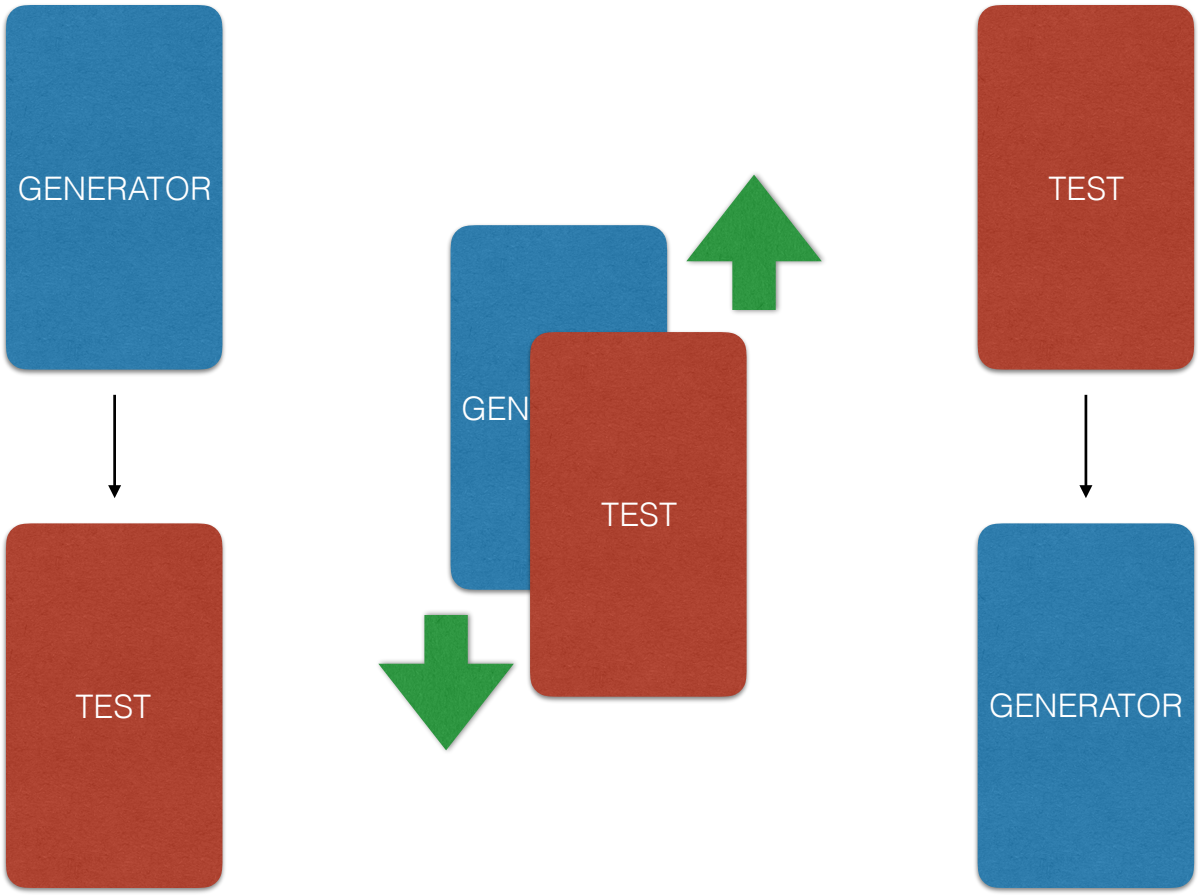
W Prologu podstawową metodą poszukiwania rozwiązania jest „generowanie i testowanie”.

?- GENERATOR, TEST.

Warunek **GENERATOR** dostarcza kolejne rozwiązanie a **TEST** sprawdza czy jest ono dopuszczalne.

W przypadku niepowodzenia następuje nawrót i wycofanie się do generowania kolejnego rozwiązania.

```
hetmany_gt(N, P) :-
    numlist(1, N, X),
    permutation(X, P),   % GENERATOR
    \+ niebezpieczna(P). % TEST

niebezpieczna(P) :-
    append(_, [I | L1], P),
    append(L2, [J | _], L1),
    length(L2, K),
    abs(I - J) =:= K + 1.

?- time(hetmany_gt(8, X)).
% 93,593 inferences, 0.036 CPU in 0.046 seconds (78% CPU, 2592173 Lips)
X = [1, 5, 8, 6, 3, 7, 2, 4] .
?- time(hetmany_gt(10, X)).
% 1,366,508 inferences, 0.488 CPU in 0.948 seconds (51% CPU, 2799648 Lips)
X = [1, 3, 6, 8, 10, 5, 9, 2, 4|...] .
?- time(hetmany_gt(12, X)).
% 90,688,062 inferences, 26.720 CPU in 39.410 seconds (68% CPU, 3393964 Lips)
X = [1, 3, 5, 8, 10, 12, 6, 11, 2|...] .
?- time(hetmany_gt(14, X)).
^CAction (h for help) ? abort
% 284,900,718 inferences, 81.443 CPU in 131.441 seconds (62% CPU, 3498158 Lips)
% Execution Aborted
```




```
:- use_module(library(clpfd)). % dyrektywa dla kompilatora

hetmany(N, P) :-
    length(P, N),
    P ins 1..N,
    bezpieczna(P),        % TEST
    labeling([ffc], P).   % GENERATOR

bezpieczna([]).
bezpieczna([I | L]) :-
    bezpieczna(L, I, 1),
    bezpieczna(L).

bezpieczna([], _, _).
bezpieczna([J | L], I, K) :-
    I #\= J,
    abs(I - J) #\= K,
    K1 is K + 1,
    bezpieczna(L, I, K1).
```

```
?- time(hetmany(10, X)).
% 42,161 inferences, 0.006 CPU in 0.007 seconds (89% CPU, 6709262 Lips)
X = [1, 3, 6, 9, 7, 10, 4, 2, 5|...] .

?- time(hetmany(20, X)).
% 230,209 inferences, 0.033 CPU in 0.036 seconds (91% CPU, 6941324 Lips)
X = [1, 3, 5, 14, 17, 4, 16, 7, 12|...] .

?- time(hetmany(40, X)).
% 562,761 inferences, 0.080 CPU in 0.084 seconds (95% CPU, 7051171 Lips)
X = [1, 3, 5, 26, 33, 4, 28, 7, 34|...] .

?- time(hetmany(80, X)).
% 2,158,637 inferences, 0.318 CPU in 0.334 seconds (95% CPU, 6784838 Lips)
X = [1, 3, 5, 44, 42, 4, 50, 7, 68|...] .

?- time(hetmany(160, X)).
% 10,400,403 inferences, 1.619 CPU in 1.680 seconds (96% CPU, 6423880 Lips)
X = [1, 3, 5, 65, 68, 4, 74, 7, 85|...] .
```

all_different([X1, …, Xn])
all_distinct([X1, …, Xn])

sum([X1, X2, …, Xn], Rel, Expr)
scalar_product([C1, …, Cn], [X1, …, Xn], Rel, Expr)

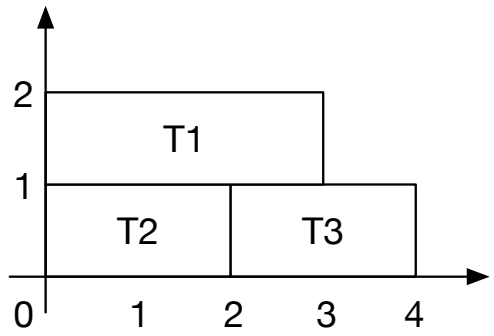
serialized([S1, …, Sn], [D1, …, Dn])

rozłączność przedziałów (Si, Si+Di)
Si + Di #=< Sj lub Sj + Dj #=< Si

cumulative([T1, …, Tn], [limit(L)])

zadania T1, …, Tn
Ti = task(Si, Di, Ei, Ci, IDi)
w żadnej chwili nie przekroczono L jednostek zasobu

```
tasks_starts(Tasks, [S1, S2, S3]) :-
    Tasks = [task(S1, 3, _, 1, _),
             task(S2, 2, _, 1, _),
             task(S3, 2, _, 1, _)].

?- tasks_starts(Tasks, Starts),
   Starts ins 0..10,
   cumulative(Tasks, [limit(2)]),
   label(Starts).
Tasks = [task(0, 3, 3, 1, _4380), task(0, 2,
2, 1, _4398), task(2, 2, 4, 1, _4416)],
Starts = [0, 0, 2] .
```



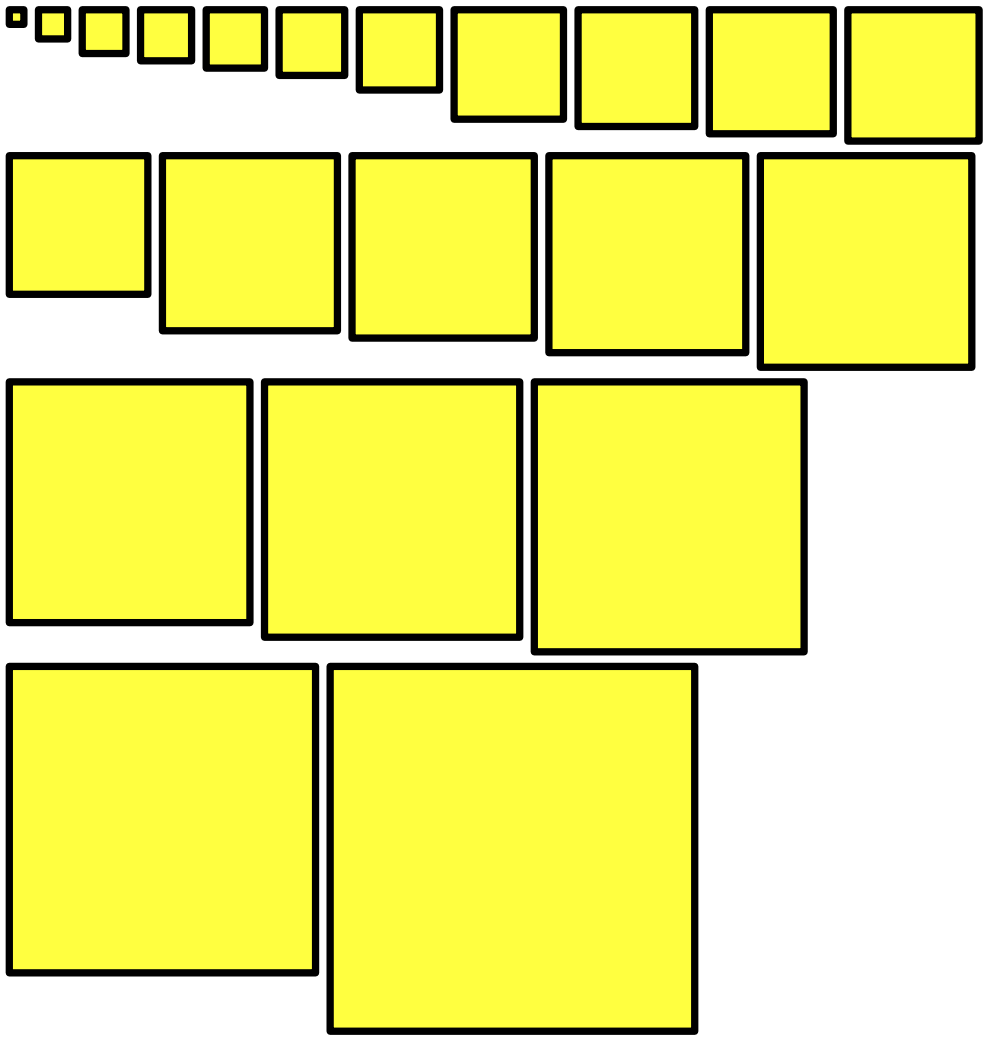
Podział kwadratu na kwadraty

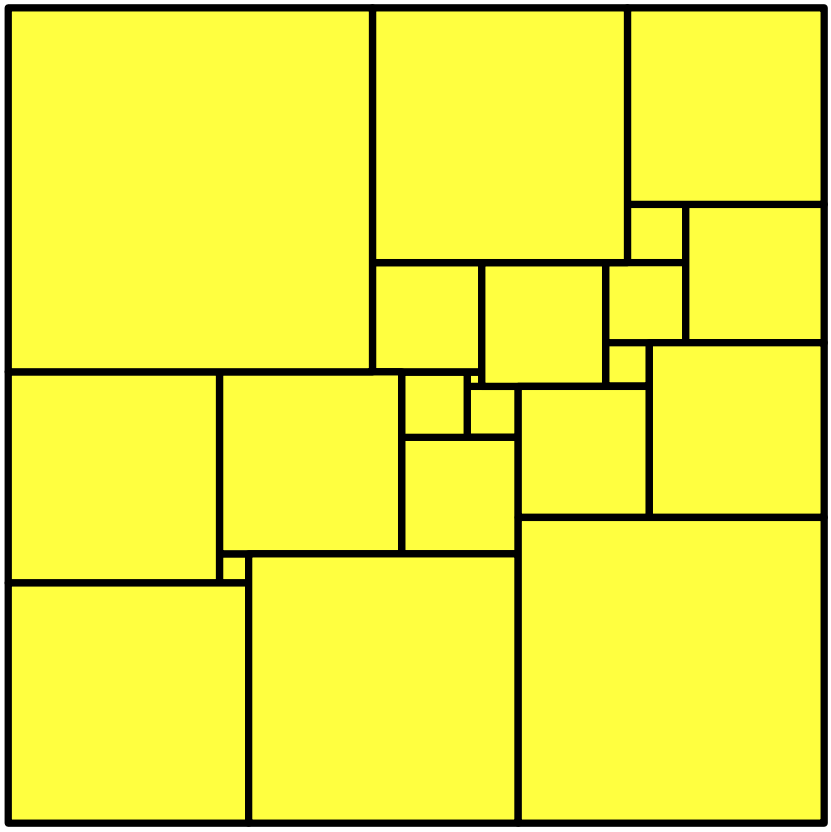
Zadanie: podzielić kwadrat na parami różne kwadraty.

Trywialne rozwiązanie:
jeden kwadrat

Najmniejsze nietrywialne rozwiązanie:
kwadrat 112x112 podzielony na 21 kwadratów o bokach 50, 42, 37, 35, 33, 29, 27, 25, 24, 19, 18, 17, 16, 15, 11, 9, 8, 7, 6, 4, 2

```
% kwadrat.pl

:- use_module(library(clpfd)).

main(Xs) :-
    kwadraty(112,
             [50,42,37,35,33,29,27,25,24,19,18,17,16,15,11,9,8,7,6,4,2],
             Xs).

kwadraty(D, Ds, Xs) :-
    length(Ds, N),
    D1 is D-1,
    length(Xs, N),
    Xs ins 0..D1,
    length(Ys, N),
    Ys ins 1..D,
    zadania(Xs, Ys, Ds, Zadania),
    cumulative(Zadania, [limit(D)]),
    % SICStus Prolog z opcją global(true) w cumulative/2 8500ms
    labeling([ffc], Xs).

zadania([], [], [], []).
zadania([X | L1], [Y | L2], [D | L3], [task(X, D, _, D, _) | L4]) :-
    Y #= X + D,
    zadania(L1, L2, L3, L4).
```





Rozwiązanie znalezione SICStus Prologiem.