

# Języki i paradygmaty programowania

Marek Gózdź

11 czerwca 2024

## Literatura podstawowa

AUT J.E. Hopcroft, R. Motwani, J.D. Ullman, *Wprowadzenie do teorii automatów, języków i obliczeń*, wyd. 2, PWN, W-wa 2012

7J B.A. Tate, *Siedem języków w siedem tygodni. Praktyczny przewodnik nauki języków programowania*, Helion 2011

## Literatura dodatkowa

- J. Bylina, B. Bylina, *Przegląd języków i paradygmatów programowania*, skrypt UMCS, Lublin 2011
- M. Lipovača, *Learn You a Haskell for Great Good! A Beginner's Guide*, No Starch Press 2011  
(dostępny również on-line: [learnyouahaskell.com](http://learnyouahaskell.com))
- W.F. Clocksin, C.S. Mellish, *Prolog. Programowanie*, Helion 2003  
(druk na żądanie)
- dokumentacja i samouczki do języków: Ruby, Haskell i Prolog

## Uwaga 1

Warunki zaliczenia:

- 1 zaliczone ćwiczenia
- 2 zdany egzamin

## Uwaga 2

5 ECTS = 125-150 godzin, czyli po 7-10 godzin pracy tygodniowo!

## Uwaga 3

Wykład dostępny jest na `kft.umcs.lublin.pl/mgozdz/`

Dlaczego

*„Języki i paradygmaty programowania”,*

a nie

*„Paradygmaty i języki programowania”?*

O jakie języki chodzi?

Proces tworzenia programu:

- ❶ cel – co program ma robić
- ❷ projekt – dobór algorytmu
- ❸ opis – dobór głównego paradygmatu
- ❹ technika – dobór technik programowania
- ❺ narzędzia – wybór języka programowania
- ❻ wykonanie – napisanie kodu, sprawdzenie, uruchomienie

Punkty 1–5 to praca koncepcyjna, dopiero ostatni punkt to klepanie w klawiaturę.

Paradygmat określa ogólny sposób podejścia do opisu problemu. Jeśli

- znam algorytm, który jest liniowy (ew. drzewiasty) i niezbyt złożony, to wybieram czysty paradygmat imperatywny
- algorytm zawiera wiele powtarzających się fragmentów, wygodnie jest sięgnąć po paradygmat strukturalny/proceduralny
- algorytm zawiera klasy abstrakcji, klasyfikujące dane po ich właściwościach – paradygmat obiektowy
- nie wiem jak, ale wiem z czym startuję, co mogę zrobić i co chcę uzyskać na końcu – paradygmat deklaratywny
  - ▶ mogę wszystko sprowadzić do matematycznego opisu – paradygmat deklaratywny funkcyjny
  - ▶ mogę wszystko sprowadzić do wyrażeń Boole'a – paradygmat deklaratywny w logice

Nie jest prawdą, że każdy język programowania jest tak samo dobry do realizacji danego zadania, co inny. Istnieją wyspecjalizowane języki ułatwiające rozwiązywanie konkretnych typów problemów.

Wybór odpowiedniego narzędzia jest ważny!

**Przykład:** wyświetl drugie i trzecie pole każdej linii (pola oddzielane są dwukropkiem) pliku tekstowego

```
1 awk 'BEGIN{FS=":"};{print $2,$3}' plik.txt
```

A teraz proszę napisać to samo w C.

Skoro konstrukcja języka programowania podlega paradygmatowi, dlaczego  
„*Języki i paradygmaty programowania*”,

a nie

„*Paradygmaty i języki programowania*”?

Ponieważ zajmować się będziemy głównie abstrakcyjnymi strukturami i właściwościami *języków formalnych* opisujących *automaty*, *wyrażenia regularne* i *gramatyki*. Te struktury pozwalają na stworzenie spójnego języka programowania, ale mogą służyć również do opisu języków naturalnych i innych.

To nie jest kolejny wykład z języka programowania!

*Chociaż dwa nowe języki się pojawią...*



## Plan wykładów

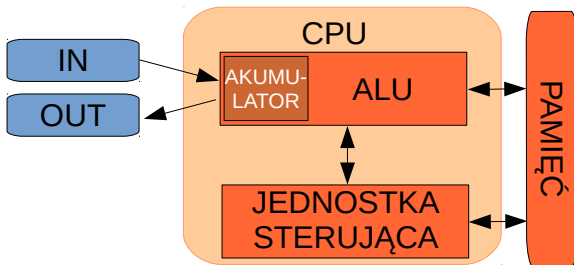
- Maszyny programowalne, języki programowania
- Podstawowe paradygmaty programowania:
  - ▶ imperatywny
  - ▶ imperatywny obiektowy, Ruby  $\rightarrow$ 7J[2.4]
  - ▶ deklaratywny funkcyjny, Haskell  $\rightarrow$ 7J[8.x]
  - ▶ deklaratywny w logice, Prolog  $\rightarrow$ 7J[4.x]
- Automaty, alfabety, języki  $\rightarrow$ AUT[1.5]
- Automaty deterministyczne  $\rightarrow$ AUT[2.2]
- Automaty niedeterministyczne  $\rightarrow$ AUT[2.3],[2.5]
- Wyrażenia regularne  $\rightarrow$ AUT[3.1]–[3.4]
- Gramatyki bezkontekstowe  $\rightarrow$ AUT[5.1],[5.2]
- Kompilator i interpreter. Analiza leksykalna i syntaktyczna kodu  $\rightarrow$ AUT[5.3] i inne

- Wieloznaczność gramatyk  $\rightarrow$  AUT[5.4]
- Automaty ze stosem  $\rightarrow$  AUT[6.1]–[6.4]
- Zmienne i typy danych
- Programowanie funkcyjne: rachunek lambda
- Programowanie funkcyjne: monady  $\rightarrow$  7J[8.4]
- Programowanie funkcyjne: Haskell  $\rightarrow$  7J[8.1]–[8.5] i inne
- Programowanie w logice: klauzule Horna, funkcje i relacje
- Programowanie w logice: nawracanie
- Programowanie w logice: wnioskowanie góra-dół i dół-góra
- Programowanie w logice: Prolog  $\rightarrow$  7J[4.1]–[4.5] i inne

## Programowanie maszyn

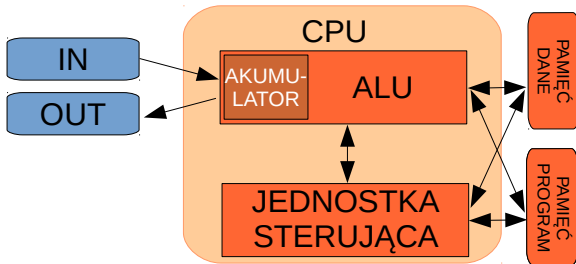
- sprzęt musi być programowalny
  - ▶ nie każdy układ elektroniczny to procesor
  - ▶ sam procesor to jeszcze nie komputer
- zazwyczaj nie programujemy elektroniki bezpośrednio – warstwy abstrakcji oddzielające użytkownika od sprzętu
  - ▶ systemy wbudowane (np. kontrolery przemysłowe, telefony komórkowe)
  - ▶ systemy operacyjne (np. bankomaty, komputery)
- do wydawania instrukcji używany jest język programowania (oparty o jakiś paradygmat)
  - ▶ język maszynowy – asembler
  - ▶ języki wyższego poziomu
  - ▶ kompilator
  - ▶ interpreter
  - ▶ maszyna wirtualna

## Architektura Princeton (koncepcja von Neumanna)



- pamięć przechowuje zarówno dane jak i instrukcje
- CPU nie może jednocześnie pobrać instrukcji i wykonać operacji na danych

## Architektura Harward



- pamięć danych i pamięć rozkazów są rozdzielone
- CPU może jednocześnie dokonywać odczytu/zapisu do pamięci i wykonywać instrukcje programu

Tradycyjnie mówi się, że **język programowania** musi dać możliwość

- operowania na zmiennych
- operowania na podstawowych strukturach danych
- wykonania instrukcji warunkowej
- wykonania pętli lub ew. zdefiniowania struktur rekurencyjnych

tak, aby dało się w nim zapisać dowolny algorytm (język uniwersalny).

Nie każdy język umożliwia programowanie:

- języki znacznikowe (np. HTML)
- część języków skryptowych
- większość makrojęzyków

Nie każde podanie ciągu instrukcji jest programowaniem!

## Paradygmat

*«przyjęty sposób widzenia rzeczywistości w danej dziedzinie, doktrynie»*  
[sjp.pwn.pl]

### Paradygmaty programowania

- paradygmat jest to wzorzec, teoria i metodologia
- coś narzucone, dane odgórnie
- na początku paradygmat programowania był determinowany przez architekturę maszyny
- obecnie ten problem został zepchnięty na poziom kompilatora – często języki są uniwersalne, kod jest przenośny pomiędzy maszynami
- dobry przykład: jądro linuxa (napisane w C, można skompilować na kilkunastu różnych architekturach)

## Podstawowy podział paradygmatów

### Imperatywny

- ściśle oparty na architekturze von Neumanna
- zmiana *stanu* programu wraz z kolejnymi *krokami*
- struktury kontrolne programu określają kolejność wywołań instrukcji
- programy przypominają instrukcję obsługi, czyli musimy znać metodę osiągnięcia celu (algorytm prowadzący od danych do wyniku)

### Deklaratywny

- wzorowany na językach naturalnych i ludzkich procesach myślowych
- *nieliniowość* wykonania, komenda na końcu kodu może zmodyfikować wywołanie z jego początku
- powszechne stosowanie rekurencji w miejsce pętli
- definiuje problem (dane i reguły) i poszukuje odpowiedzi na zadane pytanie



## Dalszy podział paradygmatu imperatywnego

- **strukturalny** – bazuje na strukturze kodu (bloki), pozbawiony jest instrukcji skoku goto; kod jest czystszy i łatwiejszy do analizy, program, za wyjątkiem pętli, wykonywany jest po kolei
- **proceduralny** – odmiana strukturalnego, w której część kodu jest zestawiana w podprogramy:

- ▶ procedury,
- ▶ funkcje;

podprogramy mogą być wywoływane wielokrotnie z głównego programu lub z innych podprogramów

- **obiektowy** – definiuje się metody działania na klasach danych, nie na pojedynczych instancjach; przynależność do klasy, a więc właściwości, obiekt dziedziczy hierarchicznie

## Dwa główne nurty programowania deklaratywnego

- **funkcyjne** – zamiast instrukcji używa się *wyrażeń*, a więc nie ma pojęcia stanu programu; cały algorytm sprowadza się do obliczenia wartości pewnej (zazwyczaj złożonej) funkcji matematycznej; u podstaw programowania funkcyjnego leży rachunek lambda
- **w logice** – zamiast instrukcji używa się *zdań logicznych*, a więc każde działanie sprowadza się do określenia, czy rozważane stwierdzenie jest logicznie prawdziwe, czy fałszywe

Obecnie wyróżnia się cztery główne paradygmaty, często traktowane jako niezależne:

- imperatywny
- obiektowy
- funkcyjny
- w logice

Większość języków (niemal wszystkie nowoczesne) jest hybrydowa i nie podlega dokładnie pod tylko jeden paradygmat.

## Program w języku imperatywnym

- ściśle powiązany z koncepcją maszyny von Neumanna
- polega na wydaniu sekwencji uszeregowanych czasowo rozkazów, zmieniających stan maszyny
- stan maszyny oznacza stan jej procesora i zawartość pamięci oraz rejestrów
- języki niskiego poziomu (asemblery) operują niemal bezpośrednio na jednostkach pamięci
- języki wysokiego poziomu wprowadzają twory abstrakcyjne reprezentujące składowe maszyny, np. zmienne reprezentują jednostki pamięci

## Przykład: kod w Pascalu

```

1  program silnia;
2  var i, n, s: integer;
3  begin
4      read(n);
5      s := 1;
6      for i := 2 to n do
7          s := s * i;
8      write(s)
9  end.

```

## Przykład: kod w Fortranie

```

1  program silnia
2  integer i, n, s
3  read(*,*) n
4  s=1
5  do 10 i=2,n
6      s = s * i
7 10 continue
8  write(*,*) s
9  end program

```

Języki czysto imperatywne: C, Pascal, Fortran, Basic, asemblery...

## Program w języku obiektowym

- przede wszystkim jest to rodzina imperatywna, więc struktura i logika kodu jest taka sama
- zmiana następuje na poziomie struktury danych: dane grupowane są z operacjami, które można na nich wykonać, w nowy twór – **obiekt**
- **hermetyzacja (enkapsulacja)** polega na ukryciu implementacji obiektu, zaś udostępnieniu jedynie niektórych danych i metod w postaci **interfejsu**
- obiekty można grupować w hierarchicznie definiowane klasy, które mają pewne wspólne cechy (mechanizm dziedziczenia w ramach hierarchii)
- hierarchia klas implikuje zawieranie się, co prowadzi do **polimorfizmu dynamicznego** – metody dobierane są automatycznie zgodnie z przynależnością do klasy
- **abstrakcja danych** pozwala na definiowanie klas w postaci ogólnych szablonów, a na ich podstawie dopiero tworzenia szczegółowych definicji

Główne pojęcia związane z paradygmatem obiektowym:

- **Klasa** to abstrakcyjny typ danych, definiowany programowo poprzez podanie dopuszczalnych na nim operacji oraz jego reprezentację.
- **Obiekt** to element należący do pewnej klasy.
- **Klasa potomna** (pochodna, podklasa) jest klasą wywiedzioną poprzez dziedziczenie z innej klasy.
- **Klasa macierzysta** (nadrzędna, nadklasa) jest klasą, z której wywodzi się klasa potomna.
- **Metoda** to podprogram (operacja) działający na obiektach danej klasy zgodnie z jej i jego definicją.
- **Pole** to zmienna zamknięta wewnątrz obiektu.
- **Komunikat** to wywołanie metody.
- **Protokół obiektu** to zbiór sposobów odwołania się do obiektu.

To, co jest w ramach danego obiektu ukryte, a co dostępne, określają *kwalifikatory* przy metodach i polach:

- `public` oznacza nieograniczoną dostępność z zewnątrz
- `private` oznacza ukrycie przed dostępem z zewnątrz i pozwala przez to na definiowanie typów abstrakcyjnych
- `protected` oznacza ukrycie dla wszystkich za wyjątkiem klas potomnych

W wielu językach kwalifikatory przyjmują domyślne stałe wartości i jawnie się ich nie używa.



Przykład: kod w C++

```
1 unsigned int factorial(unsigned int x)
2 {
3     unsigned int value = 1;
4     for(unsigned int i = 2; i <= x; i++)
5     {
6         value = value * i;
7     }
8     return value;
9 }
```

Jeśli nie wykorzystuje się obiektów, język obiektowy jest nie do odróżnienia od czystego języka imperatywnego.

## Przykład: kod w Ruby

```
1  class Song
2    def initialize(name, artist, duration)
3      @name = name
4      @artist = artist
5      @duration = duration
6    end
7  end
8
9  aSong = Song.new("Bicylops", "Fleck", 260)
10 aSong.inspect
```

Języki obiektowe: C++, C#, Delphi / Object Pascal, Java, Ada, Python, Ruby...

Jest to dominujący obecnie paradygmat.

Ruby jest językiem niezwykle elastycznym, do tego stopnia, że struktura klas nie jest sztywna. Mówi się, że Ruby posiada **otwarte klasy**:

- definicje klas i metod są dostępne dla programisty
- można je (niemal) dowolnie modyfikować...
- ... ponosząc konsekwencje naruszenia struktury

Ingerencja w klasy i metody jest czasami bardzo wygodna, ale programista musi pamiętać, że z powodu dziedziczenia, zmiany w klasie nadrzędnej będą się propagowały do klas potomnych. Może to prowadzić do trudnych do wykrycia błędów.

## Przykład: zdefiniowanie operatora + w obrębie klasy Fixnum

[piotrsarnacki.com/2008/01/08/otwarte-klasy-w-rubim-i-ich-praktyczne-wykorzystanie/](http://piotrsarnacki.com/2008/01/08/otwarte-klasy-w-rubim-i-ich-praktyczne-wykorzystanie/)

Plik klasa.rb

```
1  # -*- coding: iso-8859-2 -*-
2  class Fixnum
3      alias old_plus +
4
5      def +(wyrażenie)
6          (self.old_plus wyrażenie) % 7
7      end
8  end
9
10 a = 4 + 4
11 puts a
```

Uruchomienie programu

```
1  [bash]$ ruby klasa.rb
2  1
```

Programowanie deklaratywne (funkcyjne i logiczne) będzie omawiane dokładniej w dalszej części wykładu. Na razie przykłady:

### Definicja funkcji silnia w Haskellu

```
1  silnia :: Integer -> Integer
2  silnia 0 = 1
3  silnia x = x * silnia (x-1)
```

### Definicja predykatu silnia w Prologu

```
1  silnia(0,X) :- X=1, !.
2  silnia(N,X) :- N>0, L is N-1,
3                  silnia(L,Y), X is Y*N.
```

W językach deklaratywnych bardzo często efektywniejsze są algorytmy rekurencyjne, niż te oparte o pętle. Niekiedy pętli nie da się w ogóle zdefiniować ze względu na brak możliwości użycia zmiennych sterujących przebiegiem pętli.

## UWAGA

Jak na razie wszystkie komputery<sup>\*)</sup> oparte są na imperatywnej architekturze von Neumanna. Dlatego też, niezależnie od użytego języka programowania, kompilator i tak tłumaczy każdy kod na zestaw uszeregowanych czasowo instrukcji zmieniających stan maszyny.

*\*) Wyjątkiem są komputery kwantowe, które są oparte na zupełnie innej zasadzie działania zarówno pod względem budowy, programowania jak i realizacji algorytmów. W ich przypadku jednak póki co nie istnieją języki programowania, podobnie jak nie istnieją jeszcze uniwersalne programowalne komputery kwantowe.*

## Automaty

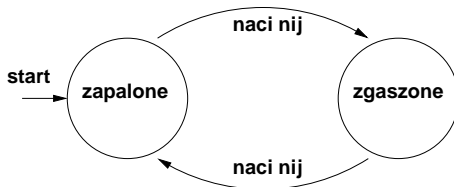
**Automat** to abstrakcyjny model opisu pewnej klasy maszyn obliczeniowych i oprogramowania. Jest użyteczny w przypadku procesów, w których można wyodrębnić zbiór parametrów w całości je definiujący. W przypadku oprogramowania może symulować instrukcje warunkowe, skoki i pętle/rekurencje.

Automat składa się ze zbioru stanów i reguł przejścia między nimi:

- **stan** jest definiowany wartościami parametrów opisujących cały układ; można go porównać z położeniem zatrzymanego układu trybów w mechanizmie, stanu napięcia sprężyn, stanu wahadła itd.
- **reguły przejścia** określają, jak zmieniają się parametry aby dokonało się przejście z pewnego stanu do innego (nie wszystkie przejścia muszą być dozwolone)

**Automat skończony** zawiera skończoną liczbę stanów, w odróżnieniu od automatów nieskończonych, które są nieograniczone. Tymi drugimi nie będziemy się zajmować.

Przykład: w(y)łącznik światła



- „zapalone” i „zgaszone” to dwa możliwe stany przełącznika
- przejście między tymi stanami następuje poprzez zaistnienie sytuacji „naciśnij”
- ta sytuacja jest odczytywana z wejścia automatu (nie zaznaczonego na rysunku), czyli automat odczytuje, analizuje i reaguje na dane wejściowe



## Podstawowe definicje

- **Alfabet**  $\Sigma$  to skończony niepusty zbiór symboli.
- **Słowo** (łańcuch) to ciąg symboli wybranych z pewnego alfabetu.  
Specjalnym słowem jest **łańcuch pusty**  $\varepsilon$ , który należy do każdego alfabetu.
- **Język**  $\mathcal{L}$  to zbiór wszystkich słów podlegających pewnej regule, jakie można ułożyć z danego alfabetu.

### Przykładowe alfabety

$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$	dziesiętny
$\Sigma = \{0, 1\}$	binarny
$\Sigma = \{a, b, \dots, z\}$	małe litery
$\Sigma = \text{ASCII}$	znaki ASCII

Przykładowe słowa w alfabecie binarnym  $\Sigma = \{0, 1\}$ :

$\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \dots$

Wprowadza się **potęgi alfabetu**, oznaczające wybór słów pewnej długości:

$$\Sigma^0 = \{\varepsilon\}$$

$$\Sigma^1 = \{w : |w| = 1\} = \Sigma$$

$$\Sigma^2 = \{w : |w| = 2\} = \Sigma\Sigma$$

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \dots$$

$$\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots = \{\varepsilon\} \cup \Sigma^+$$

Symbol  $\Sigma^*$  w praktyce oznacza zbiór wszystkich słów (również słowa pustego) nad alfabetem  $\Sigma$ . Określenie „słowo nad alfabetem  $\Sigma$ ” oznacza „łańcuch należący do  $\Sigma^*$ ”

## Stwierdzenie

**Konkatenacja**  $ab$  dwóch słów  $a$  i  $b$  nad alfabetem  $\Sigma$  jest słowem nad alfabetem  $\Sigma$ . Jeśli  $a \in \Sigma^n$  i  $b \in \Sigma^m$ , to  $ab \in \Sigma^{n+m}$ . Elementem neutralnym konkatenacji jest  $\varepsilon$ .

## Stwierdzenie

Konkatenacja  $ab$  dwóch słów  $a \in \Sigma_a^n$  i  $b \in \Sigma_b^m$  jest słowem nad alfabetem  $\Sigma_{ab} = (\Sigma_a \cup \Sigma_b)$ .

## Definicja: język nad alfabetem

**Językiem**  $\mathcal{L}$  nad alfabetem  $\Sigma$  nazywamy każde  $\mathcal{L} \subseteq \Sigma^*$ .

- Język nie musi wykorzystywać wszystkich symboli alfabetu, dlatego  $\mathcal{L}$  pozostaje językiem nad każdym alfabetem zawierającym w sobie  $\Sigma$ .
- Zbiór pusty  $\mathcal{L} = \emptyset$  jest językiem nad dowolnym alfabetem.
- $\mathcal{L} = \{\varepsilon\}$  również jest językiem nad dowolnym alfabetem.
- Język nie musi być skończony.

Różnica między  $\varepsilon$ ,  $\{\varepsilon\}$  a  $\emptyset$

- $\varepsilon$  jest łańcuchem pustym, zawierającym zero symboli
- $\{\varepsilon\}$  jest zbiorem jednoelementowym, zawierającym pusty łańcuch
- $\emptyset$  jest zbiorem pustym, zeroelementowym

**Przykład 1:**

Alfabetem jest alfabet łaciński z polskimi znakami diakrytycznymi.

Językiem jest język polski.

Reguła: Każde poprawne po polsku słowo będzie należało do omawianego języka. Reguła przynależności do języka determinowana występowaniem słowa w słowniku.

**Przykład 2:**

Alfabetem jest zbiór cyfr arabskich.

Językiem jest zbiór liczb parzystych.

Reguła: Każdy ciąg cyfr kończący się symbolem 0,2,4,6,8 należy do tego języka.

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$\mathcal{L} = \{xy : x \in \Sigma^*, y \in \{0, 2, 4, 6, 8\}\}$$

**Przykład 3:**

Alfabetem jest zbiór cyfr arabskich.

Językiem jest zbiór liczb naturalnych mniejszych od 1000.

Reguła: Każdy ciąg cyfr o długości nie przekraczającej 3 należy do tego języka.

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$\mathcal{L} = \{w \in \Sigma^* : |w| \leq 3\}$$

## Przykład 4:

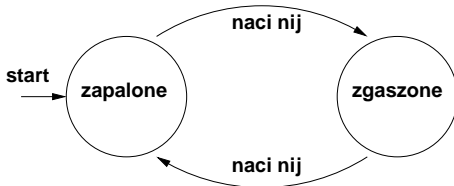
Alfabetem jest alfabet łaciński wraz z cyframi i pewnymi znakami specjalnymi.

Językiem jest zbiór poprawnych wyrażeń języka fortran.

Reguła: musi istnieć słownik oraz pewne reguły konstruowania poprawnych słów (gramatyka)

Problem: jak sprawdzić, czy pewne wyrażenie jest poprawnym wyrażeniem danego języka? Należy skonstruować **automat** rozpoznający poprawne wyrażenia danego języka. Znacznie prościej jest, jeśli język jest skończony. Dlatego też w językach programowania stosuje się reguły ograniczające programistę przy wyborze etykiet (nazw zmiennych, procedur...).

**Automat deterministyczny** to taki, w którym przejścia między stanami są dokładnie określone, tj. dla danego wejścia istnieje dokładnie jeden stan wyjściowy. Automat taki zawsze znajduje się w dobrze określonym stanie. Przykład: włącznik światła



**Automat nondeterministyczny** to taki, w którym w danych warunkach istnieją przejścia do kilku stanów wyjściowych. Automat taki może znajdować się w kilku stanach jednocześnie.

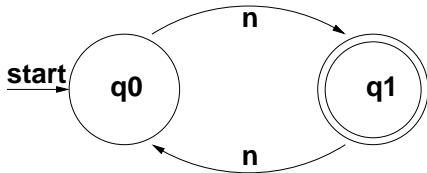


## Definicja: deterministyczny automat skończony

**Deterministyczny automat skończony (DAS)** to automat skończony, deterministyczny, zdefiniowany poprzez  $(Q, \Sigma, \delta, q_0, F)$ , gdzie

- $Q$  to skończony **zbiór stanów**
- $\Sigma$  to skończony zbiór **symboli wejściowych** (alfabet)
- $\delta$  to **funkcje przejścia** między stanami,  $\delta : Q \times \Sigma \rightarrow Q$
- $q_0 \in Q$  to **stan początkowy**
- $F \subset Q$  to zbiór **stanów akceptujących** (stanów końcowych)

Przykład: włącznik światła



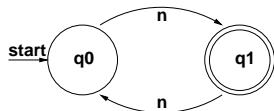
$$Q = \{q_0, q_1\}$$

$$\Sigma = \{n\}$$

$$\delta(q_0, n) = q_1, \quad \delta(q_1, n) = q_0$$

$$F = \{q_1\}$$

Przykład: włącznik światła



	$n$
$\rightarrow q_0$	$q_1$
$*q_1$	$q_0$

Tu  $\rightarrow$  oznacza stan wejściowy, zaś  $*$  znakuje stany akceptowalne.

Alfabet odczytać można z opisu pierwszego rzędu, stany z pierwszej kolumny, funkcje przejścia z treści tabelki.

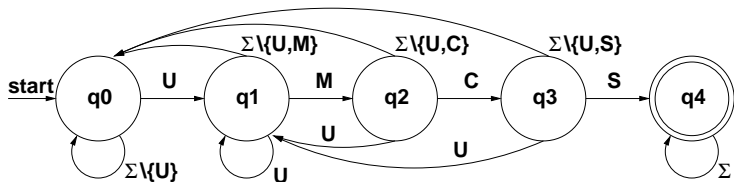
Mamy więc trzy równoważne zapisy definiujące DAS

- zapis „piątkowy”
- diagram
- tabela

**Przykład:** Automat DAS  $A_{umcs}$  rozpoznający w tekście ciąg UMCS. Alfabetem będzie alfabet polski + znaki interpunkcyjne

$$\Sigma = \{a, \dots, \acute{z}, A, \dots, \acute{Z}, 0, \dots, 9, \text{znaki specjalne}\}$$

$A_{umcs}$  najwygodniej zdefiniować diagramem:



Gdyby DAS miał rozpoznawać wyraz „UMCS” to należałoby zacząć szukać dopasowania od znaku spacji, a zakończyć na znaku spacji lub znaku przestankowym.

Ten sam DAS określony za pomocą tabeli

$A_{umcs}$	U	M	C	S	$\Sigma \setminus \{U, M, C, S\}$
$\rightarrow q_0$	$q_1$	$q_0$	$q_0$	$q_0$	$q_0$
$q_1$	$q_1$	$q_2$	$q_0$	$q_0$	$q_0$
$q_2$	$q_1$	$q_0$	$q_3$	$q_0$	$q_0$
$q_3$	$q_1$	$q_0$	$q_0$	$q_4$	$q_0$
$*q_4$	$q_4$	$q_4$	$q_4$	$q_4$	$q_4$

Definicje stanów:

- $q_0$  nie przeczytałem niczego interesującego
- $q_1$  przeczytałem U
- $q_2$  przeczytałem UM
- $q_3$  przeczytałem UMC
- $q_4$  przeczytałem UMCS

## Umowa

Reguły rysowania diagramów (grafów) definiujących DAS:

- 1 każdemu stanowi z  $Q$  odpowiada jeden wierzchołek; wierzchołki są podpisane oznaczeniem stanu i rysujemy je w postaci kółka
- 2 na stan wejściowy wskazuje strzałka z napisem START
- 3 stany akceptujące oznaczane są podwójnym kółkiem
- 4 jeśli istnieje funkcja przejścia  $\delta(q_i, s) = q_j$ , to stan  $q_i$  łączymy strzałką ze stanem  $q_j$ , a strzałkę podpisujemy symbolem  $s$  (ew. listą symboli, jeśli więcej niż jeden symbol powoduje takie przejście)
- 5 dla każdego stanu muszą istnieć funkcje  $\delta$  od każdego symbolu używanego alfabetu
- 6 fragmenty diagramu, do których nie można się dostać ze stanu początkowego, są z niego usuwane

Niekiedy wygodnie jest określić funkcję przejścia od łańcucha, a nie pojedynczego symbolu.

### Definicja: rozszerzona funkcja przejścia DAS

**Rozszerzona funkcja przejścia**  $\hat{\delta}$  zdefiniowana jest rekurencyjnie tak, że

- jeśli  $m = lx$  jest łańcuchem składającym się z łańcucha  $l$  i symbolu  $x$  ( $l \in \Sigma^*$ ,  $x \in \Sigma$ ),
- oraz jeśli DAS będący w stanie  $q$  po przetworzeniu łańcucha  $l$  znajdzie się w stanie  $p$ ,

to

$$\hat{\delta}(q, \varepsilon) = q$$

$$\hat{\delta}(q, m) = \delta(p, x) = \delta(\hat{\delta}(q, l), x)$$

Przykładowo dla  $A_{umcs}$ :

$$\hat{\delta}(q_0, \text{UMCS}) = q_4,$$

$$\hat{\delta}(q_i, m) = q_0 \text{ dla } i = 0, 1, 2, 3, m \neq \text{UMCS},$$

$$\hat{\delta}(q_4, m) = q_4 \text{ dla } m \text{ dowolnego.}$$

UWAGA:

Rozszerzona funkcja przejścia od łańcucha jednoelementowego redukuje się do normalnej funkcji przejścia

$$\hat{\delta}(q, x) = \delta(q, x)$$

Pamiętając o tym, można przyjąć to samo oznaczenie na obie funkcje. Dla klarowności nie będziemy tego tutaj robić.

Rozszerzone funkcje przejścia mogą posłużyć do zdefiniowania języka DAS.

### Definicja: język DAS

**Językiem DAS**  $\mathcal{L}(A)$  nazywamy zbiór łańcuchów przeprowadzających stan początkowy automatu  $A$  w jeden z jego stanów akceptujących:

$$\mathcal{L}(A) = \{w : \hat{\delta}(q_0, w) \in F\}$$

Kontynuując poprzedni przykład,  $\mathcal{L}(A_{umcs}) = \{\text{UMCS}\}$ .

### Definicja: język regularny

Jeśli dla danego języka  $\mathcal{L}$  istnieje taki DAS  $A$ , że  $\mathcal{L} = \mathcal{L}(A)$ , to język  $\mathcal{L}$  nazywany jest **językiem regularnym**. Innymi słowy, język jest regularny jeśli jest językiem jakiegoś DAS.



## Definicja: niedeterministyczny automat skończony

**Niedeterministyczny automat skończony (NAS)** to automat skończony, niedeterministyczny, zdefiniowany poprzez  $(Q, \Sigma, \delta, q_0, F)$ , gdzie

- $Q$  to skończony **zbiór stanów**
- $\Sigma$  to skończony zbiór **symboli wejściowych** (alfabet)
- $\delta$  to „**funkcje**” **przejścia** między stanami,  $\delta : Q \times \Sigma \rightarrow Q', \quad Q' \subset Q$
- $q_0 \in Q$  to **stan początkowy**
- $F \subset Q$  to zbiór **stanów akceptujących** (stanów końcowych)

Jedyna różnica w definicjach DAS i NAS to postać funkcji przejścia  $\delta$ :

$$\text{(DAS)} \quad \delta : Q \times \Sigma \rightarrow Q$$

$$\text{(NAS)} \quad \delta : Q \times \Sigma \rightarrow Q', \quad Q' \subset Q$$

W przypadku NAS odwzorowanie  $\delta$  mapuje stan i symbol na zbiór stanów, nie jest więc funkcją, chociaż takiej nazwy się używa.

Jak działają odwzorowania  $\delta$  w przypadku NAS?

- automat z pojedynczego stanu przechodzi do jednego lub kilku stanów
- oznacza to, że może znajdować się w kilku stanach jednocześnie
- analizuje symultanicznie kilka możliwych ścieżek wędrówki po grafie; niektóre prowadzą do stanów akceptowalnych, inne się kończą na stanach nieakceptowalnych

Przykład: NAS rozstrzygający, czy liczba jest parzysta. Dla ułatwienia, zapiszemy liczbę w postaci binarnej. Wtedy zadanie NAS sprowadza się do sprawdzenia, czy ciąg binarny **kończy się symbolem 0**.

Diagram:

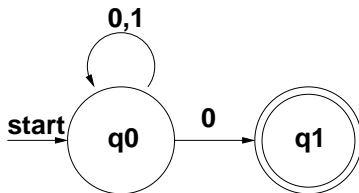


Tabela:

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
$*q_1$	$\emptyset$	$\emptyset$

Uwagi:

- z  $q_0$  wychodzą dwie linie znakowane 0 i jedna 1
- NAS kończy działanie wraz z końcem analizowanego łańcucha

Pytanie: czy dałoby się skonstruować DAS wykonujący to samo zadanie?

## Definicja: rozszerzona funkcja przejścia NAS

**Rozszerzona funkcja przejścia**  $\hat{\delta}$  zdefiniowana jest rekurencyjnie tak, że

- jeśli  $m = lx$  jest łańcuchem składającym się z łańcucha  $l$  i symbolu  $x$  ( $l \in \Sigma^*$ ,  $x \in \Sigma$ ),
- oraz jeśli NAS będący w stanie  $q$  po przetworzeniu łańcucha  $l$  znajdzie się w stanach  $\{p_1, \dots, p_k\}$ ,

to

$$\hat{\delta}(q, \varepsilon) = q$$

$$\hat{\delta}(q, m) = \bigcup_{i=1}^k \delta(p_i, x)$$

## Definicja: język NAS

**Językiem NAS**  $\mathcal{L}(A)$  nazywamy zbiór łańcuchów  $w$  przeprowadzających stan początkowy automatu  $A$  w zbiór stanów, zawierający co najmniej jeden stan akceptujący:

$$\mathcal{L}(A) = \{w : \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

- w tym wypadku, w zbiorze wszystkich stanów pośrednich, jakie generuje łańcuch  $w$ , musi się znaleźć taki podzbiór, który utworzy w grafie ścieżkę łączącą stan początkowy  $q_0$  z którymś ze stanów akceptujących
- fakt istnienia w zbiorze końcowym wielu stanów  $\notin F$  nie gra roli
- fakt istnienia wielu ścieżek, które nie kończą się stanem akceptującym, również nie ma znaczenia

## Twierdzenie (nieformalne) o równoważności NAS i DAS

Każdy język, który da się opisać za pomocą pewnego NAS, można opisać również za pomocą pewnego DAS. Automaty te są sobie równoważne.

Uwagi:

- istnieją problemy, które łatwiej opisać DAS, a inne za pomocą NAS, ale jest to jedynie kwestia wygody i ew. elegancji zapisu
- równoważność działa oczywiście w obie strony  $DAS \leftrightarrow NAS$
- z grubsza każdemu zbiorowi stanów NAS przypisuje się nowy stan DAS i na tej podstawie konstruuje funkcje przejścia  $\delta$
- jeśli NAS ma  $n$  stanów, to konstrukcja formalnie przewiduje  $2^n$  stanów dla DAS
- w praktyce wiele z tych stanów jest nieosiągalnych (tworzą część grafu rozłączną ze stanem początkowym) i powinno się je usunąć, dlatego często liczba stanów NAS i DAS jest porównywalna

Przykład: NAS rozpoznający liczby parzyste był zdefiniowany tabelką

NAS	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
$*q_1$	$\emptyset$	$\emptyset$

Można z niej wyodrębnić 4 różne stany:  $\emptyset$ ,  $q_0$ ,  $q_1$ ,  $\{q_0, q_1\}$ . Pełna wersja tej tabelki przybiera postać (uwaga na 3. linię!):

NAS	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
$*q_1$	$\emptyset$	$\emptyset$
$*\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\emptyset$	$\emptyset$	$\emptyset$

Widać, że zaczynając ze stanu  $q_0$  nie osiągniemy stanu  $q_1$  w postaci „czystej”, a jedynie pośrednio poprzez stan  $\{q_0, q_1\}$ . Podobnie stan  $\emptyset$  staje się niepotrzebny. Można więc usunąć linie drugą i czwartą z tabeli.

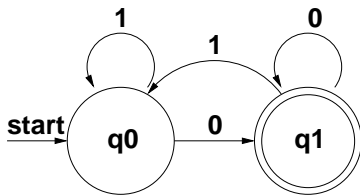
Po uporządkowaniu otrzymujemy przekształconą tabelę w postaci odpowiedniej dla DAS

DAS	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0\}$

Lepiej to widać po zmianie nazw

DAS	0	1
$\rightarrow q_0$	$q_1$	$q_0$
$*q_1$	$q_1$	$q_0$

i narysowaniu diagramu



Powyższa analiza jest przykładem tzw. **konstrukcji podzbiorów**.



## Konstrukcja podzbiorów

- jest ogólną metodą przekształcenia  $NAS \rightarrow DAS$
- polega na utworzeniu ze zbioru stanów  $NAS$  nowego stanu  $DAS$
- na tej podstawie konstruuje się funkcje przejścia  $DAS$
- operacja jest powtarzana do momentu, aż wszystkie stany zostaną opisane, a funkcja przejścia będzie przeprowadzała automat z dowolnego stanu w dokładnie jeden stan wyjściowy

## Twierdzenie o równoważności NAS i DAS

Mając dany automat NAS  $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$  i konstruowany na jego podstawie za pomocą konstrukcji podzbiorów automat DAS  $D = (Q_D, \Sigma, \delta_D, q_0, F_D)$ , zachodzi równość języków  $\mathcal{L}(D) = \mathcal{L}(N)$ .

Aby udowodnić to twierdzenie należy pokazać, że z konstrukcji podzbiorów wynika równość

$$\hat{\delta}_D(q_0, m) = \hat{\delta}_N(q_0, m)$$

dla dowolnego słowa  $m$ .

Szkic dowodu:

Niech  $m = lx$ , zaś stany  $p_i$  są osiągalne przez  $N$  ze stanu  $q_0$  po przetworzeniu łańcucha  $l$ . Konstrukcja podzbiorów daje przepis:

$$\delta_D(\{p_1, \dots, p_k\}, x) = \bigcup_{i=1}^k \delta_N(p_i, x).$$

Automat DAS traktuje  $\{p_1, \dots, p_k\}$  jako pojedynczy stan, więc zgodnie z definicją rozszerzonej funkcji przejścia dla DAS

$$\delta_D(\{p_1, \dots, p_k\}, x) = \hat{\delta}_D(q_0, m).$$

Analogiczna definicja dla NAS mówi, że

$$\bigcup_{i=1}^k \delta_N(p_i, x) = \hat{\delta}_N(q_0, m).$$

Ostatecznie więc otrzymujemy żadaną równość

$$\hat{\delta}_D(q_0, m) = \hat{\delta}_N(q_0, m).$$

- Automat  $N$  akceptuje  $m$  jeśli zbiór  $\hat{\delta}_N(q_0, m)$  zawiera stan z  $F_N$ .
- Automat  $D$  akceptuje  $m$  jeśli stan  $\hat{\delta}_D(q_0, m)$  zawiera stan z  $F_D$ .
- Na podstawie udowodnionej równości mamy  $F_N = F_D$ .

Wniosek: języki opisujące oba automaty są identyczne,  $\mathcal{L}(D) = \mathcal{L}(N)$ .

UWAGA:

Pełny dowód  $\rightarrow$  patrz AUT, rozdz. 2.3.5.

Jak na razie twierdzenie mówi tylko, że z każdego NAS można zbudować (metodą konstrukcji podzbiorów) taki DAS, że języki akceptowane przez oba automaty są identyczne.

### Obserwacja: DAS jest szczególną postacią NAS

Każdy DAS można interpretować jako szczególny NAS o funkcjach przejścia definiujących zawsze zbiór jednoelementowy:

$$\delta_D(q, x) = p \quad \Rightarrow \quad \delta_N(q, x) = \{p\}$$

i przez analogię

$$\hat{\delta}_D(q_0, m) = p \quad \Rightarrow \quad \hat{\delta}_N(q_0, m) = \{p\}.$$

Powyższa konstrukcja nie zmienia automatu, a co za tym idzie, jego języka i polega jedynie na reinterpretacji elementów automatu.

**Automaty NAS i DAS są równoważne.**

## Twierdzenie o regularności języków

Język  $\mathcal{L}$  jest akceptowany przez pewien DAS wtedy i tylko wtedy, gdy jest akceptowany przez pewien NAS.

Dowód: wynika wprost z twierdzenia o równoważności DAS i NAS:

- język jest regularny jeśli istnieje akceptujący go DAS
- DAS i NAS są sobie równoważne
- $\Rightarrow$  język jest regularny jeśli istnieje akceptujący go NAS
- $\mathcal{L}$  jest akceptowany przez DAS  $\Leftrightarrow$   $\mathcal{L}$  jest akceptowany przez NAS

Problem: automat ma rozpoznawać liczby całkowite. Przykładowe postaci to: 1, 12, -1, -12. Tak więc długość ciągu cyfr jest nieznana, zaś znak minus jest opcjonalny (może wystąpić lub nie).

Alfabet (minimalny):

$$\Sigma = \{-, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Wygodnie byłoby dopuścić przejścia od łańcucha pustego, a więc  $\delta(q, \varepsilon)$ , aby uwzględnić ewentualny znak minus!

- rozszerzenie automatu o przejścia z łańcuchem pustym oznacza wprowadzenie  $\varepsilon$  jako dozwolonego argumentu funkcji przejścia
- oznacza to, że automat może spontanicznie zmienić stan, podążając ścieżką diagramu podpisaną  $\varepsilon$
- nie oznacza to, że włączamy  $\varepsilon$  do alfabetu  $\Sigma$  (patrz uwaga pod definicją  $\varepsilon$ -NAS)

Mając do dyspozycji  $\varepsilon$ -przejścia, automat możemy skonstruować następująco:

Diagram:

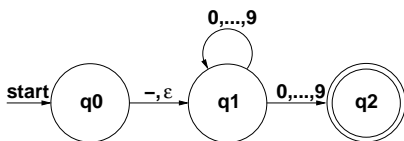


Tabela:

$\varepsilon$ -NAS	$\varepsilon$	-	$0, \dots, 9$
$\rightarrow q_0$	$\{q_1\}$	$\{q_1\}$	$\emptyset$
$q_1$	$\emptyset$	$\emptyset$	$\{q_1, q_2\}$
$*q_2$	$\emptyset$	$\emptyset$	$\emptyset$

Jest to przykład nondeterministycznego automatu skończonego z  $\varepsilon$ -przejściami:

- nondeterministyczność pozwala nie znać długości ciągów wejściowych
- $\varepsilon$ -przejścia pozwalają uwzględnić opcjonalny znak minus



## Definicja: NAS z $\varepsilon$ -przejściami

**Niedeterministyczny automat skończony z  $\varepsilon$ -przejściami ( $\varepsilon$ -NAS)** to automat skończony, niedeterministyczny, zdefiniowany poprzez  $(Q, \Sigma, \delta, q_0, F)$ , gdzie

- $Q$  to skończony **zbiór stanów**
- $\Sigma$  to skończony zbiór **symboli wejściowych** (alfabet)
- $\delta$  to „**funkcje**” **przejścia** między stanami,  
 $\delta : Q \times \Sigma \cup \{\varepsilon\} \rightarrow \{Q'\} \subset Q$
- $q_0 \in Q$  to **stan początkowy**
- $F \subset Q$  to zbiór **stanów akceptujących** (stanów końcowych)

UWAGA:  $\Sigma$  jest zbiorem symboli mogących pojawić się na wejściu automatu. Co prawda  $\varepsilon$  formalnie należy do każdego alfabetu, ale w przypadku  $\varepsilon$ -NAS wyklucza się ten symbol z  $\Sigma$ . Stąd zapis  $\Sigma \cup \{\varepsilon\}$  w definicji funkcji przejścia.

## Definicja: $\varepsilon$ -domknięcie stanu

$\varepsilon$ -**domknięciem stanu**  $q$  nazywamy zbiór stanów osiągalnych z  $q$  poprzez  $\varepsilon$ -przejścia. Definicja rekurencyjna:

$$q \in \text{edomk}(q)$$

$$[p \in \text{edomk}(q)] \wedge [\exists r : \delta(p, \varepsilon) = r] \Rightarrow [r \in \text{edomk}(q)]$$

Analogicznie do poprzednich przypadków definiuje się dla  $\varepsilon$ -NAS rozszerzone funkcje przejścia i język.

## Definicja: rozszerzona funkcja przejścia $\varepsilon$ -NAS

**Rozszerzona funkcja przejścia**  $\hat{\delta}$  zdefiniowana jest rekurencyjnie tak, że

- jeśli  $m = lx$  jest łańcuchem składającym się z łańcucha  $l$  i symbolu  $x$  ( $l \in \Sigma^*$ ,  $x \in \Sigma$ ),
- oraz jeśli  $\varepsilon$ -NAS będący w stanie  $q$  po przetworzeniu łańcucha  $l$  znajdzie się w stanach  $\{p_1, \dots, p_k\}$ ,
- oraz jeśli symbol  $x$  zamienia stany  $\{p_1, \dots, p_k\} \rightarrow \{r_1, \dots, r_s\}$ , czyli  $\bigcup_{i=1}^k \delta(p_i, x) = \{r_1, \dots, r_s\}$ ,

to

$$\hat{\delta}(q, \varepsilon) = \text{edomk}(q)$$

$$\hat{\delta}(q, m) = \bigcup_{j=1}^s \text{edomk}(r_j)$$

## Definicja: język $\varepsilon$ -NAS

**Językiem  $\varepsilon$ -NAS**  $\mathcal{L}(A)$  nazywamy zbiór łańcuchów  $w$  przeprowadzających stan początkowy automatu  $A$  w zbiór stanów, zawierający co najmniej jeden stan akceptujący:

$$\mathcal{L}(A) = \{w : \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

Formalnie definicja ta ma taką samą postać jak definicja dla NAS i analogiczną do definicji dla DAS. Różnica polega na uwzględnieniu w funkcjach  $\hat{\delta}$   $\varepsilon$ -przejsć.

## Twierdzenie o równoważności DAS i $\varepsilon$ -NAS

Język  $\mathcal{L}$  jest akceptowany przez pewien  $\varepsilon$ -NAS wtedy i tylko wtedy, gdy jest regularny (czyli jest akceptowany przez pewien DAS).

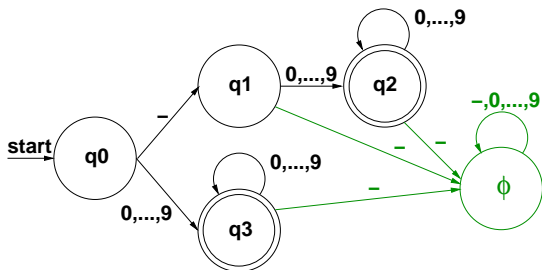
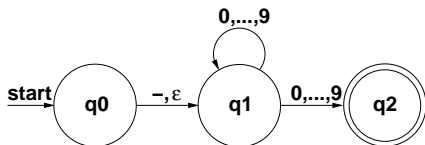
Aby to pokazać wystarczy udowodnić, że każdy  $\varepsilon$ -NAS można sprowadzić do NAS. Ponieważ NAS i DAS są równoważne, otrzymamy powyższe twierdzenie.

Oznacza ono, że dla każdego  $\varepsilon$ -NAS możemy podać równoważny mu DAS. Procedura wymaga:

- pozbycia się przejść  $\varepsilon$  (otrzymujemy NAS)
- przekształcenia NAS na DAS wg schematu konstrukcji podzbiorów

Formalna procedura eliminacji  $\varepsilon$ -przejść  $\rightarrow$  patrz AUT rozdz. 2.5.5.

## Przykład równoważnych automatów $\varepsilon$ -NAS i DAS



- przejście z  $\varepsilon$  powoduje rozgałęzienie grafu DAS
- mamy dwa stany akceptujące DAS, odpowiadające liczbom dodatnim i ujemnym
- mamy stan pusty  $\emptyset$  (zielona część), którego się zazwyczaj nie rysuje; reprezentuje on niewłaściwy ciąg i nie jest stanem akceptowalnym, chociaż na nim DAS może skończyć działanie

## Podsumowanie

- $\varepsilon$ -NAS pozwala na najbardziej kompaktowe zakodowanie skomplikowanych operacji
- każde  $\varepsilon$ -przejście w  $\varepsilon$ -NAS oznacza rozgałęzienie ścieżek w NAS (i duplikację części stanów powtarzających się w rozłącznych ścieżkach)
- znalezienie DAS równoważnego danemu NAS jest trudniejsze (procedura konstrukcji podzbiorów)
- tak skonstruowane DAS mogą wymagać wprowadzenia stanu pustego
- części diagramu nieosiągalne ze stanu początkowego formalnie istnieją, ale często się je pomija, przez co (pozornie!) diagram DAS może wyglądać na niekompletny
- nieosiągalność ze stanu początkowego może wynikać z zewnętrznych warunków, np. z narzuconych postaci łańcuchów wejściowych

Pokazaliśmy więc, że wszystkie trzy typy automatów: DAS, NAS i  $\varepsilon$ -NAS mogą być w sobie wzajemnie przekształcane.

## Wyrażenia regularne

### Wyrażenia regularne (RE, regexp)

- język opisu wzorców czy wzorce opisujące język?
- są wykorzystywane w zadaniach dopasowania i rozpoznawania łańcuchów, m.in. w wyszukiwarkach, parserach, analizatorach treści...
- technicznie, wyrażenie regularne jest przez aplikację zamieniane na odpowiedni automat
- prowadzi to do (słusznego) podejrzenia, że automaty i RE nawet jeśli nie są sobie równoważne, to mają wiele wspólnego
- konkretne symbole używane do budowanie RE różnią się pomiędzy aplikacjami i systemami, ale podstawowe zasady są niezmiennie



Na językach można wykonać trzy podstawowe operacje. Są to:

- **suma**, którą definiuje się jako sumę zbiorów łańcuchów należących do obu języków

$$\forall a [a \in \mathcal{L} \vee a \in \mathcal{M}] \Leftrightarrow [a \in \mathcal{L} \cup \mathcal{M}]$$

- **konkatenacja**, którą definiuje się jako zbiór łańcuchów utworzonych jako konkatenacje łańcuchów należących do obu języków

$$\forall a, b [a \in \mathcal{L} \wedge b \in \mathcal{M}] \Leftrightarrow [ab \in \mathcal{LM}]$$

- **domknięcie**, które jest definiowane jako suma łańcuchów należących do kolejnych konkatenacji języka z samym sobą (suma potęg języka)

$$\mathcal{L}^* = \bigcup_{i=0}^{\infty} \mathcal{L}^i = \mathcal{L}^0 \cup \mathcal{L}^1 \cup \mathcal{L}^2 \cup \dots = \{\varepsilon\} \cup \mathcal{L} \cup \mathcal{LL} \cup \dots$$

## Definicja: algebra RE

**Algebrę wyrażeń regularnych** definiuje się następująco:

- $\varepsilon$  jest RE reprezentującym język  $\{\varepsilon\}$ , czyli  $\mathcal{L}(\varepsilon) = \{\varepsilon\}$
- $\emptyset$  jest RE reprezentującym język  $\emptyset$ , czyli  $\mathcal{L}(\emptyset) = \emptyset$
- jeśli  $R_1$  i  $R_2$  są RE, to:

(suma)	$\mathcal{L}(R_1 + R_2) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$
(konkatenacja)	$\mathcal{L}(R_1 R_2) = \mathcal{L}(R_1) \mathcal{L}(R_2)$
(domknięcie)	$\mathcal{L}(R_1^*) = (\mathcal{L}(R_1))^*$

- jeśli  $R$  jest RE, to  $(R)$  jest również RE reprezentującym ten sam język co  $R$ ,  $\mathcal{L}((R)) = \mathcal{L}(R)$

## UWAGI:

- pojedyncze symbole alfabetu są RE
- suma i konkatenacja RE (domknięcie (dopełnienie) jest kombinacją obu tych operacji) tworzą poprawne RE
- RE możemy grupować za pomocą nawiasów

## Kolejność wykonywania operacji na RE:

- domknięcie („*potęgowanie*”)
- konkatenacja („*mnożenie*”)
- suma

## Twierdzenie o odpowiedniości automatu i RE

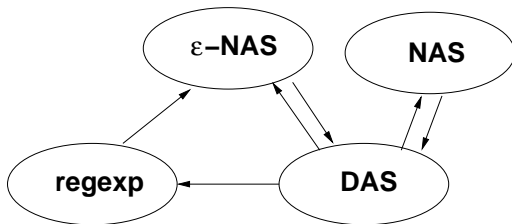
Jeśli język  $\mathcal{L}$  jest akceptowany przez pewien automat, to istnieje wyrażenie regularne, które opisuje ten język.

## Twierdzenie o odpowiedniości RE i automatu

Jeśli język  $\mathcal{L}$  jest opisywany przez wyrażenie regularne, to istnieje automat  $\epsilon$ -NAS, który akceptuje ten język.

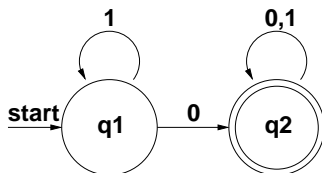
W połączeniu z poprzednimi twierdzeniami, otrzymujemy równoważność wszystkich czterech struktur:

- DAS
- NAS
- $\epsilon$ -NAS
- regexp



## Przykład przejścia DAS → RE.

Rozważmy automat DAS akceptujący ciągi bitowe zawierające co najmniej jeden symbol 0



Ogólny ciąg akceptowany przez DAS ma

- dowolną liczbę 1 na początku
- 0
- dowolną liczbę 0 lub 1 na końcu

co zapisać można jako

$$1^*0(0 + 1)^*$$

Formalna procedura dla ogólnego przypadku  $\varepsilon$ -NAS  $\rightarrow$  RE.

- ponumerujemy stany automatu kolejnymi liczbami,  $1, 2, \dots$
- oznaczmy przez  $R_{ij}^{(n)}$  RE, które umożliwia przejście między stanami  $i \rightarrow j$  wykonując skoki przez stany pośrednie o numerach co najwyżej równych  $n$
- każde  $R_{ij}^{(n)}$  może być  $\emptyset$  lub zbiorem niepustym, z uwzględnieniem ewentualnych  $\varepsilon$ -przejęć
- można pokazać ( $\rightarrow$  AUT [3.2]), że

$$R_{ij}^{(n)} = R_{ij}^{(n-1)} + R_{in}^{(n-1)} \left( R_{nn}^{(n-1)} \right)^* R_{nj}^{(n-1)}$$

czyli przejście  $i \rightarrow j$  przez  $n$  można zrealizować bez odwiedzania węzła  $n$  lub odwiedzając węzeł pośredni  $n$ , z możliwością zostania w  $n$  dowolnie długo (symbol  $*$ )

- na koniec sumuje się RE prowadzące od stanu początkowego do stanu akceptującego automatu

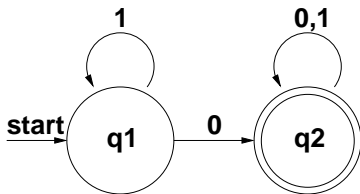
Wracając do przykładu, przejścia bez węzła pośredniego można wyczytać bezpośrednio z grafu:

$$R_{11}^{(0)} = \varepsilon + 1 = 1$$

$$R_{12}^{(0)} = 0$$

$$R_{21}^{(0)} = \emptyset$$

$$R_{22}^{(0)} = \varepsilon + 0 + 1 = 0 + 1$$



Z ogólnej reguły  $R_{ij}^{(1)} = R_{ij}^{(0)} + R_{i1}^{(0)} \left( R_{11}^{(0)} \right)^* R_{1j}^{(0)}$  generujemy drugi rząd przybliżenia, czyli przejścia z jednym węzłem pośrednim  $q_1$

$$R_{11}^{(1)} = \varepsilon + 1 + (\varepsilon + 1)(\varepsilon + 1)^*(\varepsilon + 1) = 1^*$$

$$R_{12}^{(1)} = 0 + (\varepsilon + 1)(\varepsilon + 1)^*0 = 1^*0$$

$$R_{21}^{(1)} = \emptyset + \emptyset(\varepsilon + 1)^*(\varepsilon + 1) = \emptyset$$

$$R_{22}^{(1)} = \varepsilon + 0 + 1 + \emptyset(\varepsilon + 1)^*0 = 0 + 1$$

Analogicznie reguła  $R_{ij}^{(2)} = R_{ij}^{(1)} + R_{i2}^{(1)} \left( R_{22}^{(1)} \right)^* R_{2j}^{(1)}$  daje drugi rząd przybliżenia z węzłem pośrednim  $q_2$

$$R_{11}^{(2)} = 1^* + 1^*0(0+1)^*\emptyset = 1^*$$

$$R_{12}^{(2)} = 1^*0 + 1^*0(0+1)^*(0+1) = 1^*0(0+1)^*$$

$$R_{21}^{(2)} = \emptyset + (0+1)(0+1)^*\emptyset = \emptyset$$

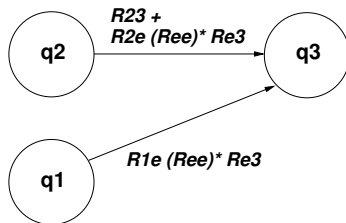
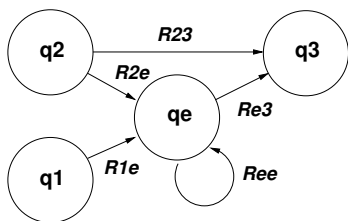
$$R_{22}^{(2)} = 0 + 1 + (0+1)(0+1)^*(0+1) = (0+1)^*$$

Przejście od stanu początkowego ( $q_1$ ) do akceptującego ( $q_2$ ) opisywane jest wyrażeniem  $R_{12}^{(2)} = 1^*0(0+1)^*$ , co kończy formalnie konstrukcję.



## Przekształcenie DAS $\rightarrow$ RE metodą eliminacji stanów

- zamieniamy opisy etykiet z symboli alfabetu na RE
- przekształcamy graf automatu usuwając z niego po kolei stany pośrednie
- rekompensujemy to, dodając etykiety z RE odpowiadające przejściom przez eliminowany stan pośredni



- po dojściu do automatu jednostanowego (ze stanem akceptującym) otrzymane wyrażenie jest zapamiętywane
- redukcję prowadzi się osobno dla każdego stanu akceptującego
- na koniec dodajemy wyrażenia RE odpowiadające stanom akceptującym

Metoda eliminacji stanów sprowadza się do:

- znalezienia RE odpowiadających ścieżkom w grafie automatu (dowolnego typu), prowadzących od stanu początkowego do stanu akceptującego,
- zsumowaniu tych RE dla wszystkich stanów akceptujących.

Jest to więc **konstrukcja języka automatu** (patrz definicje języków i funkcji  $\hat{\delta}$ ) i symboliczny jego zapis za pomocą umownych wyrażeń (język automatu jest regularny, więc te wyrażenia też nazywają się regularne).

## Przekształcanie RE $\rightarrow \epsilon$ -NAS

Generalna strategia:

- rozkładamy RE na najprostsze podwyrażenia
- symbole RE odpowiadają etykietom przejść między stanami
- konkatencja RE to ciąg stanów połączonych funkcją przejścia
- suma RE to rozgałęzienie grafu
- gwiazdka odpowiada pętli zwrotnej

Szczegółowe instrukcje  $\rightarrow$  AUT[3.2.3]

Dowód twierdzenia równoważności RE i  $\varepsilon$ -NAS.

**Cel:** Wykazać, że dla dowolnego wyrażenia regularnego istnieje odpowiadający mu automat (w sensie równości języków).

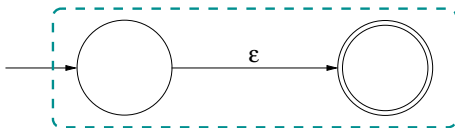
**Dowód:** Zauważmy, że każde wyrażenie regularne jest postaci:

- prymitywnej – symbolu  $\varepsilon$ , zbioru pustego  $\emptyset$  lub symbolu  $x \in \Sigma$
- złożonej – kombinacji powyższych w formie sumy  $R_1 + R_2$ , konkatenacji  $R_1 R_2$  lub gwiazdki (domknięcia)  $R^*$
- wyrażenia regularne mogą być również grupowane za pomocą nawiasów,  $(R)$

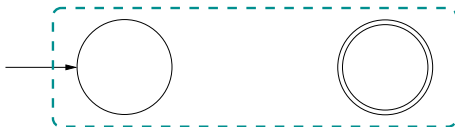
Grupowanie nawiasami nie zmienia języka, tj.  $\mathcal{L}(R) = \mathcal{L}((R))$ , więc nie musimy się tym osobno zajmować.

## Automaty odpowiadające prymitywnym postaciom RE

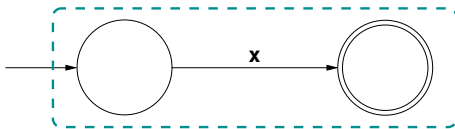
- symbol  $\varepsilon$



- zbiór pusty  $\emptyset$

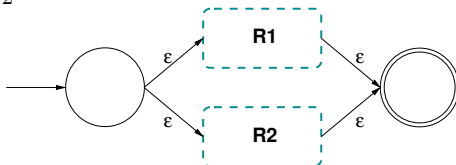


- symbol  $x$



## Automaty odpowiadające złożonym postaciom RE

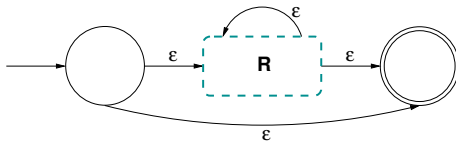
- suma  $R_1 + R_2$



- złożenie  $R_1 R_2$



- domknięcie  $R^*$

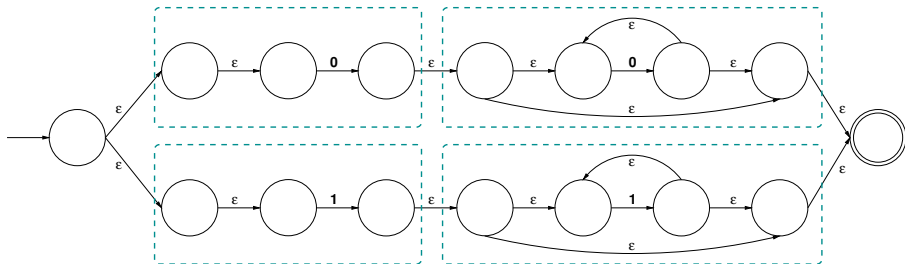


Automat odpowiadający dowolnemu RE buduje się

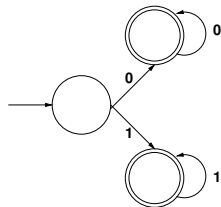
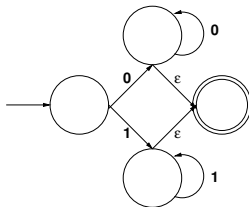
- dokonując dekompozycji RE tak długo, aż otrzymamy zestaw operacji wykonywanych na prymitywnych formach
- podstawiając pod każdy element składowy odpowiedni automat
- łącząc wyjścia i wejścia automatów zgodnie z kolejnością występowania odpowiadających im fragmentów w RE
- redukcji „nadmiarowych” fragmentów grafu:
  - ▶ większość  $\varepsilon$ -przejęć
  - ▶ bezpośrednich ścieżek do stanu  $\emptyset$ , a więc takich, które odpowiadają słowom nie należącym do języka automatu

**Przykład:** Konstrukcja automatu odpowiadającego wyrażeniu

$$0^+ + 1^+ = 00^* + 11^*.$$



Po zredukowaniu niczego  
nie wnoszących  $\varepsilon$ -przejść  
otrzymujemy wersję  
z jednym ( $\varepsilon$ -NAS) lub  
dwoma (DAS) stanami  
akceptującymi.





Korzystając z regexp w praktyce, nie stosuje się zapisu algebraicznego

- symbol  $a+b$  oznacza „a lub b” i jest zastępowany zapisem z pionową kreską  $a|b$  albo zapisem w nawiasach kwadratowych  $[ab]$
- rezygnuje się z  $\varepsilon$
- rezygnuje się z  $\emptyset$

Przykład:

$$(0 + 1)^* \rightarrow [01]^* \Leftrightarrow (0|1)^*$$

Składnia RE w różnych systemach i aplikacjach jest różna. Istnieje pewien standard, który jest rozszerzany i poprawiany przez programistów na użytek konkretnego programu.

## Podstawowa składnia regexp w linuxie

.	pojedynczy znak
r*	zero lub więcej wystąpień r
r+	jedno lub więcej wystąpień r
r?	zero lub jedno wystąpienie r
r\{n\}	dokładnie n powtórzeń r
r\{n,m\}	co najmniej n ale nie więcej niż m wystąpień r
[abc]	a lub b lub c
[^abc]	wszystko tylko nie a, b i c
[a-c]	zakres (sprawdzany po kodach znaków)
^	początek linii
\$	koniec linii
r1\ r2	r1 lub r2
\( ... \)	grupowanie RE

Dodatkowo definiuje się klasy, wykorzystywane przez aplikacje takie jak grep i sed.

<code>[:alnum:]</code>	znaki alfanumeryczne (alfabet+liczby)
<code>[:alpha:]</code>	znaki alfabetu
<code>[:blank:]</code>	znaki puste: SPC, TAB
<code>[:digit:]</code>	cyfry
<code>[:lower:]</code>	[a-z]
<code>[:punct:]</code>	znaki specjalne i interpunkcyjne
<code>[:space:]</code>	TAB, NL, VT (vertical tab), FF (form feed), CR, SPC
<code>[:upper:]</code>	[A-Z]

Zaletą posługiwania się klasą jest niewrażliwość RE na kolejność znaków w używanym kodowaniu, zależnym często od ustawionej lokalizacji.

**Przykład** regexp w konwencji uniksowej/linuksowej: adres email.

Warunki:

- email zajmuje całą linię
- format to `login@host.domena.kod`
- login może zawierać małe litery, cyfry, kropki, podkreślenie i myślnik (minus)
- `host.domena` podobnie jak login
- kod to kropka i dokładnie 2, 3 lub 4 litery (`.pl`, `.com`, `.info`)

Jedno z rozwiązań:

```
/^([a-z0-9_\. -]+)@([a-z0-9_\. -]+)\.([a-z]{2,4})$/
```

Znajdź wady tego zapisu!

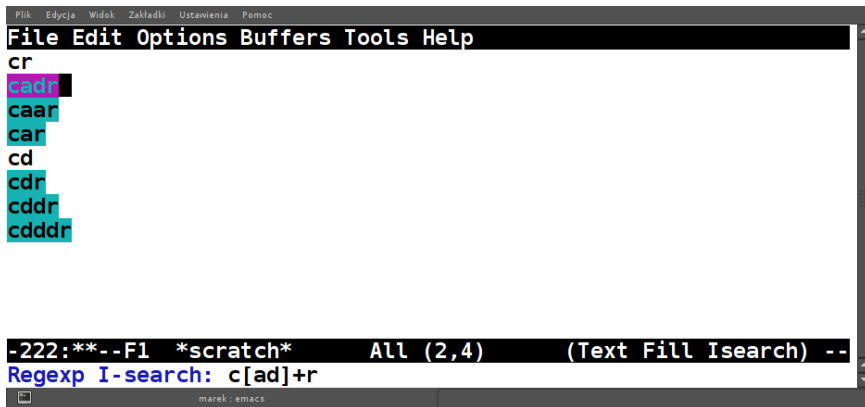
**Przykład** regexp w konwencji emacs, podobny do tego, jaki wykorzystywany jest do wykrycia przez edytor końca zdania (wg systemu anglosaskiego).

Warunki:

- zdanie kończy się odpowiednim znakiem przestankowym
- zdanie lub jego końcówka mogą być ujęte w nawiasy lub cudzysłowy – wtedy zamykający nawias/cudzysłów stawia się po znaku przestankowym

```
[.?!] [[]\''')}]*
```

**Przykład** bezpośrednio z edytora. Wyszukiwanie w tekście słów pasujących do RE (skrót C-M-s, komenda M-x `isearch-forward-regexp`)



```
File Edit Options Buffers Tools Help
cr
cadr
caar
car
cd
cdr
cddr
cdddr

-222:**--F1 *scratch* All (2,4) (Text Fill Isearch) --
Regexp I-search: c[ad]+r

marek: emacs
```

## Przykłady za [www.emacswiki.com](http://www.emacswiki.com)

Specjalne kombinacje z emacsa w tych przykładach:

- `\w` oznacza pojedyncze słowo
- `C-q C-j` to kombinacja oznaczająca przejście do nowej linii
- `\< \>` to znaki ograniczające wyrazy

<code>[-+[:digit:]]</code>	cyfra, + lub -
<code>\(\\+\\ -\\)?[0-9]+\\(\\. [0-9]+\\)?</code>	liczba z kropką dziesiętną
<code>\\(\\w+\\) +\\1\\&gt;</code>	powtarzające się wyrazy
<code>\\&lt;[[:upper:]]\\w*</code>	wyraz pisany wielką literą
<code>+\$</code>	spacje na końcu linii
<code>\\w\\{20,\\}</code>	wyraz, 20 liter lub więcej
<code>\\w+ski\\&gt;</code>	wyraz kończący się na ski
<code>\\(19\\ 20\\)[0-9]\\{2\\}</code>	lata 1900-2099
<code>^\\.\\{6,\\}</code>	6+ znaków na początku linii
<code>^[a-zA-Z0-9_]\\{3,16\\}\$</code>	przykładowy login
<code>&lt;tag[^&gt; C-q C-j ]*&gt;\\(.*?\\)&lt;/tag&gt;</code>	znacznik HTML o nazwie tag

## Prawa algebraiczne dla regexp.

Niech  $\mathcal{L}$ ,  $\mathcal{M}$  i  $\mathcal{N}$  będą językami regularnymi. Obowiązują wtedy dla nich następujące prawa:

- **przemienność sumy**  $\mathcal{L} + \mathcal{M} = \mathcal{M} + \mathcal{L}$
- **łączność sumy**  $(\mathcal{L} + \mathcal{M}) + \mathcal{N} = \mathcal{L} + (\mathcal{M} + \mathcal{N})$
- **łączność konkatenacji**  $(\mathcal{L}\mathcal{M})\mathcal{N} = \mathcal{L}(\mathcal{M}\mathcal{N})$
- **element neutralny sumy**  $\emptyset + \mathcal{L} = \mathcal{L} + \emptyset = \mathcal{L}$
- **element neutralny konkatenacji**  $\varepsilon\mathcal{L} = \mathcal{L}\varepsilon = \mathcal{L}$
- **anihilator konkatenacji**  $\emptyset\mathcal{L} = \mathcal{L}\emptyset = \emptyset$
- **prawostronna rozdzielność konkatenacji względem sumy**  
 $\mathcal{L}(\mathcal{M} + \mathcal{N}) = \mathcal{L}\mathcal{M} + \mathcal{L}\mathcal{N}$
- **lewostronna rozdzielność konkatenacji względem sumy**  
 $(\mathcal{M} + \mathcal{N})\mathcal{L} = \mathcal{M}\mathcal{L} + \mathcal{N}\mathcal{L}$



Ponieważ w regexp nie występują jawnie symbole  $\varepsilon$  i  $\emptyset$ , które są potrzebne do zbudowania niektórych grafów opisujących automaty, prawa dotyczące

- elementów neutralnych
- anihilatorów

pozwalają na upraszczanie wyrażeń przy przejściach  $\varepsilon$ -NAS  $\rightarrow$  regexp.

Dodatkowo trzeba podkreślić, że:

- konkatenacja nie jest przemienna
- suma nie posiada anihilatora

Dalsze prawa i właściwości:

- **idempotentność sumy**  $\mathcal{L} + \mathcal{L} = \mathcal{L}$
- **idempotentność domknięcia**  $(\mathcal{L}^*)^* = \mathcal{L}^*$
- **domknięcie zbioru pustego**  $\emptyset^* = \{\varepsilon\}$ , bo

$$\mathcal{L}(\emptyset^*) = \mathcal{L}(\emptyset)^* = \{\varepsilon\} \cup \mathcal{L}(\emptyset) \cup \mathcal{L}(\emptyset)\mathcal{L}(\emptyset) \cup \dots = \{\varepsilon\} \cup \emptyset \cup \emptyset \dots = \{\varepsilon\}$$

- **domknięcie łańcucha pustego**  $\varepsilon^* = \varepsilon$
- **przemienność z własnym domknięciem**  $\mathcal{L}^+ = \mathcal{L}\mathcal{L}^* = \mathcal{L}^*\mathcal{L}$

## Gramatyki

Do tej pory mówiliśmy o *językach* (regularnych), czyli zbiorach łańcuchów symboli konstruowanych według pewnej reguły. *Języki* tożsame są automatom i wyrażeniom regularnym.

*Językiem* może być dla pewnego języka (sic!) programowania zbiór etykiet znakujących zmienne, znakujących stałe, zbiór słów kluczowych, zbiór symboli oznaczających operatory itd.

Potrzebna jest więc szersza struktura, która łączyłaby w sobie wiele *języków*, aby opisać np. interpreter/kompilator języka programowania.

## Definicja: gramatyka bezkontekstowa

Gramatyką bezkontekstową  $G$  nazywamy czwórkę  $G = (V, T, P, S)$ , gdzie

- $V$  to zbiór **zmiennych** – każda zmienna jest tożsama z jednym językiem; jedna ze zmiennych jest wyróżniona odpowiadając symbolowi początkowemu  $S$ , reszta to pomocnicze klasy łańcuchów
- $T$  to zbiór **symboli końcowych** – jest to niepusty zbiór skończony, zawierający łańcuchy definiowanego języka
- $P$  to zbiór **produkcji** – skończony zbiór reguł pozwalających na rekurencyjne definiowanie języka
- $S$  to **symbol początkowy**

Można to w zwarty sposób zapisać następująco:

$$\begin{array}{ll} T \neq \emptyset & S \in V \\ T \cap V = \emptyset & P \subset V \times (T \cup V)^* \end{array}$$

## Słowniczek pojęć:

- zmienna = symbol nieterminalny
- symbol końcowy = symbol terminalny, terminal
- łańcuch języka = leksem, token

Każda **produkcja** jest parą  $(A, \zeta)$ , gdzie  $A \in V$  jest pojedynczą zmienną (symbolem nieterminalnym),  $\zeta \in (T \cup V)^*$  jest dowolnym symbolem terminalnym lub nieterminalnym, również łańcuchem pustym. Zapisuje się je w postaci:

$$A \rightarrow \zeta$$

Innymi słowy produkcja jest regułą typu:

$$\text{GŁOWA} \rightarrow \text{CIAŁO}$$

gdzie

- GŁOWA to zmienna definiowana przez daną produkcję
- CIAŁO to  $\varepsilon$  lub łańcuch utworzony ze zmiennych i terminali

Cała reguła określa sposób tworzenia łańcuchów w języku reprezentowanym przez zmienną będącą głową.

W praktyce produkcje tożsame są wyrażeniom regularnym, jednak tych ostatnich nie używa się jawnie na poziomie gramatyk.

Tak więc gramatyka jest konstrukcją szerszą<sup>\*)</sup> niż omawiane do tej pory:

- operuje na więcej niż jednym języku, a więc odpowiada jej cały zestaw automatów lub pojedynczy automat złożony
- RE spełniają rolę wzorców łańcuchów akceptowanych przez ten automat
- produkcje wykorzystują RE do określania reguł budowania akceptowanych łańcuchów

*\*) „Szerokość” gramatyki nie polega na tym, że zawiera w sobie informacje, których nie ma w DAS i RE, ale informacje te podane są w sposób jawny poprzez produkcje, które można utworzyć rozkładając RE na czynniki pierwsze.*

**Przykład:** Gramatyka opisująca liczby parzyste w zapisie binarnym. Alfabetem binarnym jest  $\Sigma = \{0, 1\}$ . Każda liczba parzysta w zapisie binarnym kończy się symbolem 0. W zapisie regexp będzie to:

$$(0 + 1)^*0$$

Zbiorem zmiennych  $V$  jest więc zbiór zawierający element reprezentujący liczbę parzystą w zapisie binarnym  $\{lparz\}$ . Ponieważ mamy tylko jedną zmienną, pełni ona automatycznie rolę symbolu początkowego  $S = lparz$ . Zbiorem terminali  $T$  jest alfabet binarny  $\{0, 1\}$ . Zbiór produkcji  $P$  nazwiemy  $Prod$  i zawiera on:

$$lparz \rightarrow 0 \ lparz$$

$$lparz \rightarrow 1 \ lparz$$

$$lparz \rightarrow 0$$

Ostatecznie gramatykę zapisujemy jako:

$$G = (V, T, P, S) = (\{lparz\}, \{0, 1\}, Prod, lparz).$$



## Definicja: łańcuch bezpośrednio wyprowadzalny

Mamy dane gramatykę  $G = (V, T, P, S)$  oraz dwa łańcuchy  $\zeta, \eta \in (T \cup V)^*$  (być może puste). Łańcuch  $\eta$  jest **bezpośrednio wyprowadzalny** z łańcucha  $\zeta$ :

$$\zeta \xRightarrow{G} \eta$$

wtedy i tylko wtedy, gdy istnieją

- łańcuchy  $\alpha, \beta, \gamma \in (T \cup V)^*$
- produkcja  $A \rightarrow \gamma$

takie, że

$$\zeta = \alpha A \beta$$

$$\eta = \alpha \gamma \beta$$

## Definicja: łańcuch pośrednio wyprowadzalny

Łańcuch  $\eta$  jest **pośrednio wyprowadzalny** z  $\zeta$ , jeśli istnieje ciąg łańcuchów  $\alpha_i$ ,  $i = 1, \dots, n$ , takich, że

$$\alpha_1 = \zeta$$

$$\alpha_i \xrightarrow{G} \alpha_{i+1}$$

$$\alpha_n = \eta$$

Innymi słowy:

- łańcuch bezpośrednio wyprowadzalny jest możliwy do osiągnięcia poprzez zastosowanie pojedynczej produkcji
- łańcuch pośrednio wyprowadzalny jest możliwy do osiągnięcia po zastosowaniu szeregu produkcji

Do definiowania gramatyk używa się najczęściej **wyprowadzeń**, czyli produkcji w kierunku  $GŁOWA \rightarrow CIAŁO$ . W ten sposób z symboli/łańcuchów należących do języka danej zmiennej budujemy nowe, które siłą rzeczy również do tego języka należą.

Można też używać **wnioskowania rekurencyjnego**, tj. odwrócić kierunek produkcji  $GŁOWA \leftarrow CIAŁO$ . Bierze się wtedy wynikowy łańcuch i dopasowuje do niego istniejące produkcje, rozkładając go na podłańcuchy. Jeśli w ten sposób da się otrzymać najprostsze symbole, analizowany łańcuch musi należeć do języka głowy produkcji.

W analogii do uogólnionych funkcji przejścia  $\hat{\delta}$ , które przyjmowały jako argument całe łańcuchy, a nie pojedyncze symbole, definiuje się **uogólnione wyprowadzenie**, oznaczające zero lub więcej kroków pośrednich wyprowadzenia:

$$\alpha \xRightarrow[G]{*} \beta$$

Zbiorczo oznacza to, że:

- przy zero krokach pośrednich  $\beta$  będzie łańcuchem bezpośrednio wyprowadzalnym z  $\alpha$
- przy jednym lub więcej krokach pośrednich  $\beta$  będzie łańcuchem pośrednio wyprowadzalnym z  $\alpha$

**Przykład:** Niech gramatyka  $G = (V, T, P, S)$  opisuje wyrażenia arytmetyczne, zbudowane z trzech zmiennych (sic!)  $x, y, z$  i czterech działań  $+, -, *, /$  oraz nawiasów  $(, )$ .

Zbiór zmiennych (symboli nieterminalnych)

Możemy wyodrębnić dwa typy obiektów: zmienną i operator. Tak więc zbiór zmiennych składa się z

$$V = \{\text{zmienna}, \text{operator}, S\}$$

gdzie  $S$  to symbol początkowy.

Zbiór symboli terminalnych

Terminalami są wszystkie używane w zapisie wyrażen arytmetycznych symbole, a więc

$$T = \{x, y, z, +, -, *, /, (, )\}$$

## Produkcje

Reguły tworzenia nowych łańcuchów i decydowania, czy dany łańcuch jest poprawnym wyrażeniem arytmetycznym. Ogólny schemat wygląda tak:

$$S \rightarrow \text{zmienna}$$

$$S \rightarrow S \text{ operator } S$$

$$S \rightarrow (S)$$

$$\text{zmienna} \rightarrow x$$

$$\text{zmienna} \rightarrow y$$

$$\text{zmienna} \rightarrow z$$

$$\text{operator} \rightarrow +$$

$$\text{operator} \rightarrow -$$

$$\text{operator} \rightarrow *$$

$$\text{operator} \rightarrow /$$

Problem: sprawdzić, czy  $x + (y/x)$  jest poprawnym wyrażeniem arytmetycznym w sensie omawianej gramatyki. Wyrażenie to będzie poprawne, jeśli da się podać ciąg wyprowadzeń, prowadzący od symbolu początkowego, do tego wyrażenia:  $S \xRightarrow[G]{*} x + (y/x)$

Przykładowy ciąg:

$$\begin{aligned}
 S &\xRightarrow[G]{} S \text{ operator } S \xRightarrow[G]{} S \text{ operator } (S) \xRightarrow[G]{} S \text{ operator } (S \text{ operator } S) \\
 &\xRightarrow[G]{} S \text{ operator } (S/S) \xRightarrow[G]{} S + (S/S) \xRightarrow[G]{} \text{zmienna} + (S/S) \\
 &\xRightarrow[G]{} \text{zmienna} + (\text{zmienna} / S) \\
 &\xRightarrow[G]{} \text{zmienna} + (\text{zmienna} / \text{zmienna}) \\
 &\xRightarrow[G]{} x + (\text{zmienna} / \text{zmienna}) \\
 &\xRightarrow[G]{} x + (y / \text{zmienna}) \\
 &\xRightarrow[G]{} x + (y/x)
 \end{aligned}$$

Podane wyprowadzenie jest „bałaganiarskie” w kwestii kolejności zamiany wyrażeń i podstawień terminali.

**Wyprowadzenie lewostronne** to takie, w którym budujemy wyrażenie od lewej do prawej. Pierwsza zmienna od lewej zastępowana jest jednym z ciał jej produkcji.

**Wyprowadzenie prawostronne** to takie, w którym budujemy wyrażenie od prawej do lewej. Pierwsza zmienna od prawej zastępowana jest jednym z ciał jej produkcji.



Poprzednie wyprowadzenie w wersji czysto lewostronnej:

$$S \xRightarrow{G,l} S \text{ operator } S$$

$$\xRightarrow{G,l} \text{ zmienna operator } S$$

$$\xRightarrow{G,l} x \text{ operator } S$$

$$\xRightarrow{G,l} x + S$$

$$\xRightarrow{G,l} x + (S)$$

$$\xRightarrow{G,l} x + (S \text{ operator } S)$$

$$\xRightarrow{G,l} x + (\text{zmienna operator } S)$$

$$\xRightarrow{G,l} x + (y \text{ operator } S)$$

$$\xRightarrow{G,l} x + (y/S)$$

$$\xRightarrow{G,l} x + (y/\text{zmienna}) \qquad \xRightarrow{G,l} x + (y/x)$$

## Definicja: język gramatyki

**Językiem gramatyki**  $G = (V, T, P, S)$  nazywamy zbiór łańcuchów symboli końcowych, dla których istnieje wyprowadzenie z symbolu początkowego

$$\mathcal{L}(G) = \{w \in T^* : S \xRightarrow[G]{*} w\}$$

Jeśli  $G$  jest gramatyką bezkontekstową, to  $\mathcal{L}(G)$  jest **językiem bezkontekstowym**.

## Definicja: forma zdaniowa

Wyprowadzenie z symbolu początkowego nazywane jest **formą zdaniową**. Wyróżnia się lewo- i prawostronne formy zdaniowe. Tak więc  $\mathcal{L}(G)$  jest zbiorem form zdaniowych, składających się wyłącznie z symboli końcowych.

Do reprezentacji wyprowadzeń używa się grafu zwanego **drzewem wyprowadzenia**. Jeśli  $G = (V, T, P, S)$  jest gramatyką, to drzewa wyprowadzeń dla  $G$  podlegają regułom

- wierzchołki wewnętrzne opisane są zmiennymi z  $V$
- liście opisane są zmiennymi, terminalami lub łańcuchem pustym  $\varepsilon$
- jeśli liść opisany jest przez  $\varepsilon$  to musi to być jedyny liść dla tego rodzica
- jeśli wierzchołek opisany zmienną  $A \in V$  ma liście opisane symbolami  $X_1, X_2, \dots, X_n$ , to musi istnieć produkcja  $A \rightarrow X_1 X_2 \dots X_n$  ( $X_i \neq \varepsilon$ )

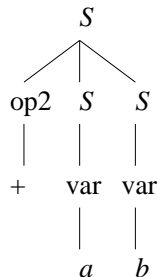
**Przykład:** część gramatyki opisującej prefiksowe działanie operatora dwuargumentowego.

Przyjmijmy, że mamy daną gramatykę z symbolami końcowymi op2 (operator binarny, prefiksowy), var (zmienna), +, a, b i być może innymi. Istnieją też odpowiednie produkcje. Sprawdzamy, czy wyrażenie + a b jest poprawne w kontekście tej gramatyki.

Wyprowadzenie  
lewostronne

$$\begin{aligned}
 S &\Rightarrow_{G,l} \text{op2 } S \ S \\
 &\Rightarrow_{G,l} + \ S \ S \\
 &\Rightarrow_{G,l} + \ \text{var } S \\
 &\Rightarrow_{G,l} + \ a \ S \\
 &\Rightarrow_{G,l} + \ a \ \text{var} \\
 &\Rightarrow_{G,l} + \ a \ b
 \end{aligned}$$

Odpowiadające mu  
drzewo wyprowadzenia



## Twierdzenie o równoważności

Niech  $G = (V, T, P, S)$  będzie gramatyką. Poniższe procedury rozstrzygające, czy łańcuch końcowy  $w$  należy do języka zmiennej  $A$  są sobie równoważne:

- wnioskowanie rekurencyjne
- uogólnione wyprowadzenie  $A \xRightarrow{*}_G w$
- uogólnione wyprowadzenie lewostronne  $A \xRightarrow{*}_{G,l} w$
- uogólnione wyprowadzenie prawostronne  $A \xRightarrow{*}_{G,p} w$
- konstrukcja drzewa wyprowadzenia o korzeniu  $A$  i łańcuchu wynikowym  $w$

Ścisłe dowody  $\rightarrow$  AUT[5.2]

Równoważność drzewa i wyprowadzenia lewostronnego – szkic dowodu.

Przypuśćmy, że drzewo opisane jest korzeniem  $A \in V$  i liśćmi

$w = w_1 w_2 \dots w_k$ ,  $w_i \in T$ .

- ❶ Jeśli wysokość drzewa  $n = 1$ , to musi istnieć produkcja  $A \rightarrow w$ , która jest jednocześnie wyprowadzeniem lewostronnym  $A \xRightarrow[G,l]{*} w$ .
- ❷ Jeśli wysokość drzewa to  $n + 1$  i po  $n$  krokach mamy

$$A \xRightarrow[G,l]{*} X_1 X_2 \dots X_k$$

to

- ▶  $X_i$  jest albo elementem z  $T$ , wtedy  $w_i = X_i$
  - ▶  $X_i$  jest głową produkcji  $X_i \rightarrow w_i$
- ❸ W obu przypadkach, postępując w kierunku rosnących  $i$  otrzymamy lewostronne wyprowadzenie łańcucha  $w$ .

Istnienie wyprowadzenia prawostronnego pokazuje się w analogiczny sposób. Istnienie wyprowadzenia nieuporządkowanego wynika bezpośrednio z istnienia wyprowadzeń lewo- i prawostronnych.

Równoważność wyprowadzeń i wnioskowania – szkic dowodu.

- ➊ Wnioskowanie rekurencyjne polega na sprawdzeniu, że analizowany symbol jest ciałem produkcji, bezpośrednio lub pośrednio wyprowadzalnym.
- ➋ Jeśli więc istnieje wyprowadzenie, które dowodzi, że dany łańcuch należy do języka gramatyki, to wystarczy odwrócić operacje, aby z wyprowadzeń otrzymać wnioskowanie.

Formalnie dowód przeprowadza się rekurencyjnie względem liczby kroków wyprowadzenia.

## Zastosowanie gramatyk w programowaniu

Języki programowania działają się na dwa typy:

- interpretowane (np. AWK, Ruby, Bash, Python, Perl)
- kompilowane (np. C, Fortran, Java, Haskell)

*Istnieją jeszcze języki znacznikowe (np. html, xml), ale nie są to najczęściej języki programowania w takim sensie, w jakim tego określenia używam na wykładzie.*

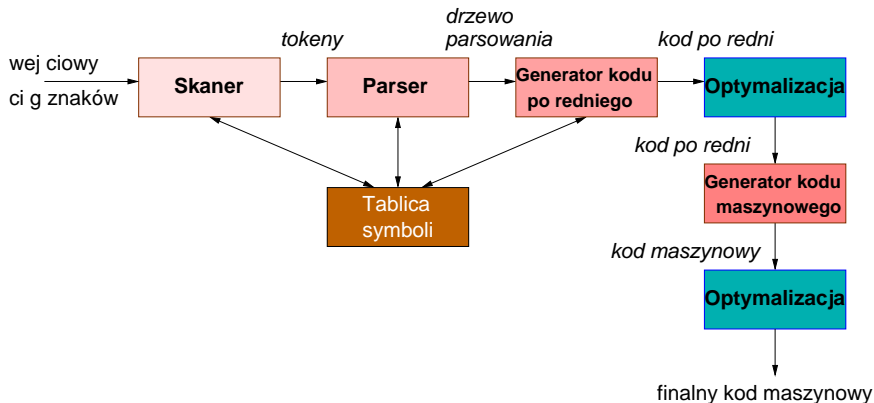
Proces prowadzący od odczytu kodu do jego wykonania zawiera kilka etapów. W fazie wstępnej mamy:

- analizę leksykalną
- analizę syntaktyczną
- analizę semantyczną

Później albo działa kompilator generujący kod wykonywalny, albo interpreter uruchamiający kod w locie.

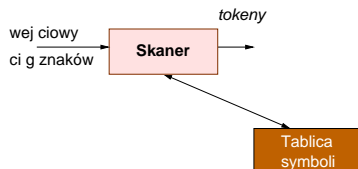


# Typowy schemat kompilacji



## Analiza leksykalna

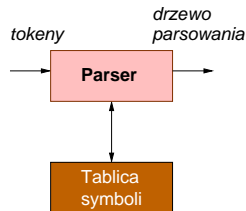
- jest wykonywana przez **skaner**
- jest wykonywana na podstawie reguł, które do skanera dostarczone są z zewnątrz (czyli ktoś musi taki skaner najpierw oprogramować)
- polega na analizie kodu i klasyfikacji poszczególnych łańcuchów znaków za pomocą znaczników („tokenizacja kodu”); **tokeny** to zmienne późniejszej gramatyki
- skaner tworzy plik z rozbiorem leksykalnym analizowanego kodu, który staje się jednym z plików wejściowych dla parsera
- w linuxie używany jest `flex` (*fast lexical analyser generator*), kompatybilny z leciwym już programem `lex`



Skaner (analizator leksykalny) zazwyczaj posługuje się językiem regularnym, w którym alfabetem są znaki używane do zapisania kodu.

## Analiza syntaktyczna

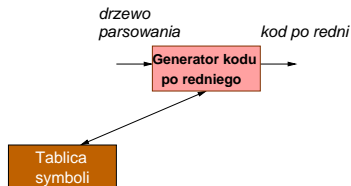
- jest wykonywana przez **parser** (generator parserów)
- opiera się na regułach parsera oraz pliku z danymi pochodzącymi od skanera
- reguły parsera to produkcje gramatyki zdefiniowane na tych samych tokenach, które były używane w regułach skanera
- parser tworzy pełną gramatykę (w postaci drzewa wyprowadzeń, tu nazywanym drzewem parsowania)
- w linuxie parser (funkcję parsującą) generuje bison, następca programu yacc (*parser generator*)



Generator parserów działa w oparciu o język bezkontekstowy, zdefiniowany na tokenach wygenerowanych wcześniej przez skaner.

## Analiza semantyczna

- analiza znaczeniowa, czyli przypisanie do abstrakcyjnych tworów opisywanych tokenami (zmiennymi gramatyki) takimi jak *wyrażenie*, *etykieta*, *zmienna*, *operator* itd. konkretnych wartości
- analizowany jest kod pośredni generowany przez skaner i parser
- zajmuje się tym część kompilatora
- sprawdzane są zgodności typów zmiennych
- wyszukiwane są takie same etykiety w programie i twory im przypisane są utożsamiane
- analizowane są pętle i rekurencje
- generowane komunikaty o błędach
- tworzony jest nowy kod pośredni



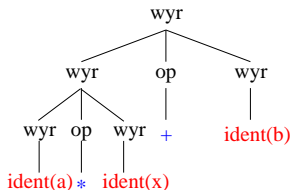
**Przykład:** Analiza wyrażenia  $a * x + b$  na podstawie gramatyki z produkcjami (wprowadzamy zapis skrótowy)

$$\begin{aligned} \text{wyr} &\rightarrow \text{ident} \mid \text{liczba} \mid - \text{wyr} \mid ( \text{wyr} ) \mid \text{wyr op wyr} \\ \text{op} &\rightarrow + \mid - \mid * \mid / \end{aligned}$$

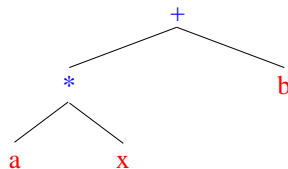
### Analiza leksykalna

$a * x + b$   
 $\downarrow$   
 $a \ b \ x$     ident  
 $* \ +$     op

### Analiza syntaktyczna



### Analiza semantyczna



**Przykład:** Problemy w analizie leksykalnej.

Zazwyczaj skaner ignoruje białe znaki w kodzie, tj. nie ma znaczenia, jak programista graficznie zapisał kod. W niektórych językach wcięcia są istotnym elementem składni (Fortran77 (fixed-format), Haskell), w innych nie, ale zapisy

```
1  a = 2.0
```

```
2  a=2.0
```

są najczęściej nierozróżnialne. Porównaj dwa kody:

```
1  DO 5 I = 1.25
```

```
1  DO 5 I = 1,25
```

F77 ignoruje spacje, więc w pierwszym przykładzie skaner powinien rozpoznać podstawienie pod zmienną D05I wartości 1.25, zaś w drugim przykładzie początek pętli, zamkniętej etykietą 5 (tej etykiety to już będzie poszukiwał parser, aby zdecydować, czy jest to prawidłowa konstrukcja języka), o zmiennej I przyjmującej kolejne wartości 1,2,...,25.

## Problem:

- skaner musi jakoś zakwalifikować napotkaną konstrukcję
- nie wie, co znajduje się dalej (kropka czy przecinek?)
- musi czytać kod z pewnym nadmiarem naprzód i swoje decyzje uzależniać od tego, co następuje
- kod może być błędny!

Wniosek: skaner z buforem odczytu (pamięcią) → automaty ze stosem

**Dalsze losy kodu:** kod pośredni trafia do

- interpretera i jest wykonywany
- kompilatora i jest tłumaczony na kod maszynowy

Kompilator dokonuje szeregu dalszych operacji, m.in. **optymalizacji kodu**. W fazie wstępnej następuje optymalizacja niezależna od architektury (np. rozwijane są zagnieżdżone pętle), potem optymalizacja pod konkretną architekturę (wykorzystanie rozszerzeń instrukcji CPU, rejestrów, wielowątkowość, współbieżność itd.). Pod koniec może mieć miejsce **linkowanie dynamiczne** lub **linkowanie statyczne**.

W przypadku rozrzucenia programu po wielu plikach schemat kompilacji musi jeszcze obejmować **scalanie kodu**. Jeśli program wykorzystuje podprogramy (procedury, funkcje, biblioteki zewnętrzne) musi nastąpić test spójności pełnego kodu.

Jak łatwo się domyśleć, bardzo wygodne dynamiczne typowanie, obecne w niektórych językach (np. ruby, awk), nie jest proste do zrealizowania na poziomie kompilatora!



**Przykład – lex:** analiza stdin, rozpoznawanie liczb i słów. Kod dla lex'a składa się w najprostszej wersji z dopasowania (regexp) i akcji

```
1  %{
2  #include <stdio.h>
3  %}
4
5  %%
6  [0123456789]+           printf("NUMBER\n");
7  [a-zA-Z][a-zA-Z0-9]*    printf("WORD\n");
8  %%
```

Po kompilacji tworzony jest kod w C. Część ograniczona %{ i %} jest bezpośrednio przenoszona do tego pliku. Reszta jest generowana z reguł postaci regexp – akcja.

## Przykład – kalkulator – lex: wyodrębnianie liczb w ciągach znakowych, ignorowanie znaków białych

```
1 %{
2 #include "y.tab.h"
3 extern int yylval;
4 %}
5
6 %%
7 [0-9]+ { yylval = atoi (yytext);
8         printf ("liczba %d\n", yylval);
9         return NUMBER; }
10 [ \t] { printf ("białe znaki\n"); }
11 \n { printf ("koniec linii\n");
12     return 0; }
13 . { printf ("inne dane \"%s\"\n", yytext);
14     return yytext[0]; }
15 %%
```

## Przykład – kalkulator – yacc: yacc może teraz rozpoznawać wyrażenia arytmetyczne i je obliczać

```
1 %{
2 #include <stdio.h>
3 int yylex();
4 int yyerror (char *msg);
5 %}
6 %token NAME NUMBER
7 %%
8 stmt:  NAME '=' expr    { printf("przypisanie %s
9                        do %d\n", $1, $3) }
10      |  expr            { printf("= %d\n", $1) }
11      ;
12 expr:  expr '+' NUMBER { $$ = $1 + $3 }
13      |  expr '-' NUMBER { $$ = $1 - $3 }
14      |  NUMBER         { $$ = $1 }
15 %%
16 int main (void) {
17     return yyparse();
18 }
19
20 int yyerror (char *msg) {
21     return fprintf (stderr, "YACC: %s\n", msg);
22 }
```

## Rodzaje błędów:

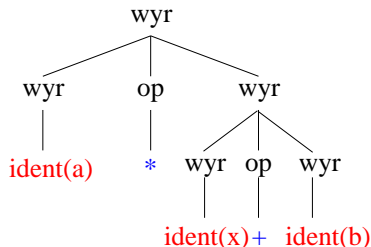
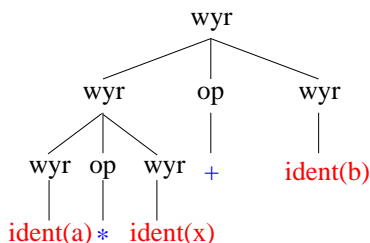
- **leksykalne** (słownikowe) – pomyłone identyfikatory zmiennych, słów kluczowych, brak cudzysłówów
- **syntaktyczne** (składniowe) – brak znaków specjalnych (przecinek, kropka, średnik), niedomknięte nawiasy, niedokończone struktury blokowe (`if...then...fi`, `begin...end`)
- **semantyczne** (znaczeniowe) – zazwyczaj jest to niezgodność typów, czyli użycie czegoś w kontekście, w którym to coś nie ma sensu (np. operacja na liście wykonywana na zmiennej napisowej)
- **logiczne** – błędy programisty, który koduje nie to, co zamierza, np. używa operatora porównania `==` zamiast przypisania `=`
- **algorytmiczne** – błędy programisty, który używa niepoprawnego algorytmu

Dla wielu gramatyk dane (poprawne) wyrażenie można wyprowadzić na kilka sposobów. Pomimo tego, że efekt końcowy wyprowadzenia jest ten sam, wyprowadzenia te mogą nie być sobie równoważne na poziomie semantycznym.

**Wieloznaczność gramatyki** polega na tym, że można podać kilka nierównoważnych sobie poprawnych wyprowadzeń danego łańcucha.

- przeprojektowanie gramatyki może rozwiązać problem wieloznaczności
- nie wszystkie gramatyki można poprawić – istnieją takie, w których niektóre łańcuchy zawsze będą określone wieloznacznie

Porównaj: oba poniższe wyprowadzenia prowadzą do wyrażenia  $a * x + b$ , ale lewe sugeruje poprawną kolejność działań  $(a * x) + b$ , natomiast prawe niepoprawną  $a * (x + b)$ . Poprawność i niepoprawność istnieje jedynie na poziomie semantycznym, gdyż nadajemy symbolom  $*$  i  $+$  znaczenie i właściwości mnożenia i dodawania.



W językach programowania problem wieloznaczności można rozwiązać (przynajmniej częściowo)

- deklarując lewostronną łączność operatorów – oznacza to, że czytając wyrażenie od lewej do prawej operator „przykleja” się do wyrażenia po jego lewej stronie
- w niektórych sytuacjach deklaruje się łączność prawostronną
- deklarując priorytety rozpatrywania operatorów – priorytety mogą np. łamać regułę ścisłego lewostronnego wyprowadzania poprawności wyrażień przez funkcję parsującą

Chcąc zadeklarować standardowe reguły arytmetyczne w yacc'u należy w kodzie przed definicją gramatyki podać deklaracje

```
1 %left '-' '+'
2 %left '*' '/'
3 %nonassoc UMINUS
```

gdzie UMINUS jest minusem przed liczbą ujemną.

## Automaty ze stosem

Aby móc analizować partie tekstu, a nie pojedyncze symbole, skaner/parser potrzebuje jakiegoś rodzaju pamięci. Odpowiednikiem takiego urządzenia w teorii automatów jest **automat ze stosem** (AZS). Jest to zwykły automat, który

- czyta i przetwarza symbole wejściowe
- ma do dyspozycji (nieograniczony) **stos**, na który może odkładać lub z niego zdejmować symbole
- przejścia między stanami uzależnione mogą być od obecnego stanu, analizowanego symbolu i szczytu stosu



## Stos

- obowiązuje kolejka FILO
- w każdym cyklu automat zastępuje łańcuch ze szczytu stosu  $x$  nowym  $y$ , co oznacza się  $x/y$
- jeśli  $y = \varepsilon$  to efektywnie zdejmowany jest szczytowy łańcuch ze stosu i cały stos podjeżdża w górę
- jeśli  $y = x$  to stos się nie zmienia
- jeśli  $y = zx$ , to efektywnie do stosu dodawany jest nowy łańcuch  $z$  na jego szczycie
- stos jest nieograniczony, dlatego automaty ze stosem są automatami skończonymi (skończona liczba stanów), mogącymi zapamiętać **nieskończoną** ilość informacji
- praktyczna realizacja automatu będzie oczywiście miała swoje ograniczenia (stos będzie ograniczony rozmiarem dostępnej pamięci fizycznej)

## Definicja: automat ze stosem

**Automat ze stosem (AZS)** to automat zdefiniowany poprzez  $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ , gdzie

- $Q$  to skończony **zbiór stanów**
- $\Sigma$  to skończony zbiór **symboli wejściowych** (alfabet)
- $\Gamma$  to skończony **alfabet stosowy**, czyli zbiór symboli, które można odkładać na stos
- $\delta$  to „**funkcje**” **przejścia** między stanami,  
 $\delta : (Q \times \Sigma \cup \{\epsilon\} \times \Gamma) \rightarrow \{Q \times \Gamma^*\}$
- $q_0 \in Q$  to **stan początkowy**
- $Z_0 \in \Gamma$  początkowy symbol na stosie
- $F \subset Q$  to zbiór **stanów akceptujących** (stanów końcowych)

## UWAGI:

### 1. Funkcja przejścia

$$\delta(q, a, s) \ni (p, s')$$

opisuje więc, że jeśli automat znajduje się w stanie  $q$ , analizuje symbol wejściowy  $a$ , zaś na szczycie stosu znajduje się symbol stosowy  $s$ , to automat ten może zmienić swój stan na  $p$ , zastępując  $s$  na stosie symbolem  $s'$ . Pełną przeciwdziedziną funkcji  $\delta(q, a, s)$  jest zbiór wszystkich możliwych par  $\{(p, s')\}$ .

### 2. Stan $p$ i symbol $s'$ stanowią nierozzerwalną parę. Jeśli

$$\delta(q, a, s) = \{(p_1, s'_1), (p_2, s'_2)\}$$

to niedozwolone jest np. przejście  $(q, a, s) \rightarrow (p_1, s'_2)$ .

3. Diagramy AZS rysuje się jak poprzednio, tylko konieczne jest uwzględnienie również stanu stosu przy symbolach przejścia. Mając funkcję  $\delta(q, a, s) \ni (p, s')$  i chcąc opisać linię pomiędzy stanami  $q$  i  $p$  należy nadać jej etykietę  $a, s/s'$ .

### Definicja: opis chwilowy AZS

Do pełnego opisu chwilowej konfiguracji AZS podaje się trójkę  $(q, w, s)$  gdzie

- $q$  to obecny stan automatu
- $w$  jest nieprzeanalizowaną jeszcze częścią wejścia
- $s$  zawartością stosu (w kierunku od lewej do prawej: wierzchołek  $\rightarrow$  dno stosu)

## Definicja: ewolucja AZS ( $\vdash$ )

Niech  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$  będzie AZS. Przypuśćmy, że  $\delta(q, a, s) \ni (p, s')$ . Wtedy dla wszystkich  $w \in \Sigma^*$  oraz  $\gamma \in \Gamma^*$  zachodzi

$$(q, aw, s\gamma) \vdash (p, w, s'\gamma)$$

Oznacza to, że to co pozostaje na wejściu ( $w$ ), jak również cały stos za wyjątkiem jego szczytu ( $\gamma$ ), nie mają wpływu na zmianę konfiguracji AZS.

W podobny sposób wprowadza się ciąg ewolucyjny  $\vdash^*$  opisujący zero lub więcej zmian konfiguracji AZS.

## Twierdzenie o ewolucji AZS

Niech  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$  będzie AZS. Jeśli

$$(q, a, s) \vdash^* (p, b, s')$$

to dla wszystkich  $w \in \Sigma^*$  oraz  $\gamma \in \Gamma^*$

$$(q, aw, s\gamma) \vdash^* (p, bw, s'\gamma)$$

## Twierdzenie odwrotne o ewolucji AZS

Niech  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$  będzie AZS. Jeśli

$$(q, aw, s\gamma) \vdash^* (p, bw, s'\gamma)$$

to również

$$(q, a, s) \vdash^* (p, b, s')$$

**Definicja: język AZS – akceptacja przez stan końcowy**

Niech  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$  będzie AZS. **Językiem akceptowanym przez  $P$**  nazywamy zbiór

$$\mathcal{L}(P) = \{w : (q_0, w, Z_0) \stackrel{*}{\vdash} (q, \varepsilon, \gamma), q \in F, \gamma \in \Gamma^*\}$$

Formalna definicja mówi, że  $\mathcal{L}(P)$  to zbiór takich łańcuchów wejściowych, które przeprowadzają AZS ze stanu początkowego do jakiegokolwiek stanu akceptującego, przy czym zawartość stosu po tej operacji nie ma znaczenia.

Dla AZS istnieje druga metoda akceptacji (po akceptacji poprzez stany końcowe), polegająca na **doprowadzeniu do opróżnienia stosu**.

**Definicja: akceptacja poprzez pusty stos (*null stack*)**

Dla każdego AZS  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$  definiujemy zbiór

$$N(P) = \{w : (q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon), q \in Q\}$$

Zgodnie z definicją,  $N(P)$  jest zbiorem łańcuchów wejściowych, które przeprowadzają AZS ze stanu początkowego w stan, w którym na stosie nie ma żadnych symboli (*null stack*).

$N(P)$  definiuje klasę języków  $\mathcal{L} = N(P)$ , akceptowanych w ten specyficzny sposób przez automat ze stosem.



**Przykład:** Chcemy zdefiniować automat, który będzie analizował kod programu sprawdzając, czy każdy nawias otwierający "(" ma swój nawias zamykający ")". Problemy:

- nie wiemy, jak długi tekst znajduje się w nawiasie
- nawiasy mogą się zagnieżdżać

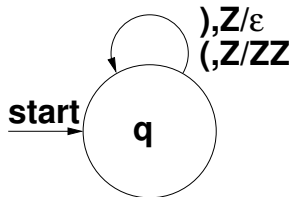
Kiedy automat na pewno natrafi na błąd? Gdy liczba nawiasów zamykających przekroczy chwilową liczbę napotkanych nawiasów otwierających. Widać więc, że automat musi mieć pamięć, w której będzie przechowywany licznik nawiasów. Właściwym typem automatu dla tego problemu jest automat wyposażony w stos.

Założmy, że liczba symboli na stosie będzie zwiększana za każdy napotkany nawias ( i zmniejszana o jeden za każdy nawias ). Wejście zawierające błąd będzie zerowało stos AZS. Czyli automat nie tyle będzie sprawdzał poprawność, co niepoprawność wejścia.

$$P_N = (\{q\}, \{ (, ) \}, \{Z\}, \delta_N, q, Z, \{q\})$$

gdzie po kolei podane są:

- pojedynczy stan  $q$
- alfabet wejściowy (nawiasy)
- pojedynczy symbol stosowy
- $\delta_N(q, (, Z) = \{(q, ZZ)\}$   
 $\delta_N(q, ), Z) = \{(q, \varepsilon)\}$
- stan początkowy
- początkowy stan stosu (pojedynczy symbol  $Z$ )
- stan akceptujący (nie gra roli)

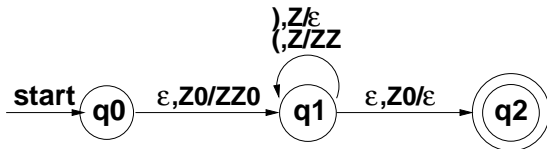


- etykieta  $), Z/\varepsilon$  oznacza napotkanie symbolu  $)$ , przy symbolu  $Z$  zdejmowanym ze szczytu stosu
- etykieta  $(, Z/ZZ$  oznacza napotkanie symbolu  $($ , przy symbolu  $Z$  dodawanym na szczyt stosu

## UWAGI:

- omawiany automat w praktyce będzie częścią większej struktury, która odsyła do niego tylko napotkane nawiasy
- stos pełni rolę licznika – każdy nawias zmienia liczbę odłożonych na stosie symboli
- symbol stosowy nie jest istotny, ważne żeby był
- struktura automatu nie zawiera informacji o sposobie akceptowania przez niego łańcuchów
- akceptowanie przez opróżnienie stosu jest umową (choć automat z pojedynczym stanem można o to podejrzewać...)

Automat akceptujący ten sam język poprzez stan końcowy, a nie wyzerowanie stosu, będzie miał następującą strukturę:



$$P_F = (\{q_0, q_1, q_2\}, \{(\cdot, \cdot)\}, \{Z, Z_0\}, \delta_F, q_0, Z_0, \{q_2\})$$

$$\delta_F(q_0, \varepsilon, Z_0) = \{(q_1, ZZ_0)\}$$

$$\delta_F(q_1, (, Z) = \{(q_1, ZZ)\}$$

$$\delta_F(q_1, ), Z) = \{(q_1, \varepsilon)\}$$

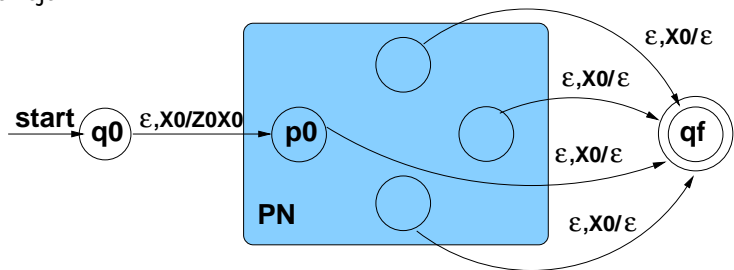
$$\delta_F(q_1, \varepsilon, Z_0) = \{(q_2, \varepsilon)\}$$

Można udowodnić, że dla każdego AZS akceptującego język poprzez stan końcowy istnieje AZS akceptujący ten sam język poprzez wyzerowanie stosu i na odwrót.

## Twierdzenie o odpowiedności $P_N$ i $P_F$

Jeśli AZS  $P_N$  akceptuje język  $\mathcal{L}$  przez pusty stos,  $\mathcal{L} = N(P_N)$ , to istnieje AZS  $P_F$  akceptujący ten sam język przez stan końcowy,  $\mathcal{L} = \mathcal{L}(P_F)$ .

Konstrukcja:

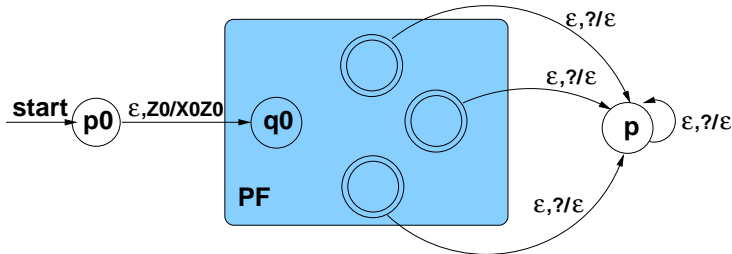


gdzie  $X_0$  jest symbolem początkowym stosu  $P_F$ , a  $Z_0$  symbolem początkowym stosu  $P_N$ .  $P_F$  przechodzi do swojego stanu akceptującego  $q_f$  gdy  $P_N$  opróżni stos (na szczycie stosu pojawi się symbol  $X_0$ ).

## Twierdzenie o odpowiedniości $P_F$ i $P_N$

Jeśli AZS  $P_F$  akceptuje język  $\mathcal{L}$  przez stan końcowy,  $\mathcal{L} = \mathcal{L}(P_F)$ , to istnieje AZS  $P_N$  akceptujący ten sam język przez pusty stos,  $\mathcal{L} = \mathcal{L}(P_N)$ .

Konstrukcja:



gdzie ? oznacza dowolny symbol stosowy.  $\epsilon$ -przejścia, jakie następują po wejściu  $P_F$  w jakikolwiek stan akceptujący powodują opróżnienie stosu bez pobierania kolejnego symbolu z wejścia.

## Twierdzenie o odpowiedniości GBK i $P_N$

Niech  $G = (V, T, P, S)$  będzie gramatyką bezkontekstową. Automat ze stosem akceptujący język tej gramatyki przez pusty stos,  $N(P_N) = \mathcal{L}(G)$ , ma postać:

$$P_N = (\{q\}, T, V \cup T, \delta, q, S, F)$$

gdzie dla każdej zmiennej  $A \in V$  i każdego symbolu końcowego  $a \in T$

$$\delta(q, \varepsilon, A) = \{(q, \beta) : A \rightarrow \beta \text{ jest produkcją } G\}$$

$$\delta(q, a, a) = \{(q, \varepsilon)\}$$

Funkcje innej postaci niż dwie wymienione wyżej są puste  $\delta(\dots) = \emptyset$ .

- AZS jest jednostanowy, tak więc operuje tylko zawartością stosu
- automat odgaduje produkcje i jeśli uda mu się doprowadzić łańcuch do jakiegoś symbolu terminalnego – opróżnia stos

**Przykład:** Gramatyka wyrażeń arytmetycznych dana jest produkcjami ( $v$  to zmienna,  $o$  to operator)

$$S \rightarrow v \mid S o S \mid (S)$$

$$v \rightarrow x \mid y \mid z$$

$$o \rightarrow + \mid - \mid * \mid /$$

Symbole terminalne  $T = \{x, y, z, +, -, *, /, (, )\}$  oraz symbole nieterminalne  $V = \{S, v, o\}$  tworzą łącznie alfabet stosowy  $\Gamma = V \cup T$ .  
Funkcje przejścia AZS o pojedynczym stanie  $q$  to:

$$\delta(q, \varepsilon, S) = \{(q, v), (q, S o S), (q, (S))\}$$

$$\delta(q, \varepsilon, v) = \{(q, x), (q, y), (q, z)\}$$

$$\delta(q, \varepsilon, o) = \{(q, +), (q, -), (q, *), (q, /)\}$$

$$\delta(q, t, t) = \{(q, \varepsilon)\} \quad (\text{dla wszystkich } t \in T)$$



## Twierdzenie o odpowiedniości AZS i GBK

Niech  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$  będzie automatem ze stosem. Wtedy istnieje gramatyka bezkontekstowa  $G$  taka, że  $\mathcal{L}(G) = N(P)$ .

Dowód  $\rightarrow$  AUT[6.3.2]

Na podstawie poznanych twierdzeń wnioskujemy, że są sobie równoważne:

- RE
- GBK
- AZS akceptujący przez stany końcowe
- AZS akceptujący przez pusty stos

## Kwestia determinizmu automatów ze stosem.

W ogólności AZS są niedeterministyczne, tj. w danym stanie, przy określonym wejściu i symbolu stosowym, mają określone więcej niż jedno możliwe przejście ( $\rightarrow$  przykłady w AUT rozdz. 6).

Jeśli odpowiednio ograniczymy nieoznaczoność wyborów kolejnych stanów chwilowych AZS możemy utworzyć deterministyczny automat ze stosem.

### Definicja: deterministyczny AZS

**Deterministyczny automat ze stosem (DAZS)** jest AZS, który spełnia warunki

- dla dowolnych argumentów ( $a$  może być  $\varepsilon$  lub dowolnym symbolem alfabetu  $\Sigma$ )  $\delta(q, a, X)$  jest zbiorem co najwyżej jednoelementowym
- jeśli dla pewnego  $a \in \Sigma$  mamy  $\delta(q, a, X) \neq \emptyset$ , to  $\delta(q, \varepsilon, X) = \emptyset$

UWAGA: dopuszcza się ewolucję AZS bez pobierania kolejnego symbolu z wejścia ( $\varepsilon$ -przejścia), ale dalej musi to być jedyne dozwolone przejście dla danych parametrów.

Podklasa DAZS ma przydatne właściwości:

- akceptuje wszystkie języki regularne...
- ...choć nie wszystkie języki akceptowane przez DAZS są regularne
- jeśli DAZS  $P$  akceptuje język  $\mathcal{L} = N(P)$ , to język ten ma jednoznaczną gramatykę bezkontekstową
- jeśli DAZS  $P$  akceptuje język  $\mathcal{L} = \mathcal{L}(P)$ , to język ten ma jednoznaczną gramatykę bezkontekstową

## Zmienne i typy danych

Jedną z podstawowych funkcji języka programowania jest możliwość definiowania i operowania na zmiennych oraz typach danych.

Zmienne charakteryzowane są przez 6 atrybutów. Są to:

- nazwa
- adres
- wartość
- typ
- okres życia
- zakres widoczności

## Nazwa zmiennej

- etykieta używana przez programistę, aby odwołać się do komórki pamięci przechowującej dane
- nie ma znaczenia dla programu po kompilacji
- różne konwencje w różnych j.programowania, zazwyczaj dozwolone są litery, cyfry i znak podkreślenia
  - ▶ Fortran77: zmienne o etykietach 6-znakowych pisanych wielkimi literami
  - ▶ Fortran77: pierwsza litera etykiety zmiennej określała jej domyślny typ (A-H, O-Z REAL, I-N INTEGER)
  - ▶ C: zaczął rozróżniać wielkie i małe litery
  - ▶ Ada (2005): dopuszcza wszystkie znaki unicode
  - ▶ Prolog: etykieta zmiennej zaczyna się wielką literą, stałej małą
- nie każda zmienna ma nazwę, np. w C++ deklarując przydział pamięci poleceniem `new` nazwę nadajemy wskaźnikowi, a nie zmiennej

## Adres

- adres komórki pamięci przechowującej *początek* wartości zmiennej
- długość danych zależy od typu zmiennej
- adres jest unikatowy, w odróżnieniu od nazw zmiennych, które mogą się powtarzać w obrębie podprogramów danego programu jako zmienne lokalne
- adres nazywany bywa **l-wartością**, bo jest to wartość występująca po lewej stronie instrukcji przypisania:

(adres) == (wartość)

**Aliasowanie** to podpięcie pod jeden adres kilku wskaźników, np.

```
1  int x, *p, *q; p = &x; q = &x;
```

Prowadzi to do dwóch różnych etykiet dla tej samej wartości (de facto – dla tej samej zmiennej) i raczej powinno być unikane ze względów praktycznych (trudno analizować tak napisany kod, łatwo o błędy).

## Wartość

- dane reprezentowane przez zmienną
- zawartość komórek pamięci przypisanych zmiennej
- zwana też **r-wartością**, bo występuje po prawej stronie instrukcji przypisania:

(adres) == (wartość)

## Typ

- technicznie rzecz biorąc, na poziomie maszynowym, typ narzuca ilość pamięci rezerwowanej dla danej zmiennej
- od strony semantycznej, typ określa
  - ▶ interpretację danych bitowych zawartych w komórkach pamięci (znaki ASCII, liczby...)
  - ▶ ich strukturę (pojedyncze znaki, łańcuchy, tablice, rekordy...)
  - ▶ zbiór operacji, jakim dana zmienna może być poddana

## Wiązania występują na każdym poziomie

- projektowanie języka programowania (operatory wbudowane, np. + jako dodawanie liczb)
- implementacja kompilatora (real zgodny z długością słowa konkretnej architektury)
- kompilacja (związanie zmiennej z zadeklarowanym dla niej typem)
- wykonanie programu (związanie zmiennej statycznej z adresem pamięci, zmiennej lokalnej z adresem na stosie)

## Wiązanie dzielimy na

- **statyczne** – dokonywane są przed wykonaniem programu i są niezmiennie (są ustalane w trakcie kompilacji)
- **dynamiczne** – dokonywane są podczas działania programu i mogą ulegać zmianom
- **sprzętowe** – niewidoczne i niedostępne dla programisty (wiązanie statyczne może odpowiadać różnym komórkom pamięci fizycznej, a szczególnie wirtualnej, jeśli system operacyjny dokonał takiej reorganizacji)



**Wiązanie typu** polega na przypisaniu zmiennej typu. Określenie typu dla zmiennej może się odbywać na kilka sposobów

- **jawnie** – poprzez deklarację programisty
- **niejawnie** – poprzez konwencję użytej nazwy, domyślne zachowanie kompilatora
- **wnioskując z kontekstu** – statycznie lub dynamicznie

Istnieją języki (np. Ruby), które przypisują typ zmiennej na bieżąco, „w miarę potrzeby”.

```

1  irb(main):001:0> x = 1000 ; puts(x) ; x.class
2  1000
3  => Fixnum
4  irb(main):002:0> x = x*100000 ; puts(x) ; x.class
5  100000000
6  => Fixnum
7  irb(main):003:0> x = x*100000 ; puts(x) ; x.class
8  100000000000000
9  => Bignum
10 irb(main):004:0> x = x*100000 ; puts(x) ; x.class
11 100000000000000000000
12 => Bignum
13 irb(main):005:0> x = x/100000000000 ; puts(x) ; x.class
14 100000000
15 => Fixnum
16 irb(main):007:0> x = x/100000 ; puts(x) ; x.class
17 1000
18 => Fixnum

```

UWAGA: Języki programowania określa się jako typowane statycznie/dynamicznie, silnie/słabo. Jaka jest różnica?

### Typowanie dynamiczne vs. statyczne

Język typowany **statycznie** ma typy zmiennych ustalane podczas kompilacji. Te typy się nie mogą zmieniać.

Język typowany **dynamicznie** ma typy zmiennych ustalane w trakcie wykonywania programu. Te typy mogą się zmieniać. W praktyce takiego programu nie sposób skompilować, trzeba go interpretować w jakiś sposób.

## Typowanie silne vs. słabe

Język typowany **silnie** nie pozwala na mieszanie typów w pojedynczych wyrażeniach i nie zgaduje, co autor miał na myśli.

```
1 y = "3"  
2 x = y + 1  
3 --> ERROR
```

Język typowany **słabo** stara się dopasować operacje do użytych typów i mimo niespójności deklaracji zaproponować *jakiś* wynik

```
1 y = "3"  
2 x = y + 1  
3 --> (x = "31") LUB (x = 4)
```

Język może również nie być typowany. Nie istnieje w nim wtedy pojęcie typu, a dokładniej to wszystko jest jednego i tego samego typu. Przykład: Asembler (bit pattern), Tk (text), MatLab core (complex-valued matrix).

JĘZYK	statycznie	dynamicznie	słabo	silnie
Basic	x		x	
C	x		x	
C++	x			x
C#	x			x
Delphi	x			x
Fortran	x			x
Haskell	x			x
Java	x			x
Javascript		x	x	
Lisp		x		x
Pascal	x			x
Python		x	(z wyjątkami)	
Ruby		x		x

Wiele języków nie daje się tak prosto zakwalifikować. Powstają określenia takie jak: „hybrydowe”, „głównie silne”, „nie silne” itp. Takim językiem jest np. Prolog, co do którego klasyfikacji trwają debaty.

**Duck typing** – typowanie „kacze” – polega na zastosowaniu zasady: *Jeśli chodzi jak kaczką i kwacze jak kaczką, to musi być kaczką*. Jest to rodzaj typowania słabego. Rozpoznanie typu następuje poprzez analizę jego właściwości (np. metody obiektu), a nie poprzez deklarację. Na tej zasadzie działa Ruby, Python i inne.

**Okres życia zmiennej** to czas między alokacją (przydzieleniem) a dealokacją (zwolnieniem) pamięci dla tej zmiennej.

- zmienne statyczne – związane z pamięcią przed rozpoczęciem wykonania programu, dealokacja następuje w momencie kończenia pracy programu
- zmienne dynamiczne na stosie – alokacja (na stosie) w momencie dotarcia przez program podczas działania do deklaracji takiej zmiennej, dealokacja w momencie zamknięcia bloku programu, w której występowała jej deklaracja
- zmienne dynamiczne jawne na sterpie – alokacja i dealokacja jawnie w programie, odwołanie poprzez wskaźniki a nie nazwy
- zmienne dynamiczne niejawne na sterpie – alokacja i dealokacja nie są jawnie zadeklarowane w programie, proces odbywa się automatycznie bez kontroli ze strony programisty (np. tworzenie tablic)

## Zakres widoczności zmiennej

- określa bloki programu, w których zmienna jest widoczna (można się do niej odwołać)
- zmienne mogą być **globalne** lub **lokalne**
- języki mogą sprawdzać zakres widoczności zmiennych w sposób **statyczny** i w sposób **dynamiczny**



## Statyczny zakres widoczności

Kompilator napotyka odwołanie do zmiennej. Poszukiwania jej deklaracji

- zaczyna od bieżącej jednostki programu
- w dalszej kolejności przeszukuje jednostkę okalającą
- później kolejną aż do jednostki głównej (zmienne globalne)
- najbliższa **strukturalnie** deklaracja przesłania dalsze
- niektóre języki dają możliwość odwołania się do dalszych instancji zmiennej (np. C++ `klasa::zmienna`, Ada `jednostka.zmienna`)
- niektóre języki dają możliwość deklarowania zmiennej lokalnej w obrębie bloku (np. pętli `for`); są to wtedy zmienne dynamiczne na stosie

W zakresie statycznym (leksykalnym) istotne jest miejsce zadeklarowania zmiennej. Ważna w takim razie jest struktura programu. Jest to dominująca obecnie forma ustalania zakresu widoczności zmiennych.

## Przykład – zakres statyczny:

```
1   int f () {  
2       int a  
3       a=3.0  
4       int g () {  
5           print(a)  
6       }  
7       print(a)  
8   }  
9  
10  a=2.0  
11  write "g(a)=", g()  
12  write "f(a)=", f()  
13  
14 >>>  
15     g(a)=3.0  
16     f(a)=3.0
```

## Dynamiczny zakres widoczności

Kompilator napotyka odwołanie do zmiennej. Poszukiwania jej deklaracji

- zaczyna od bieżącego podprogramu
- w dalszej kolejności przeszukuje podprogram, z którego nastąpiło wywołanie bieżącego podprogramu
- później poprzedni aż do jednostki głównej (zmienne globalne)
- najbliższa **czasowo** deklaracja przesłania dalsze

W zakresie dynamicznym istotna jest kolejność wywołań w programach i podprogramach. Ważny w takim razie jest chronologiczny układ programu.

**Przykład – zakres dynamiczny:** funkcja g jest zadeklarowana wewnątrz f, ale wywoływana z poziomu funkcji głównej

```
1  int f () {  
2      int a  
3      a=3.0  
4      int g () {  
5          print(a)  
6      }  
7      print(a)  
8  }  
9  
10 a=2.0  
11 write "g(a)=", g()  
12 write "f(a)=", f()  
13  
14 >>>  
15 g(a)=2.0  
16 f(a)=3.0
```

# HASKELL

deklaratywne programowanie funkcyjne

Dodatkowa literatura:

- Paweł Urzyczyn (UW), materiały do wykładu *Rachunek Lambda*, [www.mimuw.edu.pl/~urzy/Lambda/erlambda.pdf](http://www.mimuw.edu.pl/~urzy/Lambda/erlambda.pdf)
- Małgorzata Moczurad, Włodzimierz Moczurad (UJ), materiały do wykładu *Paradygmaty programowania*, wykład 8, [wazniak.mimuw.edu.pl](http://wazniak.mimuw.edu.pl)

## Rachunek lambda

Opracowany w latach 30' XX wieku niezależnie przez Alonzo Church'a i Haskella Curry. Rachunek  $\lambda$  to reguły składania i wykonywania obliczeń, w których podstawowym elementem składowym jest **funkcja**.

Dwa główne podejścia:

- funkcje jako grafy – odwzorowanie wejścia na wyjście (podejście nowoczesne) – dwie funkcje są tożsame, jeśli dla danego wejścia dają równe sobie wyjścia
- funkcje jako reguły – dane najczęściej wzorem – dwie funkcje są tożsame, jeśli wykonują na wejściu równoważne sobie operacje (co skutkuje równym sobie wynikiem, ale teraz nie tylko wynik, ale sposób jego otrzymania jest brany pod uwagę)

Zapis funkcji  $x^2$ 

$f : x \rightarrow x^2$	funkcja jako przyporządkowanie
$f(x) = x^2$	funkcja jako reguła
$\lambda x.x^2$	funkcja w rachunku $\lambda$
$\lambda x.xx$	funkcja w rachunku $\lambda$

W zapisie funkcyjnym stosuje się wiele uproszczeń notacji:

- opuszczane są wszystkie niepotrzebne nawiasy
- opuszcza się kropkę tam, gdzie można  $\lambda xM$  (nie będę z tego korzystał)
- zakłada się, że kropka sięga najdalej na prawo jak to ma tylko sens
- funkcję kilku zmiennych  $\lambda x(\lambda y(\dots(\lambda zM)\dots))$  zapisuje się krócej jako  $\lambda xy\dots z.M$
- dla złożenia kilku wyrażeń  $MNK$  zakłada się łączność lewostronną  $(MN)K$

**Przykład:** złożenie  $(f \circ f)$  funkcji kwadratowych  $f(x) = x^2$  dla  $x = 5$

$$((\lambda f.\lambda x.f(f(x)))(\lambda y.y^2))(5) = 625$$

Napisy w rachunku  $\lambda$  to **termy** należące do zbioru  $\Lambda$ . Zbiór ten konstruuje się z pewnego przeliczalnego zbioru zmiennych  $\text{Var}$  jako zbiór słów nad alfabetem

$$\Sigma = \text{Var} \cup \{ (, ), \lambda \}$$

Mamy następujące typy termów

- **zmienna**  $x \in \text{Var} \Rightarrow x \in \Lambda$
- **lambda-abstrakcja**  $(\lambda x.M) \in \Lambda$  dla dowolnych  $M \in \Lambda$  i  $x \in \text{Var}$
- **aplikacja**  $(MN) \in \Lambda$  dla dowolnych  $M, N \in \Lambda$

To samo w notacji BNF (*Backus–Naur Form*)

$$\Lambda : \quad M, N ::= x \mid (\lambda x.M) \mid (MN)$$



Zmienne mogą być związane (lokalne) lub wolne (globalne).

Zbiór **zmiennych wolnych** (*free variables*) w danym termie definiuje się następująco:

$$\text{FV}(x) = \{x\}$$

$$\text{FV}(MN) = \text{FV}(M) \cup \text{FV}(N)$$

$$\text{FV}(\lambda x.M) = \text{FV}(M) \setminus \{x\}$$

Zmienne stają się **związane** za sprawą operatora lambda-abstrakcji  $\lambda$ . W wyrażeniu  $\lambda x.M$  zmienna  $x$ , która najpewniej występuje w termie  $M$ , staje się zmienną związaną (lokalną, ograniczoną do tego termu). Etykieta zmiennej związanej jest nieistotna i można ją zmienić na inną, nie występującą w danym termie. Oznacza to, że niektóre termy są sobie równoważne.

## Reguły podstawiania ( $\alpha$ -konwersja)

$$x[x := N] = N$$

$$y[x := N] = y \quad (y \neq x)$$

$$(PQ)[x := N] = P[x := N]Q[x := N]$$

$$(\lambda x.P)[x := N] = \lambda N.(P[x := N])$$

$$(\lambda y.P)[x := N] = \lambda y.(P[x := N]) \quad (y \neq x, y \notin \text{FV}(N))$$

Podstawianie umożliwia upraszczanie wyrażeń i znajdowanie termów sobie równoważnych.

## Twierdzenie o równoważności

(wer.1) Termy powiązane ze sobą za pomocą relacji podstawiania są sobie równoważne ( $\alpha$ -równoważne, podlegają  $\alpha$ -konwersji).

(wer.2) Relacja  $\alpha$ -konwersji między termami jest równoważna stwierdzeniu, że są one reprezentowane przez taki sam  $\lambda$ -graf.

(wer.3) Relacja  $\alpha$ -konwersji między termami to najmniejsza relacja taka, że dla wszystkich termów  $M$  i zmiennych  $y$  nie występujących w  $M$  zachodzi

$$\lambda x.M =_{\alpha} \lambda y.(M[x := y])$$

## $\beta$ -redukcja

**Relacja  $\beta$ -redukcji** to najmniejsza relacja w zbiorze termów, spełniająca warunki

$$(1) \quad (\lambda x.M)N \rightarrow_{\beta} M[x := N]$$

$$(2) \quad \text{jeśli } M \rightarrow_{\beta} M', \text{ to}$$

$$MZ \rightarrow_{\beta} M'Z, \quad ZM \rightarrow_{\beta} ZM', \quad \lambda x.M \rightarrow_{\beta} \lambda x.M'$$

## Definicja

Term postaci  $(\lambda x.M)N$  nazywany jest  **$\beta$ -redexem**.

## Definicja

Jeśli w termie  $M$  nie występuje żaden  $\beta$ -redex, to dla żadnego  $N$  nie zachodzi  $M \rightarrow_{\beta} N$  (czyli term nie podlega  $\beta$ -redukcji). Mówi się, że takie termy są w **postaci normalnej**.

## Podsumowanie:

- $\alpha$ -konwersja to zamiana etykiet zmiennych związanych; równoważność w tym sensie oznacza termy, które można przeprowadzić jeden w drugi na drodze skończonej liczby zmian nazw
- $\beta$ -redukcja to obliczenia wykonywane przez funkcję na podanym argumencie; równoważność termów w tym sensie oznacza, że można przeprowadzić jeden w drugi na drodze skończonej liczby operacji obliczenia wartości funkcji

**Przykład:** Znaleźć wyrażenie równoważne do  $(\lambda xy.xy)y$

- 1 zmienna  $y$  występuje raz w postaci związanej i raz w postaci wolnej, co powoduje problem z interpretacją notacji; stosujemy  $\alpha$ -konwersję, aby to naprawić

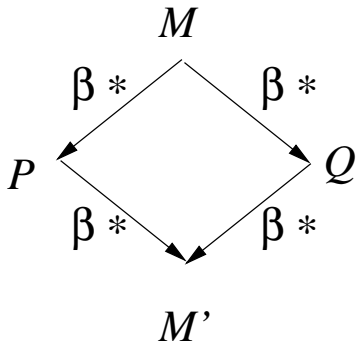
$$(\lambda xy.xy)y = (\lambda xz.xz)y$$

- 2 teraz można obliczyć wartość tego wyrażenia, stosując reguły  $\beta$ -konwersji  $(\lambda x.P)Q \rightarrow_{\beta} P[x := Q]$

$$(\lambda xz.xz)y = (\lambda x.(\lambda z.xz))y \rightarrow_{\beta} (\lambda z.xz)[x := y] = \lambda z.yz$$

## Twierdzenie Churcha–Rossera (właściwość rombu)

Podstawowe twierdzenie rachunku lambda. Jeśli term  $M$  poprzez pewne ciągi  $\beta$ -redukcji (oznaczane przez  $\xrightarrow{*}_{\beta}$ ) redukuje się niezależnie do dwóch różnych termów  $P$  i  $Q$ , to istnieje term  $M'$ , do którego można zredukować termy  $P$  i  $Q$ .



- właściwość rombu nie zachodzi dla  $\rightarrow_{\beta}$ , tylko dla  $\xrightarrow{*}_{\beta}$
- interpretacja: jeśli istnieje postać normalna, to jest ona unikalna z dokładnością do  $\alpha$ -konwersji

W termach złożonych (więcej niż jeden  $\beta$ -redeks),  $\beta$ -redukcje można wykonywać w różnej kolejności, otrzymując różne wyniki. Strategia **redukcji lewostronnej** (redukujemy pierwszy redeks od lewej) jest **strategią normalizującą**, czyli prowadzi do postaci normalnej, o ile taka istnieje.

### Twierdzenie o standaryzacji

Wyprowadzenie  $M_0 \rightarrow M_1 \rightarrow \dots M_n$  jest **standardowe**, jeżeli do dowolnego  $0 \leq i \leq (n - 2)$ , w kroku  $M_i \rightarrow M_{i+1}$  redukowany jest redeks położony nie dalej od początku termu niż redeks, który będzie redukowany w kroku następnym  $M_{i+1} \rightarrow M_{i+2}$ .

**Przykład:** redukcja lewostronna, standardowa

$$\begin{aligned} & (\lambda x.xx)((\lambda zy.z(zy))(\lambda u.u)(\lambda w.w)) \\ = & (\lambda x.xx)((\lambda z.(\lambda y.z(zy)))(\lambda u.u)(\lambda w.w)) \\ \rightarrow_{\beta} & (\lambda x.xx)((\lambda y.z(zy))[z := (\lambda u.u)](\lambda w.w)) \\ = & (\lambda x.xx)((\lambda y.(\lambda u.u)((\lambda u.u)y))(\lambda w.w)) \\ \rightarrow_{\beta} & (\lambda x.xx)((\lambda u.u)((\lambda u.u)y))[y := (\lambda w.w))] \\ = & (\lambda x.xx)((\lambda u.u)((\lambda u.u)(\lambda w.w))) \\ \rightarrow_{\beta} & (\lambda x.xx)((\lambda u.u)(\lambda w.w)) \\ \rightarrow_{\beta} & (\lambda x.xx)(\lambda w.w) \\ \rightarrow_{\beta} & (\lambda x.xx) \end{aligned}$$



**Przykład:** term  $TT$ , gdzie  $T = \lambda x.xx$  nie posiada postaci normalnej

$$TT = (\lambda x.xx)T \rightarrow_{\beta} TT$$

**Przykład:** różne strategie redukcji prowadzą do różnych wyników

$$\begin{array}{ll} (\lambda x.y)(TT) \rightarrow_{\beta} (\lambda x.y)(TT) \rightarrow_{\beta} \dots & \text{„redukując” } TT \\ \rightarrow_{\beta} y[x := TT] = y & \text{redukując lewostronnie} \end{array}$$

## Związek rachunku $\lambda$ z programowaniem

Wyrażenia Boole'a Przyjmuje się definicje

$$\mathbf{true} \equiv \lambda xy.x$$

$$\mathbf{false} \equiv \lambda xy.y$$

Pozwalają one na zbudowanie instrukcji warunkowej **if-then-else (IFE)**

$$\mathbf{IFE} \equiv \lambda x.x$$

tak, że **if (warunek) then P else Q** zapisujemy jako

$$\mathbf{IFE} \ \mathbf{true} \ P \ Q \xrightarrow{*}_{\beta} P$$

$$\mathbf{IFE} \ \mathbf{false} \ P \ Q \xrightarrow{*}_{\beta} Q$$

## Sprawdzenie

$$\begin{aligned}\text{IFE true } P \ Q &\rightarrow_{\beta} (\lambda x.x)(\lambda z.(\lambda y.z))P \ Q \\ &\rightarrow_{\beta} (\lambda x.x)(\lambda y.P)Q \\ &\rightarrow_{\beta} (\lambda x.x)(P[y := Q]) \\ &= (\lambda x.x)P \\ &\rightarrow_{\beta} P\end{aligned}$$

$$\begin{aligned}\text{IFE false } P \ Q &\rightarrow_{\beta} (\lambda x.x)(\lambda z.(\lambda y.y))P \ Q \\ &\rightarrow_{\beta} (\lambda x.x)(\lambda y.y[z := P])Q \\ &= (\lambda x.x)(\lambda y.y)Q \\ &\rightarrow_{\beta} (\lambda x.x)Q \\ &\rightarrow_{\beta} Q\end{aligned}$$

Możliwe jest zdefiniowanie operacji logicznych, np.

**AND**  $\equiv \lambda xy.(xy \text{ false}) :$

**AND** true true  $\xrightarrow{*}_{\beta}$  true

**AND** true false  $\xrightarrow{*}_{\beta}$  false

**AND** false true  $\xrightarrow{*}_{\beta}$  false

**AND** false false  $\xrightarrow{*}_{\beta}$  false

Alternatywną definicją jest **AND**  $\equiv \lambda xy.yxy$

W podobny sposób można skonstruować definicje pozostałych operacji logicznych. Podsumowując:

- **AND**  $\equiv \lambda xy.xy \text{ false}$
- **OR**  $\equiv \lambda xy.x \text{ true } y$
- **NEG**  $\equiv \lambda x.x \text{ false true}$

Liczebniki Churcha. W notacji rachunku  $\lambda$  liczby są reprezentowane przez abstrakcje

$$\lambda f x . x = \mathbf{0}$$

$$\lambda f x . f x = \mathbf{1}$$

$$\lambda f x . f (f x) = \mathbf{2}$$

$$\lambda f x . f (f (f x)) = \mathbf{3}$$

$$\lambda f x . f^n x = \mathbf{n}$$

Interpretacja: liczba  $\mathbf{n}$  to  $n$ -krotnie zastosowana operacja (funkcja) „następnik” do liczby zero.

**Przykład:**  $f(x) = x^2$ ,  $x = 2$

$$\begin{aligned}(\lambda x.xx)\mathbf{2} &\rightarrow_{\beta} \mathbf{2\ 2} = (\lambda f.(\lambda y.f(fy)))\mathbf{2}\\&\rightarrow_{\beta} \lambda y.\mathbf{2(2y)} = \lambda y.(\lambda f.(\lambda x.f(fx))(\mathbf{2y}))\\&\rightarrow_{\beta} \lambda yx.\mathbf{2y(2yx)} = \lambda yx.(\lambda f.(\lambda z.f(fz)))y(\mathbf{2yx})\\&\rightarrow_{\beta} \lambda yx.(\lambda z.y(yz))(\mathbf{2yx}) = \lambda yx.y(y\mathbf{2yx})\\&\rightarrow_{\beta} \lambda yx.y(y(\lambda f.(\lambda z.f(fz)))yx) = \lambda yx.y(y(\lambda z.y(yz))x)\\&\rightarrow_{\beta} \lambda yx.y(y(y(yx))) = \lambda fx.f^4x = \mathbf{4}\end{aligned}$$

## Standardowe funkcje wykorzystywane w rachunku lambda

NAZWA	SYMBOL	DEFINICJA	DZIAŁANIE
następnik	<b>succ</b>	$\lambda n f x. f(n f x)$	$\text{succ } a \xrightarrow{*}_{\beta} a + 1$
dodaj	<b>add</b>	$\lambda m n f x. m f(n f x)$	$\text{add } a \ b \xrightarrow{*}_{\beta} a + b$
pomnóż	<b>mult</b>	$\lambda m n f x. m(n f x)$	$\text{mult } a \ b \xrightarrow{*}_{\beta} a \cdot b$
czy zero?	<b>zero</b>	$\lambda m. m(\lambda y. \text{false}) \text{true}$	$\text{zero}(0) \xrightarrow{*}_{\beta} \text{true}$ $\text{zero}(n) \xrightarrow{*}_{\beta} \text{false}$

Istnieją alternatywne postaci tych definicji w literaturze.

## Podstawowa składnia języka Haskell

**Haskell** jest językiem

- wysokopoziomowym
- funkcyjnym
- silnie typowanym
- z typami wnioskowanymi
- leniwym



## Programowanie leniwe i gorliwe

- Funkcje leniwe – program wykonuje tylko te operacje, które w danej chwili są potrzebne.
- Funkcje gorliwe – zawsze wykonywana jest cała funkcja.

### Leniwe obliczanie funkcji

Wyrażenia nie są obliczane w momencie wiązania ich do zmiennej, ale dopiero, gdy napotkane zostanie odwołanie do konkretnego wyniku.

#### Zalety:

- wzrost wydajności poprzez oszczędne wykonywanie instrukcji
- można posługiwać się niewyliczalnymi wyrażeniami, np. nieskończonymi pętlami, nieskończonymi listami etc.

#### Wady:

- problem z występowaniem skutków ubocznych obliczeń (I/O, wyjątki, kontrola błędów etc.)
- problem z oszacowaniem wymaganej pamięci dla wykonania programu
- problem z kontrolą kolejności wykonywanych operacji

## Tablicowanie wartości funkcji

Aby usprawnić korzystanie z obliczeń leniwych korzysta się z tablic, przechowujących wyniki już obliczonych wartości danego wyrażenia dla pewnych argumentów. Przy kolejnym wywołaniu funkcji sprawdza się, czy tablica nie przechowuje interesującego wyniku, aby uniknąć jego powtórnego wyliczania ( $\rightarrow$  *memoization*).

### Przykład leniwego obliczania wartości funkcji:

```
1 takeWhile (<50) (map kw [0..]) where kw x = x*x
```

```
Prelude> let kw x = x*x
```

```
Prelude> takeWhile (<50) (map kw [0..])
```

$[0, 1, 4, 9, 16, 25, 36, 49]$

### Definicja nieskończonej listy $[1, 1, 1, \dots]$

```
1 x = 1:x in x
```

```
Prelude> let x = 1:x in x
```

1,Interrupted.

## Pytania:

- po co nam były wszystkie te „teoretyczne” wywody o gramatykach, skoro koniec końców i tak chodzi o napisanie kodu?
- nie wystarczy wziąć manual do danego języka i zapoznać się z jego składnią?

## Dokument Haskell Report 2010

[www.haskell.org/definition/haskell2010.pdf](http://www.haskell.org/definition/haskell2010.pdf)

na stronach 7–8 i dalszych podaje składnię leksykalną Haskella.

These notational conventions are used for presenting syntax:

$[pattern]$	optional
$\{pattern\}$	zero or more repetitions
$(pattern)$	grouping
$pat_1 \mid pat_2$	choice
$pat_{\langle pat' \rangle}$	difference—elements generated by $pat$ except those generated by $pat'$
<code>fibonacci</code>	terminal syntax in typewriter font

<i>program</i>	→	{ <i>lexeme</i>   <i>whitespace</i> }
<i>lexeme</i>	→	<i>qvarid</i>   <i>qconid</i>   <i>qvarsym</i>   <i>qconsym</i>   <i>literal</i>   <i>special</i>   <i>reservedop</i>   <i>reservedid</i>
<i>literal</i>	→	<i>integer</i>   <i>float</i>   <i>char</i>   <i>string</i>
<i>special</i>	→	(   )   ,   ;   [   ]   `   {   }
<i>whitespace</i>	→	<i>whitestuff</i> { <i>whitestuff</i> }
<i>whitestuff</i>	→	<i>whitechar</i>   <i>comment</i>   <i>ncomment</i>
<i>whitechar</i>	→	<i>newline</i>   <i>vertab</i>   <i>space</i>   <i>tab</i>   <i>uniWhite</i>
<i>newline</i>	→	<i>return</i> <i>linefeed</i>   <i>return</i>   <i>linefeed</i>   <i>formfeed</i>
<i>return</i>	→	a carriage return
<i>linefeed</i>	→	a line feed
<i>vertab</i>	→	a vertical tab
<i>formfeed</i>	→	a form feed
<i>space</i>	→	a space
<i>tab</i>	→	a horizontal tab
<i>uniWhite</i>	→	any Unicode character defined as whitespace

<i>comment</i>	→	<i>dashes</i> [ <i>any</i> <sub>(symbol)</sub> { <i>any</i> } ] <i>newline</i>
<i>dashes</i>	→	-- { - }
<i>opencom</i>	→	{ -
<i>closecom</i>	→	- }
<i>ncomment</i>	→	<i>opencom</i> <i>ANYseq</i> { <i>ncomment</i> <i>ANYseq</i> } <i>closecom</i>
<i>ANYseq</i>	→	{ <i>ANY</i> } { { <i>ANY</i> } ( <i>opencom</i>   <i>closecom</i> ) { <i>ANY</i> } }
<i>ANY</i>	→	<i>graphic</i>   <i>whitechar</i>
<i>any</i>	→	<i>graphic</i>   <i>space</i>   <i>tab</i>
<i>graphic</i>	→	<i>small</i>   <i>large</i>   <i>symbol</i>   <i>digit</i>   <i>special</i>   "   '

<i>small</i>	→	<i>ascSmall</i>   <i>uniSmall</i>   _
<i>ascSmall</i>	→	a   b   ...   z
<i>uniSmall</i>	→	any Unicode lowercase letter
<i>large</i>	→	<i>ascLarge</i>   <i>uniLarge</i>
<i>ascLarge</i>	→	A   B   ...   Z
<i>uniLarge</i>	→	any uppercase or titlecase Unicode letter
<i>symbol</i>	→	<i>ascSymbol</i>   <i>uniSymbol</i> <sub>(<i>special</i>   _   "   ' )</sub>

<i>ascSymbol</i>	→	!   #   \$   %   &   *   +   .   /   <   =   >   ?   @   \   ^       -   ~   :
<i>uniSymbol</i>	→	any Unicode symbol or punctuation
<i>digit</i>	→	<i>ascDigit</i>   <i>uniDigit</i>
<i>ascDigit</i>	→	0   1   ...   9
<i>uniDigit</i>	→	any Unicode decimal digit
<i>octit</i>	→	0   1   ...   7
<i>hexit</i>	→	<i>digit</i>   A   ...   F   a   ...   f

## 2.4 Identifiers and Operators

<i>varid</i>	→	$(small \{small \mid large \mid digit \mid ' \})_{\langle reservedid \rangle}$
<i>conid</i>	→	$large \{small \mid large \mid digit \mid ' \}$
<i>reservedid</i>	→	case   class   data   default   deriving   do   else   foreign   if   import   in   infix   infixl   infixr   instance   let   module   newtype   of   then   type   where   _



## 2.5 Numeric Literals

*decimal* → *digit*{ *digit* }

*octal* → *octit*{ *octit* }

*hexadecimal* → *hexit*{ *hexit* }

*integer* → *decimal*

| 0o *octal* | 0O *octal*

| 0x *hexadecimal* | 0X *hexadecimal*

*float* → *decimal* . *decimal* [*exponent*]

| *decimal exponent*

*exponent* → (e | E) [+ | -] *decimal*

## 2.6 Character and String Literals

<i>char</i>	→	' ( <i>graphic</i> <sub>&lt;'   \&gt;</sub>   <i>space</i>   <i>escape</i> <sub>&lt;\&amp;&gt;</sub> ) '
<i>string</i>	→	" { <i>graphic</i> <sub>&lt;"   \&gt;</sub>   <i>space</i>   <i>escape</i>   <i>gap</i> } "
<i>escape</i>	→	\ ( <i>charesc</i>   <i>ascii</i>   <i>decimal</i>   o <i>octal</i>   x <i>hexadecimal</i> )
<i>charesc</i>	→	a   b   f   n   r   t   v   \   "   '   &
<i>ascii</i>	→	^ <i>cntrl</i>   NUL   SOH   STX   ETX   EOT   ENQ   ACK   BEL   BS   HT   LF   VT   FF   CR   SO   SI   DLE   DC1   DC2   DC3   DC4   NAK   SYN   ETB   CAN   EM   SUB   ESC   FS   GS   RS   US   SP   DEL
<i>cntrl</i>	→	<i>ascLarge</i>   @   [   \   ]   ^   _
<i>gap</i>	→	\ <i>whitechar</i> { <i>whitechar</i> } \

## Konwencje w skrócie i bardziej po ludzku:

- argumenty funkcji podaje się bez nawiasów, chyba, że są to „krotki”
- nazwy funkcji nie zaczynają się wielkimi literami
- apostrof po nazwie funkcji zwyczajowo oznacza funkcję gorliwą (*strict function*), w odróżnieniu od szybszej funkcji leniwej (*lazy function*), ale nie ma znaczenia syntaktycznego
- bloki programu oddzielane są pustą linią
- kontynuacje deklaracji mają wcięcia (najlepiej, aby były one zgodne z początkiem deklaracji)
- deklaracje zagnieżdżone powinny mieć coraz głębsze wcięcia
- linia nie powinna przekraczać 78 znaków
- (niepolecane) bloki można też ograniczać nawiasami { ... }, zaś znaki końca linii zastępować średnikiem

## Formalna struktura kodu w Haskellu

- **moduł**
- moduł składa się z **deklaracji**, takich jak: wartości (*values*), typy danych (*datatypes*), klasy typów (*type classes*), informacje naprawcze (*fixity information*)
- na deklaracje składają się **wyrażenia**
- wyrażenia tworzone są na podstawie **składni leksykalnej** języka (na tym poziomie mówimy o pliku tekstowym zawierającym kod programu)

Wyrażenie (*expression*) jest ewaluowane do wartości z **przypisanym statycznie typem**.

## Układ kodu w Haskellu

- Haskell bazuje na blokach kodu, a definicje produkcji gramatyki tego języka zawierają często średniki i nawiasy `{ ..;..;.. }`; ograniczające te bloki
- taki styl kodowania nazywany jest ***layout-insensitive*** i jest użyteczny, gdy kod Haskellu ma być generowany przez inny automat/program
- ze względów praktycznych nawiasy ograniczające bloki mogą być pominięte, jednak wtedy konieczne jest zapisywanie komend w nowych liniach i odpowiednie używanie narastających wcięć
- taki styl kodowania nazywany jest ***layout-sensitive*** i jest powszechnie używany w większości zastosowań

Bloki są ważne ze względu na zakres widoczności (*scope*) funkcji i zmiennych.

Mniej formalnie o strukturze programu:

- deklaracja funkcji `main`
- deklaracje innych funkcji
- komentarze zaczynają się od `--` w dowolnym miejscu linii, komentarze zagnieżdżone są ograniczone `{- ... -}`

## Przykładowy program

```
1  main :: IO ()
2  main = do
3      let t=[2,1,2,3,4,5,6,7,8,9,10]
4      print (parzyste t)
5      print (parzyste [])
6
7  parzyste :: [Integer] -> [Integer]
8  parzyste [] = []
9  parzyste (g:o) = if even g then g:parzyste o
10                  else parzyste o
```

## Listy

- uszeregowane elementy jednego typu
- mogą być nieskończone i zmieniać swoją długość

```
a=[1..10]
```

```
a=['a'..'z']
```

```
a=[2,4..20]
```

Operator : dodaje element na początek listy

```
1:2:3:[]
```

pozwała też rozdzielić listę na jej **głowę** i **ogon**

## Operator ++ scala listy

```
"Hello" ++ " " ++ "world!"
```

## Operator !! pobiera element listy (numeracja od 0)

```
"Haskell!" !! 3
```

## Wbudowane funkcje operacji na listach to m.in.

```
head a           Prelude> let a=["A","B","C","D","E"]
tail a           Prelude> head a
last a           "A"
init a           Prelude> tail a
length a         ["B","C","D","E"]
                  Prelude> last a
                  "E"
                  Prelude> init a
                  ["A","B","C","D"]
                  Prelude> length a
                  5
```



## Krotki

- mają z góry ustaloną długość
- mogą zawierać elementy różnych typów
- służą np. do opisu funkcji wieloargumentowych
- definicja w nawiasach okrągłych, np. (a,b,c)

**Zbiory i przedziały** Oprócz list i krotek możliwe jest zdefiniowanie skończonego lub nieskończonego zbioru elementów, generowanych za pomocą danej reguły. Elementy te są przechowywane w postaci listy.  
Przykład: zbiór liczb parzystych mniejszych od 31

```
[x*2 | x<-[1..15]]
```

**Instrukcja warunkowa** `if..then..else..`

## Przykładowa definicja funkcji

Funkcja signum

$$\text{sgn}(x) = \begin{cases} -1 & \text{gdy } x < 0 \\ 0 & \text{gdy } x = 0 \\ +1 & \text{gdy } x > 0 \end{cases}$$

Wersja 1:

```
1 sgn1 :: Integer -> Integer
2 sgn1 n = if n>0 then 1 else if n==0 then 0 else -1
```

## Wersja 2:

```
1  sgn2  :: Integer -> Integer
2  sgn2  n
3      | n>0    = 1
4      | n==0   = 0
5      | n<0    = -1
```

## Wersja 3:

```
1  sgn3  :: Integer -> Integer
2  sgn3  0 = 0
3  sgn3  n
4      | n>0    = 1
5      | n<0    = -1
```

## Funkcje anonimowe.

Funkcje anonimowe nie mają nadawanej nazwy. Składnia:

```
1 Prelude> (\x -> x*x) 2
2 4
3 Prelude> (\x -> x*x) ((\y -> y+4) 2)
4 36
```

przypomina (celowo) zapis z rachunku  $\lambda$ .

W ten sposób można np. zweryfikować instrukcję warunkową IFE

```
1 Prelude> let true = \x y -> x
2 Prelude> let false = \x y -> y
3 Prelude> let ife = \x -> x
4 Prelude> ife true 4 6
5 4
6 Prelude> ife false 4 6
7 6
```

Definicja typu Church zamieniającego liczebnik Churcha na liczbę i odwrotnie

```
1  type Church a = (a -> a) -> a -> a
2
3  church :: Integer -> Church Integer
4  church 0 = \f x -> x
5  church n = \f x -> f (church (n-1) f x)
6
7  unchurch :: Church Integer -> Integer
8  unchurch cn = cn (+ 1) 0
```

Po załadowaniu definicji do konsoli ghci:

```
1 *Main> unchurch (\f x -> f(f(f(f x))))  
2 4  
3 *Main> church 5  
4  
5 <interactive>:4:1:  
6 No instance for (Show (Church Integer))  
7   arising from a use of 'print'  
8 Possible fix:  
9   add an instance declaration  
10   for (Show (Church Integer))  
11 In a stmt of an interactive  
12 GHCi command: print it
```

możliwe jest tłumaczenie liczebników Churcha na liczby i na odwrót.  
Kłopot jest z wyświetlaniem liczebników w postaci funkcji anonimowych.

Można dołożyć do kodu moduł

```
1 import Text.Show.Functions
```

i wtedy ghci nie zgłasza błędu (nowy zapis: \$ oznacza nawias otwierający)

```
1 *Main> church 5  
2 <function>  
3 *Main> church $ unchurch (\f x -> f(f(f(f x))))  
4 <function>  
5 *Main> unchurch $ church 4  
6 4
```

Dlaczego nie jest wyświetlana funkcja?

- Funkcja  $\backslash f\ x = f(f\ x)$  jest tożsama  $\backslash g\ y = g(g\ y)$  ( $\alpha$ -konwersja). Którą z nich miałyby Haskell wyświetlić?
- Złamana zostałaby reguła, że dane wejście funkcja ewaluuje do jednoznacznie określonego wyjścia.



## Typy polimorficzne

Typy w Haskellu określają (przeciw)dziedziny funkcji:

- Int, Integer – krótki, długi integer
- Float, Double – pojedynczej i podwójnej precyzji liczba zmiennoprzecinkowa
- Bool
- Char

Typy są pogrupowane w klasy typów (*type classes*). Klasy te

- określają wspólne właściwości typów należących do danej klasy
- niektóre zawierają się w innych
- pozwalają na pisanie definicji funkcji dla całej klasy typów, a nie dla poszczególnych typów zmiennych, tzw. **funkcje polimorficzne**

## Funkcja zwykła

```
1 plusDwa :: Integer -> Integer
2 plusDwa x = x+2
```

jest dobrze określona tylko dla zmiennej typu `Integer` i żadnej innej. Haskell ma ścisłą kontrolę typów. Zamiast pisać nowe funkcje dla typów `Int`, `Float` i `Double` wystarczy zmienić nagłówek

```
1 plusDwa :: (Num a) => a -> a
2 plusDwa x = x+2
```

Klasa typów `Num` obejmuje wszystkie typy numeryczne.

# Klasy typów w Haskellu

Definicja klasy zazwyczaj określa funkcję (operator), którą można zastosować do typów należących do danej klasy.

KLASA	DEFINICJA
Eq	równość elementów ==, /=
Ord	porządkowanie elementów <, <=, >=, >; podklasa Eq
Ordering	porządkowanie elementów GT, EQ, LT
Show	elementy reprezentowane przez ciągi znakowe (funkcja show)
Read	elementy reprezentowane przez ciągi znakowe (funkcja read)
Enum	wyliczenia (np. listy); zawiera Bool, Char, Ordering, Num
Bounded	wartości elementów ograniczone
Num	typy numeryczne; muszą należeć do Eq i Show
Integral	Int i Integer
Floating	Float i Double

**Operatory** w Haskellu też są funkcjami. Np. operator binarny, który dwa argumenty typu `a` zamienia na inny element typu `a` jest funkcją

$$a \rightarrow a \rightarrow a.$$

Aby zaznaczyć alternatywny sposób wywołania funkcji (prefix i infix) stosuje się nawiasy i odwrotne cudzysłowy.

- jeśli w definicji nazwa funkcji ujęta jest w nawiasy, to domyślnie jest to definicja infiksowa
  - ▶ przy wywołaniu prefikсовym te nawiasy muszą pozostać
  - ▶ przy wywołaniu infiksowym nawiasy się pomija
- jeśli w definicji nazwa funkcji nie jest ujęta w nawiasy, to domyślnie jest to definicja prefiksowa
  - ▶ przy wywołaniu prefikсовym podaje się nazwę funkcji
  - ▶ przy wywołaniu infiksowym nazwę funkcji otacza się odwrotnymi cudzysłowami

(Przykłady z wykładu Moczurad i Moczurad)

### Przykład 1. Definicja i poprawne wywołania

```
1  (%%%) :: Integer -> Integer -> Integer
2  (%%%) x y = x + 5 * y
```

3 %%% 7

(%%%) 3 7

### Przykład 2. Definicja i poprawne wywołania

```
1  abc :: Integer -> Integer -> Integer
2  abc x y = 2 * x + y * y
```

3 'abc' 7

abc 3 7

### Przykład 3. Składanie funkcji: dwa równoważne wywołania

(8+4) 'div' 3

(( 'div' 3 ). (+4)) 8

## Definicje rekurencyjne

### Przykład 1.1 Definicja funkcji maksimum dla list

```
1  maximum :: (Ord a) => [a] -> a
2  maximum [] = error "Pusta lista"
3  maximum [x] = x
4  maximum (x:ogon)
5      | x > maxOgon = x
6      | otherwise = maxOgon
7      where maxOgon = maximum ogon
```

### Przykład 1.2 Definicja funkcji maksimum dla list; wykorzystuje wbudowaną funkcję max

```
1  maximum :: (Ord a) => [a] -> a
2  maximum [] = error "Pusta lista"
3  maximum [x] = x
4  maximum (x:ogon) = max x (maximum ogon)
```

## Przykład 2 Funkcja odwracająca listę

```
1 reverse :: [a] -> [a]
2 reverse [] = []
3 reverse (x:ogon) = reverse ogon ++ [x]
```

- odwrotnością listy pustej jest lista pusta
- odwrotną listę można skonstruować przenosząc pierwszy element na koniec, pod warunkiem, że reszta jest już odwróconą listą
- tak więc wywołujemy reverse na coraz krótszych listach (punktem zwrotnym rekurencji jest lista pusta), trzymając elementy, które będą dopisywane na końcu w pamięci

Schematycznie:

$$\begin{aligned} r[1,2,3,4] &= r[2,3,4]+[1] = r[3,4]+[2]+[1] = \\ r[4]+[3]+[2]+[1] &= r[]+[4]+[3]+[2]+[1] = \\ [4]+[3]+[2]+[1] &= [4,3]+[2]+[1] = [4,3,2]+[1] = [4,3,2,1] \end{aligned}$$

## Monady i składanie funkcji

Ograniczenia języka czysto funkcyjnego:

- wszystko jest funkcją, więc każde „polecenie” musi zwrócić wartość
- funkcja zawsze zwraca tę samą wartość dla tego samego argumentu
- funkcja zawsze zwraca **jedną** wartość (jest jednoznacznie określona)
- funkcje można składać  $f(g(x)) = (f \circ g)(x)$

```
1 Prelude> let f x = x+2
2 Prelude> let g x = x-2
3 Prelude> (f.g) 5
4 5
5 Prelude> (g.f) 5
6 5
```



Złożenie funkcji  $f.g$  było możliwe, ponieważ ich typy na to zezwalały

```
1 Prelude> :t f
2 f :: Num a => a -> a
3 Prelude> :t g
4 g :: Num a => a -> a
```

czyli przeciwdziedzina  $g$  była zawarta w dziedzinie  $f$ . Ale nie zawsze tak musi być!

### Problem: efekty uboczne

Dla języka funkcyjnego problemem jest sytuacja, w której funkcja ma robić kilka rzeczy, np. obliczać wartość i wypisywać dane na ekranie. Czysta funkcja nie może generować efektów ubocznych, więc w takich sytuacjach efekt ten musi być częścią zwracanej wartości.

Rozwiązaniem tego i innych problemów z funkcjami są **monady**. Monada pozwala na zdefiniowanie

- reguł składania funkcji o częściowo niezgodnych typach
- reguł przekazywania wartości funkcji na wejście innej funkcji

Proszę zapoznać się z artykułami:

`blog.sigfpe.com/2006/08/`

`you-could-have-invented-monads-and.html`

`en.wikipedia.org/wiki/Monad_(functional_programming)`

`wiki.haskell.org/All_About_Monads`

Przykład: funkcje liczą i wyświetlają komunikaty

```
1  f :: Float -> (Float, String)
2  f x = (2*x, " Dziala f ")
3
4  g :: Float -> (Float, String)
5  g x = (x+1, " Dziala g ")
```

Intuicyjnie, złożenie funkcji  $f.g$  powinno działać tak, że zwraca wartość  $2*(x+1)$  i wyświetla napis " Dziala f Dziala g "

Wbudowane złożenie `f.g` jest niemożliwe ze względu na niezgodność typów

```
1 Main> (f.g) 3
2
3 <interactive>:7:4:
4   Couldn't match expected type 'Float'
5       with actual type '(Float, String)'
6   Expected type: Float -> Float
7   Actual type: Float -> (Float, String)
8   In the second argument of '(.)', namely 'g'
9   In the expression: f . g
```

Oczywiście można napisać swoją funkcję, która będzie realizowała złożenie  $f.g$

```
1 fg :: Float -> (Float, String)
2 fg x = (fst(f (fst(g x))), snd(f x) ++ snd(g x))
```

ale jest ona specyficzna dla naszego konkretnego przypadku. Poza tym nie jest ona pomocna, gdyby interesowało nas złożenie  $g.f$  mimo, że od strony typów sytuacja jest analogiczna.

Rozważmy funkcję bind (2 równoważne zapisy)

```
1 bind :: (Float -> (Float, String)) ->
2       ((Float, String) -> (Float, String))
3 bind f (gx, gs) = let (fx, fs) = f gx
4                   in (fx, fs++gs)
```

```
1 bind :: (Float -> (Float, String)) ->
2       ((Float, String) -> (Float, String))
3 bind f (gx, gs) = (fx, fs++gs)
4                   where (fx, fs) = f gx
```

Składnia z let... in... oraz where pozwala na

- łatwe oznaczanie partii kodu nową zmienną lokalną (jak w matematyce)
- utrzymanie przejrzystości wyrażeń (jak w matematyce)
- większą elegancję kodu (jak w... ?)

```
1 *Main> bind f (g 3)
2 (8.0," Dziala f Dziala g ")
3 *Main> bind g (f 3)
4 (7.0," Dziala g Dziala f ")
```

## Funkcja bind

- definiuje ogólny schemat składania funkcji typu `Float -> (Float,String)`
- pozwala na złożenie tych funkcji w dowolnej kolejności
- jest składową monady

Ogólny schemat:

powiedzmy, że mamy funkcję ukrywającą strukturę danych `Ukryj`.  
Definiujemy nowy typ danych – dane ukryte i funkcję ukrywającą

```
1 data Ukryte a = Ukryj a
2
3 return :: a -> Ukryte a
4 return x = Ukryj x
```

Aby operować na ukrytych danych bez ich odkrywania definiuje się funkcję wiążącą

```
1 bind :: (a -> Ukryte b) -> (Ukryte a -> Ukryte b)
2 bind f (Ukryj x) = f x
```

lub

```
1 bind :: (a -> Ukryte b) -> (Ukryte a -> Ukryte b)
2 bind f (return x) = f x
```



Wbudowany operator `>=>` pozwala na składanie funkcji, czyli pełni rolę funkcji `bind`.

$$\text{bind } f \text{ (return } x) \quad \Leftrightarrow \quad x \gg= f$$

Operator `>=>` jest różnie zdefiniowany, w zależności od typu monady. Najprostsza jego definicja to

wartość `>=>` funkcja

w której wartość z lewej strony jest przenoszona jako argument funkcji po prawej stronie i zwracana jest wartość tej funkcji na tym argumencie.

W Haskellu monady predefiniowane są poprzez odpowiednie klasy. Standardowa klasa `Monad` jest zdefiniowana następująco:

```
1 class Monad m where
2   (>>=)    :: m a -> (a -> m b) -> m b
3   return   :: a -> m a
4   (>>)      :: m a -> m b -> m b
5   fail     :: String -> m a
6
7   m >> k = m >>= (\_ -> k)
8   fail k = error k
```

- `m` jest **konstruktorem typu** monady
- `return` jest funkcją **tworzącą instancje** typu `m`
- `>>=` jest funkcją **wiążącą** instancje monady z obliczeniami
- `>>` operator analogiczny do `>>=`, ale nie wykorzystujący wyniku poprzedniej operacji
- `fail` jest wywoływane, gdy coś się nie powiedzie w konstrukcji złożenia

Funkcje `»` i `fail` nie występują w matematycznej definicji monady, ale są częścią klas z rodziny `Monad` w Haskellu, gdyż ułatwia to programowanie. Części składowe monady muszą spełniać 3 podstawowe prawa:

- ❶  $(\text{return } x) \gg= f == f\ x$
- ❷  $m \gg= \text{return} == m$
- ❸  $(m \gg= f) \gg= g == m \gg= (\lambda x \rightarrow f\ x \gg= g)$

których interpretacja jest następująca:

- ❶ `return` jest lewostronnym elementem neutralnym dla funkcji `»=`
- ❷ `return` jest prawostronnym elementem neutralnym dla funkcji `»=`
- ❸ rodzaj prawa łączności obowiązującego funkcję `»=`

**Przykład:** mamy do czynienia z obliczeniami, które mogą się nie udać, tj. są sytuacje, w których funkcja nie zwróci wartości. Jest to problematyczne zachowanie w języku funkcyjnym. Wykorzystuje się wtedy monadę opartą o wbudowany konstruktor typu `Maybe`

```
1 data Maybe m = Nothing | Just m
```

Można zadeklarować `Maybe` jako składnik klasy `Monad`

```
1 instance Monad Maybe where
2   Nothing >>= f      = Nothing
3   (Just x) >>= f     = f x
4   return           = Just
```

Przykład c.d. Klonowanie owiec – nie każda owca ma dobrze określonego ojca i matkę.

```
1  type Owca = ...
2
3  tata :: Owca -> Maybe Owca
4  tata = ...
5
6  mama :: Owca -> Maybe Owca
7  mama = ...
8
9  -- ojciec ojca mamy
10 oom :: Owca -> Maybe Owca
11 oom o = (Just o) >>= mama >>= tata >>= tata
```

Taka konstrukcja automatycznie będzie poprawnie traktowała sytuację, gdy w ciągu złożenia pojawi się brak wartości którejś funkcji (`Nothing`). Rozkładając ciąg

```
(Just o) >>= mama >>= tata >>= tata
```

z definicji otrzymamy

```
tata(tata(mama o))
```

o ile każda z tych funkcji zwróci wartość, w przeciwnym razie wynikiem będzie `Nothing`.

Do czego w praktyce służą monady?

- funkcje wieloargumentowe są rozbijane na jednoelementowe i sklejane razem, np.

$$\text{plus } x \ y = \ \backslash x \rightarrow (\backslash y \rightarrow x+y)$$

- listy są tworamami o nieustalonej z góry długości, być może puste, więc monada typu `List` pozwala na definiowanie obliczeń mogących zwracać 0, 1 lub więcej wartości
- operacje I/O są obsługiwane w Haskellu poprzez odpowiednią monadę, ponieważ są to najczęściej efekty uboczne działania funkcji (właściwe obliczenia plus operacja I/O)
- obsługa błędów

W Haskellu istnieje biblioteka wzorców monad (Monad Template Library), którą można włączyć w ghc wywołaniem `Control.Monad`. Niektóre z nich to:

MONADA	DO CZEGO UŻYTECZNA?
Identity	używana przy konstrukcjach innych monad
Maybe	obliczenia mogące nie zwrócić wartości
Error	obliczenia mogące zawieść
[] (List)	obliczenia o nieokreślonej liczbie wartości
IO	obliczenia połączone z operacjami I/O
State	obliczenia przekazujące parametr stanu
Reader	obliczenia czytające wejście
Writer	obliczenia produkujące dodatkowy strumień danych
Cont	obliczenia, które można wstrzymać i wznowić



# PROLOG

deklaratywne programowanie w logice

Dodatkowa literatura:

- D. Diaz, GNU Prolog Manual  
<http://www.gprolog.org/manual/gprolog.html>  
<http://www.gprolog.org/manual/gprolog.pdf>
- W.F. Clocksin, C.S. Mellish, Prolog. Programowanie (Helion)

**Prolog** oznacza Programming in Logic. Jest to język **deklaratywny** oparty na wyrażeniach **logiki matematycznej**. Opisany został w 1972 przez Alain Colmerauer (standardy ISO w 1995 i 2000). Używany w takich dziedzinach jak:

- relacyjne bazy danych
- logika matematyczna
- problemy abstrakcyjne
- języki naturalne
- automatyka
- algebra symboliczna
- biochemia
- sztuczna inteligencja

Logika użyta w Prologu jest standardową dwuwartościową logiką matematyczną, opisywaną algebrą Boole'a. Podstawowe operacje to

- i (AND)
- lub (OR)
- negacja (NOT)
- jeśli (IF)

zaś wartości logiczne to `true` i `false`.

Prolog stara się uzgodnić wszystkie zdania w programie do logicznej wartości `true`.

**Algebra Boole'a** składa się ze zbioru

$$A = \{0, 1\}$$

oraz „dodawania” i „mnożenia” zdefiniowanych dla  $a \in A$  jako:

$$0 + a = a$$

$$0 \cdot a = 0$$

$$1 + a = 1$$

$$1 \cdot a = a$$

Negacja zdefiniowana jest przez:

$$a + \bar{a} = 1$$

$$a \cdot \bar{a} = 0$$

$$\bar{\bar{a}} = a$$

## Elementy algebry odpowiadają elementom z logiki

0	FALSE
1	TRUE
+	OR
·	AND
$\bar{x}$	NOT

## Właściwości algebry Boole'a:

Łączność

$$x + (y + z) = (x + y) + z$$

Łączność

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z$$

Przemienność

$$x + y = y + x$$

Przemienność

$$x \cdot y = y \cdot x$$

Rozdzielność

$$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$$

Rozdzielność

$$x + (y \cdot z) = (x + y) \cdot (x + z)$$

Element neutralny

$$x + 0 = x$$

Element neutralny

$$x \cdot 1 = x$$

## Właściwości algebry Boole'a c.d.:

Idempotentność

$$x + x = x$$

Idempotentność

$$x \cdot x = x$$

Anihilator

$$x \cdot 0 = 0$$

Anihilator

$$x + 1 = 1$$

Absorpcja

$$x \cdot (x + y) = x$$

Absorpcja

$$x + (x \cdot y) = x$$

Dopełnienie

$$x + \bar{x} = 1$$

Dopełnienie

$$x \cdot \bar{x} = 0$$

prawo de Morgana

$$\overline{x + y} = \bar{x} \cdot \bar{y}$$

prawo de Morgana

$$\overline{x \cdot y} = \bar{x} + \bar{y}$$

**Klauzula Horna** jest wyrażeniem logicznym, w którym co najwyżej jeden człon nie jest zanegowany; jej ogólna postać to:

$$\bar{p}_1 + \bar{p}_2 + \cdots + \bar{p}_n + q$$

Można je przekształcić do postaci równoważnej, która jest użyteczna w programowaniu:

$$(\bar{p}_1 + \bar{p}_2 + \bar{p}_3 + q) \Leftrightarrow (p_1 \cdot p_2 \cdot p_3 \Rightarrow q)$$

W notacji Prologu takie stwierdzenie ma postać

1     `q :- p1 , p2 , p3 .`

co należy rozumieć jako: „q jest prawdziwe JEŚLI (p1 i p2 i p3) jest prawdziwe”

Programy w Prologu to niemal wyłącznie zestawy klauzul Horna.



Podstawowymi pojęciami w Prologu są **termy**, wiązane **uporządkowanymi relacjami**.

Przykład:

- termem mogą być jaś, małgosia
- relacjami mogą być brat, siostra
- relacje są uporządkowane, co oznacza, że jaś jest w relacji brat z termem małgosia, ale niekoniecznie na odwrót

Schemat programu w Prologu

- ❶ deklaracja **faktów** (baza danych termów i relacji)
- ❷ deklaracja **reguł** dozwolonych do manipulowania faktami
- ❸ deklaracja problemu (zadanie pytania)

Przykład bazy wiedzy (bazy danych, faktów):

```
1   rodzenstwo(jas ,malgosia) .
2   rodzenstwo(malgosia ,jas) .
3   brat(jas ,malgosia) .
4   siostra(jas ,malgosia) .
```

Uwaga:

- nie ma potrzeby definiowania, czym są użyte termy jas, brat etc.
- definicja tworzona poprzez właściwości
- predykat(argument1,argument2) . jest **relacją**
- etykiety stałych zaczynają się małą literą
- definicje relacji kończą się kropką

W powyższym przykładzie zdefiniowane zostały trzy **predykaty** (rodzenstwo/2, brat/2, siostra/2) i dwie **stałe** (jas, malgosia).

- predykaty służą do definiowania relacji
- argumentami predykatów są **termy proste** (czyli stałe lub zmienne)
- **termy złożone** to termy proste połączone **funktorami** (o znaczeniu logicznego AND, OR)

Relacje definiowane są zupełnie abstrakcyjnie, np.  $x(a,b)$ , jednak użyte etykiety sugerują użytkownikowi przypisywane im znaczenie. Poniższe relacje mają dla Prologu identyczną budowę:

```
1  x(a,b).  
2  kolor(czerwony,papryka).  
3  ojciec(zdzich,eustachy).  
4  wieksze(trzy,cztery).
```

- każda stała to osobny byt
- różne stałe reprezentują różne byty...
- ... nawet jeśli wartości przechowywane przez te stałe są takie same!

Przykład w Fortranie

```

1      x=2
2      y=2
3      if (x.eq.y) then ...      ! true

```

Przykład w Prologu

```

1      | ?- x=2, y=2, x=y.
2
3      no

```

Tak naprawdę, to już stwierdzenie `x=2` daje odpowiedź `no` w Prologu. Dlaczego?

**Zmienna** w Prologu ma nieco inne właściwości, niż w innych językach programowania:

- nie jest typowana w momencie inicjalizacji
- pierwsze wyrażenie, które przypisze zmiennej jakąś wartość, ustala tę wartość raz na zawsze (uwaga: nawracanie!)
- np., wyrażenie postaci `if X = 2 then...` w Fortranie nie podstawia pod zmienną  $X$  liczby 2, lecz analogiczne wyrażenie w Prologu może to zrobić, jeśli  $X$  jest zmienną bez przypisanej wartości\*)
- etykiety zmiennych zaczynają się wielkimi literami, np.  $X$ , Kuba...

\*) *Prolog traktuje wyrażenie  $X=2$  jako wyrażenie logiczne, które musi być spełnione, aby można było kontynuować wykonywanie programu. Dlatego też jeśli  $X$  jest zmienną nieprzypisaną, takie przypisanie nastąpi i warunek otrzyma logiczną wartość `true`.*

Zmiennych w Prologu używa się, aby

- zadawać pytania, jako miejsce do przechowania wyniku
- działały jako akumulator w algorytmach rekurencyjnych

```
1  kolor(czerwony , auto) .  
2  kolor(zielony , ufoludek) .  
3  
4  |  ?-  kolor(X, auto) .  
5  
6  X = czerwony  
7  
8  yes
```

## Podstawowe funktory logiczne

przecinek	,	AND
średnik	;	OR
dwukropek myślnik	: —	IF

Warunki czytane są i wykonywane (interpretowane) **od lewej do prawej**.

Kolejność ma znaczenie, mimo że wiele z nich, z formalnego punktu widzenia, jest połączona funktorami symetrycznymi względem przestawienia argumentów.

## Przykład:

```
1  lata(samolot).  
2  lata(mucha).  
3  kolor(czarny,mucha).  
4  kolor(czarny,samolot).  
5  
6  | ?- lata(X),kolor(czarny,X).  
7  X = samolot ? a  
8  X = mucha  
9  no  
10  
11 | ?- kolor(czarny,X),lata(X).  
12 X = mucha ? a  
13 X = samolot  
14 yes
```



Wnioski z poprzedniego przykładu:

- jeśli istnieje więcej niż jedno rozwiązanie, Prolog znajdzie je wszystkie
- kolejność znajdowania rozwiązań zależy m.in. od kolejności instrukcji w programie (są one czytane z góry na dół, od lewej do prawej)
- znajdowanie różnych rozwiązań jest możliwe, gdyż Prolog pracuje na **relacjach**, a nie na **funkcjach**
- cofnięcie się i przyjęcie innej strategii uzgadniania rozwiązania nazywane jest **nawracaniem** (ang. *backtracking*) i jest charakterystyczną cechą języków programowania w logice

**Reguły** w Prologu (fakty) mają postać

GŁOWA — IF — WARUNEK (WARUNKI)

```
1   siostra(małgosia,X) :- brat(X,małgosia),
2                               kobieta(małgosia).
```

Modus operandi Prologu w takim przypadku obejmuje

- podstaw pod X z głowy reguły stałą występującą w programie
- sprawdź czy takie X spełnia iloczyn warunków brat/2 i kobieta/1
- jeśli tak, spytaj się, czy szukać kolejnych rozwiązań
- ewentualnie przeszukaj bazę wiedzy i sprawdź kolejne X
- powtarzaj, aż wyczerpią się możliwości lub użytkownik przerwie

**Funktory** łączą termy proste tworząc termy złożone. Mogą przyjąć formę:

- predykatów,
- operatorów.

Stosowana jest notacja:

- infiksowa:  $3*4$ , 3 pomnoz 4,
- prefiksowa:  $*\ 3\ 4$ ,  $*(3,4)$ , pomnoz(3,4).

Operatorom przypisuje się różne priorytety wykonywania oraz określa się, czy są one lewo czy prawostronnie łączne. W ten sposób można zaprogramować algebrę, rachunek lambda, gramatyki bezkontekstowe etc.

## Funktor = oznacza **unifikację** termów

- dwa jednakowe termy można zawsze zunifikować

```

1      A=A .      /* yes */
2      a=a .      /* yes */
3      a=b .      /* no */

```

- zmienną nieprzypisaną można zunifikować z innym termem przypisując jej typ i wartość tego termu

```

1      A=B .      /* yes: B zmienna przypisana */
2      A=b .      /* yes */

```

- unifikacja dwóch nieprzypisanych zmiennych jest zawsze możliwa

```

1      A=B .      /* yes: A, B nieprzypisane */

```

- zarówno atomy, jak i liczby, mogą być łatwo porównywane według standardowych reguł

## UWAGA

- unifikacja nie jest przypisaniem ale...
- ... przypisanie wartości może być „efektem ubocznym” stosowanej unifikacji
- wynikiem unifikacji jest zawsze `true` albo `false`

## Przykłady:

```
1 data(1,kwie,2014) = data(1,kwie,2014)
2 /* termy zunifikowane */
```

```
4 data(1,kwie,2014) = data(D,kwie,2014)
5 /* termy zunifikowane jeśli:
6 - D można przypisać wartość 1 */
```

```
8 data(Day1,kwie,2014) = data(D2,kwie,2014)
9 /* termy zunifikowane jeśli:
10 - Day1 i D2 mają tę samą wartość;
11 - jedna z nich jest nieprzypisana
12   i można ją zunifikować z drugą;
13 - obie są nieprzypisane (i takie zostaną,
14   ale będą związane ze sobą) */
```

## Przykłady:

```
1 | ?- A=B.  
2 B = A  
3 yes  
4  
5 | ?- A=2, A=B.  
6 A = 2  
7 B = 2  
8 (1 ms) yes
```

Przykład nieco dłuższy:

```

1      | ?-    b(X,a)=b(f(Y),Y),
2          d(f(f(a)))=d(U),
3          c(X)=c(f(Z)).
4
5      U = f(f(a))
6      X = f(a)
7      Y = a
8      Z = a
9
10     yes

```



## Unifikacje i porównywanie termów ( $\rightarrow$ manual 8.2, 8.3)

<code>=</code>	true, gdy unifikacja możliwa
<code>\=</code>	true, gdy unifikacja niemożliwa
<code>==</code>	true, gdy porównanie udane
<code>\==</code>	true, gdy porównanie nieudane
<code>@&lt;</code>	true, gdy term mniejszy
<code>@=&lt;</code>	true, gdy term niewiększy
<code>@&gt;</code>	true, gdy term większy
<code>@&gt;=</code>	true, gdy term niemniejszy

- porównanie ma sens dla termów tego samego typu
- używany schemat leksykograficzny
- wynikiem jest `true` lub `false`

## Przykłady porównań:

```
1 | ?- X="Reksio", X=="Reksio".
```

```
2 X = [82,101,107,115,105,111]
```

```
3 yes
```

```
4  
5 | ?- X=="Reksio".
```

```
6 no
```

```
1 | ?- "Reksio" @> "reksio".
```

```
2 no
```

```
3  
4 | ?- "reksio" @> "Reksio".
```

```
5 yes
```

## Porównania numeryczne i leksykograficzne:

```
1 | ?- X=55, Y=6, X<Y.
```

```
2 no
```

```
3  
4 | ?- X="55", Y="6", X<Y.
```

```
5 uncaught exception:
```

```
6 error(type_error(evaluable, '.'/2), (<)/2)
```

```
7  
8 | ?- X="55", Y="6", X@<Y.
```

```
9 X = [53,53]
```

```
10 Y = [54]
```

```
11 yes
```

Typy termów ( $\rightarrow$  manual 6.3.4, 8.1)

<code>var(T)</code>	true, gdy T jest zmienną
<code>nonvar(T)</code>	true, gdy T nie jest zmienną
<code>atom(T)</code>	true, gdy T jest atomem (stałą lub napisem)
<code>atomic(T)</code>	true, gdy T jest atomem lub liczbą
<code>number(T)</code>	true, gdy T jest liczbą
<code>compound(T)</code>	true, gdy T jest złożony

Atom w Prologu to stała lub łańcuch znakowy.

```
1  var(A).           /* yes */
2  var(a).           /* no  */
3  var(2).           /* no  */
4  var('tekst').     /* no  */
5  atom(A).          /* no  */
6  atom(a).          /* yes */
7  atom(2).          /* no  */
8  atom('tekst').    /* yes */
```

## Arytmetyka (→ manual 8.6)

<code>:=</code>	równe
<code>=\=</code>	nierówne
<code>&lt;</code>	mniejsze
<code>=&lt;</code>	niewiększe
<code>&gt;</code>	większe
<code>&gt;=</code>	niemniejsze

<code>+</code>	plus
<code>-</code>	minus
<code>*</code>	razy
<code>/</code>	podzielić
<code>//</code>	cz. całk. z dzielenia
<code>mod</code>	reszta z dzielenia

Infiksowy funktor `is` podstawia pod swój lewy argument wyrażenie arytmetyczne (lub jego wynik) stojące po jego prawej stronie.

```
1  srednia(A,B,S) :- S is (A+B)/2.  
2  1 is mod(7,2)      /* true */  
3  3 is 7//2          /* true */  
4  3.5 is 7/2         /* true */
```

## Inne operacje matematyczne wbudowane w GNU Prologu

- potęgowanie, pierwiastek kwadratowy
- bitowe operacje and, or, xor, not
- wartość bezwzględna, znak
- funkcje trygonometryczne, odwrotne trygonometryczne i hiperboliczne
- logarytmy

Przykład: 3 duże pizze (32cm) vs. 2 pizze XXL (40cm). Czego jest więcej?

```
1 area(R,A) :- A is (pi * R^2).
2
3 | ?- area(16,A), C is 3*A, area(20,B), D is 2*B.
4
5 A = 804.24771931898704
6 B = 1256.6370614359173
7 C = 2412.7431579569611
8 D = 2513.2741228718346
9
10 yes
```

Liczba  $\pi$  otrzymana z odwrotnej funkcji trygonometrycznej:

```
1 | ?- X is 4.0*atan(1.0).
```

```
2  
3 X = 3.1415926535897931
```

```
4  
5 yes
```

Wynik jest „dokładny”, więc użycie wbudowanej liczby `pi` daje to samo:

```
1 | ?- X is 4.0*atan(1.0) - pi.
```

```
2  
3 X = 0.0
```

```
4  
5 yes
```

## Definicje rekurencyjne

Analiza, jak Prolog szuka rozwiązań (przykład wzięty z notatek autorstwa P. Fulmańskiego, Uniwersytet Łódzki, 2009).

Rozważmy kod:

```
1  mniej(p1,p2).  
2  mniej(p2,p3).  
3  mniej(p3,p4).  
4  mniej(X,Y) :- mniej(X,Z), mniej(Z,Y).
```

który powinien, na pierwszy rzut oka, wygenerować wszystkie pary z ciągu  $p1 < p2 < p3 < p4$ .



Tymczasem...

```

1  |  ?-  mniej(A,B).      |      A = p1
2                          |      B = p3 ? ;
3  A = p1                  |
4  B = p2 ? ;              |      A = p1
5                          |      B = p4 ? ;
6  A = p2                  |
7  B = p3 ? ;              |      Fatal Error: local stack
8                          |      overflow (size: 16384 Kb,
9  A = p3                  |      reached: 16384 Kb,
10 B = p4 ? ;              |      environment variable
11                          |      used: LOCALSZ)

```

program wpada w nieskończoną pętlę, zużywa dostępną mu pamięć i kończy pracę generując kod błędu.

3 pierwsze odpowiedzi  $(p1,p2)$ ,  $(p2,p3)$ ,  $(p3,p4)$ , pochodzą bezpośrednio z bazy wiedzy. Kolejne generowane są regułą

$$\text{mniej}(X,Y) \text{ :- } \text{mniej}(X,Z), \text{mniej}(Z,Y).$$

- jeśli  $X=p1$  wtedy  $\text{mniej}(X,Z)$  wskazuje na  $Z=p2$  i w efekcie  $Y=p3$ , co tworzy rozwiązanie  $(p1,p3)$
- mając  $X=p1$ ,  $Z=p2$ , system szuka dalszych rozwiązań dla wyrażenia  $\text{mniej}(p2,Y)$ . Posługuje się regułą tworząc  $\text{mniej}(p2,Y) \text{ :- } \text{mniej}(p2,ZZ), \text{mniej}(ZZ,Y)$ , która prowadzi do  $ZZ=p3$ ,  $Y=p4$  i rozwiązania  $(p1,p4)$
- poszukiwane jest kolejne rozwiązanie dla  $\text{mniej}(ZZ,Y) = \text{mniej}(p3,Y)$ ; korzystając z reguły  $\text{mniej}(p3,Y) \text{ :- } \text{mniej}(p3,Z3), \text{mniej}(Z3,Y)$  znaleźć można tylko  $Z3=p4$ , które spełnia  $\text{mniej}(p3,Z3)$ ; żadna reguła nie spełnia jednak  $\text{mniej}(p4,Y)$ , więc rozwijana jest ona  $\text{mniej}(p4,Y) \text{ :- } \text{mniej}(p4,Z4), \text{mniej}(Z4,Y)$ ; czego nie spełnia żadna reguła, ale można ją rozwinąć:  $\text{mniej}(p4,Z4) \text{ :- } \text{mniej}(p4,Z5), \text{mniej}(Z5,Z4)$  i tak dalej...

Problem leży w wywołaniu reguły `mniej(X,Y)`, która korzysta sama z siebie bez żadnej kontroli. Prowadzi to do niekończącej się pętli. Definicje rekurencyjne wymagają starannego zaprojektowania – w szczególności ważny jest warunek zakończenia rekurencji.

Dwa podejścia do wnioskowania rekurencyjnego:

- **top-down** – problem jest rozkładany na prostsze podproblemy tak długo, aż otrzyma się fakty z bazy wiedzy
- **bottom-up** – zaczynając od bazy wiedzy i znanych reguł próbuje się zbudować zapytanie; często potrzebne są zmienne pomocnicze (akumulatory); jest to podejście zazwyczaj szybsze niż top-down

## Przykład

### Ciąg Fibonacciego

0, 1,  $\rightarrow$  1, 2, 3, 5, 8, 13, 21, 34, ..., 6765 (20. miejsce) , ...

tworzony jest z dwóch pierwszych elementów 0, 1 (czasami 1, 1) za pomocą wzoru rekurencyjnego

$$f_n = f_{n-1} + f_{n-2}, \quad n > 2$$

## Definicja top-down

```

1 fib(N,0) :- N == 0.
2 fib(N,1) :- N == 1 ; N == 2.
3 fib(N,X) :- N1 is N-1, fib(N1,Y),
4             N2 is N-2, fib(N2,Z), X is Y+Z.

```

- zaczynamy od N i „idziemy” **w dół** (N1 is N-1, N2 is N-2) dopóki N=0
- początek definicji to punkty startowe ciągu, pełniące również rolę punktu zakończenia rekurencji
- sama rekurencja jest na końcu definicji
- wynik (X is Y+Z) liczony jest po zakończeniu rekurencji

Po uproszczeniu tej definicji

```

1  fib(N,0) :- N == 0.
2  fib(N,1) :- N == 1 ; N == 2.
3  fib(N,X) :- N1 is N-1, fib(N1,Y),
4              N2 is N-2, fib(N2,Z), X is Y+Z.

```

można otrzymać równoważną definicję top-down

```

1  fib(0,0).
2  fib(1,1).
3  fib(N,X) :- fib(N-1,Y), fib(N-2,Z), X is Y+Z.

```

## Definicja bottom-up

```

1  fibonacci(N,X) :- fib(0,0,1,N,X).
2
3  fib(N,X,_,N,X).
4  fib(N1,X1,X2,N,X) :- N1<N, N2 is N1+1,
5                        X3 is X1+X2,
6                        fib(N2,X2,X3,N,X).

```

- linia 1: ukrycie zmiennych wewnętrznych
- linia 3: warunek zakończenia; \_ to zmienna anonimowa
- linie 4-6: N, N1, N2 – numerują elementy ciągu
- linie 4-6: X1, X2, X3 – elementy ciągu
- N1, X1, X2 – akumulatory
- wynik pośredni (X3 is X1+X2) jest otrzymywany zanim następuje wywołanie fib

## Uwagi:

- definicje top-down są prostsze i krótsze
- definicje bottom-up są bardziej złożone; często wymagają użycia akumulatorów
- wersja top-down wywołuje `fib` dwukrotnie podczas każdego kroku, więc liczba operacji przy  $N$  krokach skaluje się jak  $2^N$
- wersja bottom-up wywołuje `fib` jeden raz podczas każdego kroku, więc liczba operacji przy  $N$  krokach skaluje się jak  $N$



## Przykład

Funkcja silnia zdefiniowana jest dla liczb naturalnych  $n$  jako

$$n! = 1 \cdot 2 \cdot \dots \cdot n$$

### Definicja top-down

```

1 fact(0,1).
2 fact(N,F):- N>0, N1 is N-1,
3             fact(N1,F1), F is N*F1.
```

### Definicja bottom-up

```

1 fact(N,F) :- fact(0,1,N,F).
2
3 fact(N,F,N,F).
4 fact(N1,F1,N,F):- N1<N, N2 is N1+1,
5                   F2 is N2*F1, fact(N2,F2,N,F).
```

Podsumowując:

### **top-down**

- wnioskowanie wsteczne
- zaczyna od problemu (A)
- używa odwrotnej postaci reguł aby rozłożyć A na jakąś kombinację znanych faktów
- jest powolne, gdyż sprawdza również warianty nie prowadzące do poprawnego rozwiązania

Przykład: jeśli reguła ( $A :- B.$ ) istnieje, aby udowodnić A należy udowodnić B

### **bottom-up**

- wnioskowanie naprzód
- zaczyna od znanych faktów
- używa reguł aby zbudować zapytanie (B)
- jest szybsze, chociaż może tworzyć wiele pośrednich prawdziwych stwierdzeń, które nie są powiązane bezpośrednio z pytaniem

Przykład: A jest prawdziwe, istnieje reguła ( $A :- B.$ ), wynika stąd, że B jest prawdziwe

## Przykład programu w Prologu

Konwersja między stopniami Celsjusza i Fahrenheita.

```

1  run :- write('Podaj temp. w st. Celsjusza '),
2         read(C), convert(C,F),
3         write('Temp. to '), write(F),
4         write('st. Fahrenheita'), nl,
5         warning(F,Komunikat), write(Komunikat).
6
7  convert(C,F) :- F is 9./5.*C + 32.
8
9  warning(T,'Ale upal!') :- T > 90.
10 warning(T,'Ale ziab!') :- T < 30.
11 warning(T,'') :- T >= 30, T <= 90.

```

Źródło:

[mcs.wartburg.edu/zelle/cs373/handouts/PrologExamples/PrologIntro.pdf](http://mcs.wartburg.edu/zelle/cs373/handouts/PrologExamples/PrologIntro.pdf)

## Listy

Prolog rozpoznaje dwa sposoby deklarowania **list**:

- zapis `[a,b,c]` jest przyjazny dla użytkownika
- funktor prefiksowy `.` (kropka) dodaje term na początek listy

Zapis w nawiasach kwadratowych jest wewnętrznie konwertowany do równoważnego zapisu kropkowego. Lista pusta to `[]`.

```
1 | ?- L = [a,b,c], M = .(a,.(b,.(c,[],))), L=M.
```

```
3 L = [a,b,c]
```

```
4 M = [a,b,c]
```

```
6 yes
```

```
/* L można zuniifikować z M */
```

Lista jest automatycznie dzielona na **głowę** (pierwszy element) i **ogon** (całą resztę). Zapisywane jest to za pomocą operatora `|` (pionowa kreska) `[H|T]`.

```
1 | ?- L = [a,b,c], L = [H|T].
```

```
2  
3 H = a
```

```
4 L = [a,b,c]
```

```
5 T = [b,c]
```

```
6  
7 yes
```

Poniższe zapisy dotyczą tej samej listy:

```
1 [a,b,c] = [a|[b,c]] = [a,b|[c]] = [a,b,c|[]]
```

Przykłady:

1. lista pusta nie ma głowy ani ogona:

```
1      ?-  [] = [H | T] .
2      No
```

2. dla listy jednoelementowej ogon jest listą pustą:

```
1      ?-  [1] = [H | T] .
2      H = 1
3      T = []
```

3. głowa to pojedynczy term, ogon jest **zawsze** listą:

```
1      ?-  [1, 2] = [H | T] .
2      H = 1
3      T = [2]
```

## Inne przykłady:

```
1 ?- [1,[2,3]]=[H|T].
```

```
2 H = 1
```

```
3 T = [[2,3]]      /* ogon to lista list */
```

```
4
```

```
5 ?- [[1,2],3]=[H|T].
```

```
6 H = [1,2]      /* głowa jest listą */
```

```
7 T = [3]
```

```
8
```

```
9 ?- [1,2,3,4]=[Ha,Hb|T].
```

```
10 Ha = 1
```

```
11 Hb = 2
```

```
12 T = [3,4]
```

```
13
```

```
14 ?- [[1,2,3],4]=[[H1|T1]|T2].
```

```
15 H1 = 1
```

```
16 T1 = [2,3]
```

```
17 T2 = [4]
```

Listy nie są ograniczone do elementów jednego typu

```
1 | ?- L = [[X,a],bolek,[],['Celina',[]]],
2 |     L = [H|T].
```

```
3
4 H = [X,a]
```

```
5 L = [[X,a],bolek,[],['Celina',[]]]
```

```
6 T = [bolek,[],['Celina',[]]]
```

```
1 | ?- L = [A,B|C], L = [ala, boleka, [1,2,3]].
```

```
2
3 A = ala
```

```
4 B = boleka
```

```
5 C = [[1,2,3]]
```

```
6 L = [ala, boleka, [1,2,3]]
```

→ zauważcie, że C nie jest równe [1,2,3]!



**Przykład:** `append(L1,L2,M)` sprawdza, czy  $M=L1+L2$

```
1 | ?- L1=[a,b,c], L2=[d,e,f], append(L1,L2,M)
```

```
3 L1 = [a,b,c]
```

```
4 L2 = [d,e,f]
```

```
5 M = [a,b,c,d,e,f]
```

```
7 yes
```

Schemat działania scalania list `append(L1,L2,M)`:

- rozłóż jedną z list ( $L1$  lub  $L2$ ) na osobne elementy
- dodawaj po jednym elemencie rozłożonej listy na początek lub koniec drugiej listy
- zunifikuj wynik z ostatnim argumentem wywołania ( $M$ )

**Problem:** Zdefiniuj `concat/3`, aby działało jak `append/3`.

**Strategia 1** przy wywołaniu `concat1(L1,L2,M)`

- usuń pierwszy element z L2
- dopisz go na końcu L1
- powtarzaj aż L2=[]
- zunifikuj wynik z M

```

1 concat1(L, [], L).
2 concat1(L1, L2, M) :- L2=[E|R], Z=[L1|E],
3                       concat1(Z, R, M).
4
5 | ?- L1=[a,b,c], L2=[d,e,f], concat1(L1, L2, M).
6 L1 = [a,b,c]
7 L2 = [d,e,f]
8 M = [[[[a,b,c]|d]|e]|f] ? ;
9 no

```

**Strategia 2** przy wywołaniu `concat2(L1,L2,M)`

- usunąć ostatni element z L1
- dopisać go do początku L2
- powtarzać aż L1=[]
- zunifikować wynik z M

```

1  concat2([],L,L).
2  concat2(L1,L2,M) :- L1=[P|K], concat2(K,L2,Z),
3                        M=[P|Z].
4
5  | ?- L1=[a,b,c], L2=[d,e,f], concat2(L1,L2,M).
6  L1 = [a,b,c]
7  L2 = [d,e,f]
8  M = [a,b,c,d,e,f] ? ;
9  no

```

## Strategia 1

- rekurencja niby prostsza
- wynik wymaga „spłaszczenia” do pojedynczej listy

## Strategia 2

- rekurencja mniej przejrzysta, bo wymaga rozłożenia całej listy i pracy „od końca”
- wynik jest poprawny
- jest to standardowa konstrukcja w Prologu (podejście bottom-up)

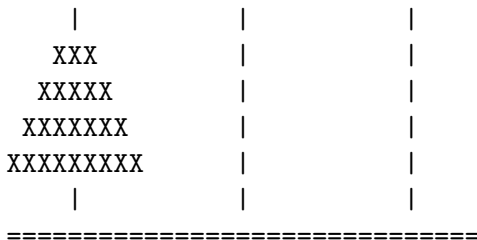
## Praca domowa

Zastanówcie się, dlaczego strategia top-down w tym przypadku zawodzi? Czy da się to poprawić (może jest błąd w mojej definicji)? W czym tkwi różnica pomiędzy dwoma, zdawałoby się logicznie równoważnymi, podejściami? A może Prolog jest „niesymetryczny” i te dwa podejścia nie są dla niego równoważne z powodu którejś z użytych konstrukcji?

**Przykład:** *Pięciu przyjaciół brało udział w wyścigu. Willy nie wygrał. Gregory przybiegł trzeci, za Danielem. Daniel nie zajął drugiego miejsca. Andrew nie wygrał ani nie był ostatni. Bill przybiegł zaraz za Willym. Ustal kolejność zawodników.*

```
1 place(W,G,D,A,B) :- L0=[1,2,3,4,5], G=3,
2                       select(W,L0,L1), W\=1,
3                       select(D,L1,L2), D\=2,
4                       select(A,L2,L3), A\=1, A\=5,
5                       select(B,L3,[G]), B is W+1,
6                       G>D.
7 | ?- place(Willy,Gregory,Daniel,Andrew,Bill).
8 Andrew = 2
9 Bill = 5
10 Daniel = 1
11 Gregory = 3
12 Willy = 4 ?
13 yes
```

## Przykład: wieże Hanoi



## Zasady gry

Przełożyć krążki na inny pręt:

- tylko jeden krążek może być przekładany na raz
- tylko szczytowy krążek może być przekładany
- nie można położyć krążka na mniejszym krążku

Najmniejsza liczba ruchów potrzebna do rozwiązania układanki to  $2^n - 1$ , gdzie  $n$  jest liczbą krążków.

## Strategia: [Clocksin, Mellish, rozdz. 7.4]

- Trzy pręty to: start (source), meta (destination) i jeden pomocniczy (temp).

Rozwiązanie jest rekurencyjne. Dla  $n$  krążków:

- ➊ przesun  $n - 1$  krążków ze startu na pomocniczy
  - ➋ przesun  $n$ -ty (największy) dysk ze startu na metę
  - ➌ przesun  $n - 1$  dysków z pomocniczego na metę
- aby wykonać (1) trzeba przesunąć  $n - 2$  krążków ze startu na (inny) pomocniczy itd., więc procedura jest rekurencyjna
  - $n$  musi być podane na wejściu do programu

Program rozwiązujący wieże Hanoi:

```
1 hanoi(N) :- move(N, left, center, right).  
2  
3 move(0, _, _, _) :- !.  
4 move(N, A, B, C) :- M is N-1,  
5                     move(M, A, C, B),  
6                     info(A, B),  
7                     move(M, C, B, A).  
8  
9 info(X, Y) :- write([X, -->, Y]), nl.
```

gdzie lewy pręt to start, środkowy meta, prawy pomocniczy.

Uwaga:

- w programie nie ma algorytmu rozwiązującego układankę,
- opisana jest reguła poprawnego przełożenia krążka między prętami,
- **Prolog sam znajduje rozwiązanie na podstawie punktu startowego i podanych reguł.**

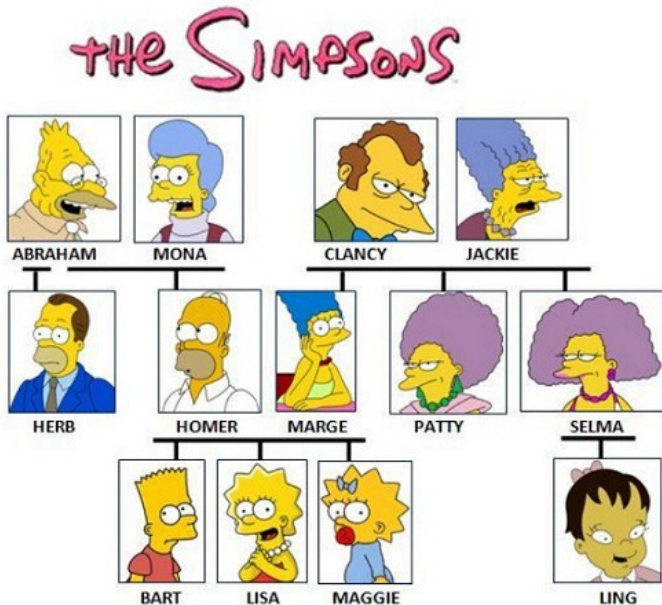


```
1 | ?- hanoi(3).  
2 [left,-->,center]  
3 [left,-->,right]  
4 [center,-->,right]  
5 [left,-->,center]  
6 [right,-->,left]  
7 [right,-->,center]  
8 [left,-->,center]
```

Dla 3 dysków potrzeba  $2^3 - 1 = 7$  ruchów, dla 4 dysków wymagane jest  $2^4 - 1 = 15$  ruchów.

```
1 | ?- hanoi(4).  
2 [left,-->,right]  
3 [left,-->,center]  
4 [right,-->,center]  
5 [left,-->,right]  
6 [center,-->,left]  
7 [center,-->,right]  
8 [left,-->,right]  
9 [left,-->,center]  
10 [right,-->,center]  
11 [right,-->,left]  
12 [center,-->,left]  
13 [right,-->,center]  
14 [left,-->,right]  
15 [left,-->,center]  
16 [right,-->,center]
```

## Przykład: drzewo genealogiczne



Baza opisująca takie drzewo może mieć postać:

```
1  male(abraham).  
2  male(clancy).  
3      ...  
4  female(mona).  
5  female(jackie).  
6      ...  
7  parents(abraham,mona,herb).  
8  parents(abraham,mona,homer).  
9  parents(clancy,jackie,marge).  
10     ...
```

Możliwe są też inne wersje, ale żeby zakodować zależności rodzinne, informacje male/1, female/1 i parents/3 są wystarczające.

Kilka przykładowych predykatów.

X jest siostrą Y:

```
1 sister(X,Y) :- parents(F,M,X), parents(F,M,Y),  
2               female(X), X\=Y.
```

X jest matką Y:

```
1 mother(X,Y) :- parents(_,X,Y).
```

X jest babcią Y:

```
1 gmother(X,Y) :- mother(X,Z),  
2                  (parents(_,Z,Y); parents(Z,_,Y)).
```

## Opis gramatyk w Prologu

Początkową motywacją A.Colmerauera do napisania Prologu była chęć maszynowej analizy języków naturalnych (*computational linguistics*). Prolog pozwala w łatwy sposób posługiwać się gramatykami, m.in. bezkontekstowymi.

Analizowany ciąg ma postać listy wyrazów. Gramatyka to zestaw reguł pozwalających na składanie wyrażeń w ciągi. Reguły muszą być definiowane rekurencyjnie, gdyż najczęściej długość ciągu nie jest ustalona.

**Przykład:** fragment gramatyki języka angielskiego  $G = (V, T, P, S)$ . Zmienne  $V = \{np, vp, n, v, det\}$ , gdzie  $np$  to część rzeczownikowa zdania,  $n$  to rzeczownik,  $vp$  to część czasownikowa zdania,  $v$  to czasownik,  $det$  to określnik. Terminalami są  $T = \{cat, dog, eats, the\}$ . Produkcje opisane są przez:

$$S \rightarrow np\ vp$$

$$np \rightarrow det\ n$$

$$vp \rightarrow v \mid v\ np$$

$$n \rightarrow cat \mid dog$$

$$v \rightarrow eats$$

$$det \rightarrow the$$

Kod w Prologu opisujący tę gramatykę ma postać

```

1 | ?- listing.
2 | % file: /home/marek/gbk.pro
3
4 | s(A, B) :- np(A, C), vp(C, B).
5 | np(A, B) :- det(A, C), n(C, B).
6 | vp(A, B) :- v(A, B).
7 | vp(A, B) :- v(A, C), np(C, B).
8 | n([cat|A], A).
9 | n([dog|A], A).
10 | v([eats|A], A).
11 | det([the|A], A).

```

Pozwala on na budowanie listy składającej się z odpowiednich wyrażeń w odpowiedniej kolejności.

Poprzedni zapis wymaga dużej uważności i nie jest łatwy do czytania. W Prologu istnieje skrótowy równoważny zapis produkcji gramatyki z wykorzystaniem operatora `-->`

```
1  s --> np, vp.  
2  np --> det, n.  
3  vp --> v.  
4  vp --> v, np.  
5  n --> [cat].  
6  n --> [dog].  
7  v --> [eats].  
8  det --> [the].
```

Pamiętać jedynie trzeba, żeby symbole terminalne wpisywać jako jednoelementowe listy.



Predykat `phrase(symbol,fraza)` pozwala na sprawdzenie, czy z symbolu startowego gramatyki można zbudować frazę.

Zdanie poprawne zwraca wartość `true`

```

1 | ?- phrase(s,[the,dog,eats]).
2
3 true ? ;
4 no

```

Zdanie nie należące do gramatyki zwraca wartość `false`

```

1 | ?- phrase(s,[the,cat]).
2
3 no

```

Aby otrzymać wszystkie możliwe zdania naszej gramatyki należy frazę zastąpić zmienną.

```

1  |  ?- phrase(s,X).
2
3  X = [the,cat,eats] ? a
4  X = [the,cat,eats,the,cat] ?
5  X = [the,cat,eats,the,dog] ?
6  X = [the,dog,eats] ?
7  X = [the,dog,eats,the,cat] ?
8  X = [the,dog,eats,the,dog]
9
10 yes

```

Uwaga: ta gramatyka zawiera tylko sześć różnych poprawnych zdań. Trudniej jest w przypadku gramatyk o nieskończonej liczbie poprawnych konstrukcji.

**Przykład:** gramatyka wyrażeń algebraicznych (cztery działania, trzy zmienne). Ta gramatyka opisuje nieskończenie wiele poprawnych wyrażeń, gdyż nie ma ustalonej maksymalnej długości ciągu znaków.

$$S \rightarrow v \mid S \circ S$$

$$v \rightarrow x \mid y \mid z$$

$$\circ \rightarrow + \mid - \mid * \mid /$$

Uwaga: pierwsza reguła z  $S$  pozwala stworzyć nowe  $S$ , co w rekurencji prowadzi do poprawnych wyrażeń o dowolnie dużej długości.

## Gramatyka zapisana regułami Prologu:

```
1  s --> v .
2  s --> s , o , s .
3  v --> [x] .
4  v --> [y] .
5  v --> [z] .
6  o --> [+] .
7  o --> [-] .
8  o --> [*] .
9  o --> [/] .
```

Poprawne wyrażenia generują się w nieskończoność:

```

1  |  ?- phrase(s,X) .
2
3  X = [x] ? ;
4  X = [y] ? ;
5  X = [z] ? ;
6  X = [x,+,x] ? ;
7  X = [x,+,y] ? ;
8  X = [x,+,z] ? ;
9  X = [x,+,x,+,x] ? ;
10 X = [x,+,x,+,y] ? ;
11 X = [x,+,x,+,z] ?
12 (...)
```

Kod nie posiada zabezpieczenia przed nieskończoną pętlą sprawdzającą pozostałe możliwe rozwiązania oraz weryfikującą ciągi nienależące do gramatyki.

```
1 | ?- phrase(s,[x/,z-,y]).
```

```
2  
3 true ? ;
```

```
4  
5 Fatal Error: local stack overflow  
6 (size: 16384 Kb, reached: 16384 Kb,  
7 environment variable used: LOCALSZ)
```

# PODSUMOWANIE

## Główne paradygmaty programowania, bez podpodziału:

Action	Imperative	Nondeterministic
Agent-oriented	--> Procedural	Parallel computing
Array-oriented	--> Object-oriented	Process-oriented
Automata-based	Literate	Probabilistic
Concurrent computing	Language-oriented	Stack-based
Data-driven	Natural-language prog.	Structured
Declarative	Discipline-specific	Block-structured
--> Functional	Domain-specific	Modular
--> Logic	Grammar-oriented	Object-oriented
Constraint	Intentional	Recursive
Dataflow	Metaprogramming	Value-level
Dynamic/scripting	Automatic	Quantum programming
Event-driven	Reflective	
Service-oriented	Homoiconic	
Time-driven	Macro	
Function-level	Template	
Point-free style	Non-structured	
Generic	Array	

[Wikipedia: *Comparison of programming paradigms*]



## Uwagi:

- Cztery wiodące paradygmaty: czysty imperatywny ( $\rightarrow$  Fortran), obiektowy ( $\rightarrow$  Ruby), deklaratywny funkcyjny ( $\rightarrow$  Haskell), deklaratywny w logice ( $\rightarrow$  Prolog).
- Nowoczesne języki są często hybrydowe, łącząc wiele różnych paradygmatów.
- Obliczenia rozproszone: wielowątkowość, wieloprocessorowość, klastry, sieci.
- Komputery stochastyczne.
- Komputery kwantowe.

## Obliczenia

- prowadzone na wielu jednostkach obliczeniowych,
- mogą być **równoległe** lub **rozproszone**.

Wykorzystuje się maszyny z wielordzeniowymi CPU, maszyny z wieloma CPU, klastry maszyn, równoległe potoki obliczeniowe (GPU), sieci rozproszone (np. SETI, blockchain).

Zrównoleglenie może być

- dokonane automatycznie przez system lub kompilator (*implicit*),
- ustawione ręcznie przez programistę (*explicit*).

Problemy w takich systemach dotyczą m.in.

- współdzielenia zasobów,
- synchronizacji obliczeń na różnych jednostkach,
- komunikacji pomiędzy jednostkami.

## Trzy największe klastry na świecie (czerwiec 2024) [www.top500.org]

Miejsce 3: Eagle - Microsoft NDv5

Gdzie: Microsoft Azure USA

CPU: Intel Xeon Platinum 8480C 48C 2.0GHz

Liczba rdzeni: 2 073 600

Moc max: 561,20 PFlop/s

Moc szczytowa: 846,84 PFlop/s

Zasilanie: ?

Miejsce 2: Aurora - HPE Cray EX

Gdzie: DOE/SC/Argonne National Laboratory USA

CPU: Intel Xeon Max 9470 52C 2.4GHz

Liczba rdzeni: 9 264 128

Moc max: 1,012 ExaFlop/s

Moc szczytowa: 1,980 ExaFlop/s

Zasilanie: 38 698 kW

Miejsce 1: Frontier - HPE Cray EX235a

Gdzie: DOE/SC/Oak Ridge National Laboratory USA

CPU: AMD Optimized 3rd Generation EPYC 64C 2GHz

Liczba rdzeni: 8 699 904

Moc max: 1,206 ExaFlop/s

Moc szczytowa: 1,714 ExaFlop/s

Zasilanie: 22 786 kW

## Algorytmy i komputery **stochastyczne**:

- część danych obarczona jest niepewnością i ich wartości opisane są rozkładami prawdopodobieństwa wokół pewnych wartości średnich,
- algorytmy muszą uwzględniać te rozkłady,
- algorytmy zwracają wyniki najczęściej w postaci wartości najbardziej prawdopodobnej, ewentualnie przedziału o określonym poziomie ufności.

Podejście statystyczne z uwzględnieniem poziomu wiarygodności danych ma zastosowanie w modelowaniu systemów o dużym stopniu złożoności takich jak np.

- plan produkcji w fabryce,
- modele i planowanie w makroekonomii,
- wybór kandydatów w konkursie na jakieś stanowisko,
- zarządzanie ruchem samochodowym w mieście,
- przewidywanie poziomu wody w zbiorniku wodnym i inne.

## Algorytmy i komputery **kwantowe**:

- bity kwantowe (kubity) mają postać  $a \cdot \mathbf{0} + b \cdot \mathbf{1}$ , tak więc mogą reprezentować jednocześnie bit zero i bit jeden w dowolnych proporcjach,
- pomiar kubitu zawsze zwróci zero lub jeden, niszcząc stan kwantowy sprzed pomiaru,
- sprytny algorytm pozwala na przyspieszenie obliczeń poprzez wykonywanie ich na odpowiednio przygotowanych kubitach.

Przykładem jest algorytm faktoryzacji Shora, który pozwala na wydajny rozkład dowolnie dużych liczb na czynniki pierwsze (tym samym czyniąc algorytm RSA bezużytecznym).

## Automat:

- skończony lub nieskończony  
→ liczba stanów,
- deterministyczny lub niedeterministyczny  
→ przejścia do jednego lub wielu stanów,
- bez lub z przejściami  $\varepsilon$   
→ przejścia bez czytania kolejnego symbolu z wejścia,
- bez lub ze stosem  
→ stan automatu + stan stosu.

Automat jest modelem układu analizującego wejściowy ciąg danych i dopasowującego go do zadanego wzorca. Wynikiem działania automatu jest Tak lub Nie.

Definicja zawiera: alfabet, stany, reguły przejścia.

Równoważne są sobie:

- DAS
- NAS
- $\varepsilon$ -NAS
- RE
- AZS
- GBK

Równoważność sprawdza się na poziomie języków.

Równoważność oznacza istnienie reguł pozwalających przekształcić jeden typ automatu na inny typ automatu.



## Przekształcenia automatów:

- DAS  $\rightarrow$  NAS: każdy DAS jest NAS
- NAS  $\rightarrow$  DAS: metoda konstrukcji podzbiorów
- NAS  $\rightarrow$   $\varepsilon$ -NAS: każdy NAS jest  $\varepsilon$ -NAS
- $\varepsilon$ -NAS  $\rightarrow$  NAS: eliminacja przejść  $\varepsilon$  poprzez rozgałęzienie grafu
- $\varepsilon$ -NAS  $\rightarrow$  RE: budujemy RE opisujące język automatu

$$R_{ij}^{(n)} = R_{ij}^{(n-1)} + R_{in}^{(n-1)} \left( R_{nn}^{(n-1)} \right)^* R_{nj}^{(n-1)}$$

- DAS  $\rightarrow$  RE: budujemy RE metodą eliminacji stanów
- RE  $\rightarrow$   $\varepsilon$ -NAS: najprostsze podwyrażenia RE znakują przejścia między stanami automatu; konkatenacja RE to ciąg stanów, suma RE to rozgałęzienie automatu, gwiazdka RE to pętla zwrotna

- GBK  $\rightarrow$  RE: produkcje gramatyki można opisać RE
- GBK  $\rightarrow$  AZS: produkcje gramatyki można opisać poprzez AZS

UWAGA: do opisu pełnej gramatyki potrzeba wielu RE/AZS i czasami dodatkowego mechanizmu wybierającego jeden z nich.

UWAGA: powyższe metody pokazują, że wszystkie omawiane struktury są sobie równoważne (choć niewygodnie byłoby mówić o gramatyce w języku automatów DAS...).

## Gramatyka bezkontekstowa.

- GBK to zestaw reguł (produkcji) poprawnego konstruowania wyrażeń w danym języku.
- Produkcje zamieniają zmienną na wyrażenie składające się ze zmiennych i symboli terminalnych.
- Po ostatnim kroku wyprowadzenia wyrażenie składa się wyłącznie z terminali.
- GBK może być reprezentowana przez zbiór automatów spiętych mechanizmem wyboru jednego z nich (co jest zbyt skomplikowane i nie jest używane).
- Może być opisana z wykorzystaniem AZS.
- Może być opisana z wykorzystaniem RE.
- Jest konieczna do zdefiniowania języka, implementacji interpretera, implementacji kompilatora.

## Uwagi:

- GBK bazuje na zbiorze produkcji, pozwalających na konstrukcję wyrażenia (wyprowadzenie) lub rozłożenie wyrażenia (wnioskowanie rekurencyjne).
- Wnioskowanie rekurencyjne, uogólnione wyprowadzenie (lewo-, prawostronne i mieszane) i konstrukcja drzewa wyprowadzenia są sobie równoważne.
- Drzewo wyprowadzenia konstruowane przez parser nazywane jest drzewem parsowania.
- Gramatyki używane są podczas analizy i kompilacji kodu.

Na działanie interpretera/kompilatora składają się m.in.:

- analiza leksykalna (słownikowa; skaner, tokenizacja kodu)  
→ przypisanie symboli do zmiennych
- analiza syntaktyczna (składniowa; parser, drzewa parsowania)  
→ wyprowadzenie wyrażeń z produkcji gramatyki języka
- analiza semantyczna (znaczeniowa, analiza błędów).

Uwaga: Gramatyki mogą być wieloznaczne. Niektóre można ujednoznaczyć, innych nie. Stosuje się dodatkowe reguły spoza gramatyki, aby poprawnie interpretować wyrażenia języka, np. priorytety operatorów, ich lewo- lub prawostronną łączność itd.

Typy błędów w kodzie programu:

- leksykalne (słownikowe) – głównie literówki w etykietach
- syntaktyczne (składniowe) – znaki specjalne, nawiasy
- semantyczne (znaczeniowe) – niezgodność typów
- logiczne – programista koduje nie to, co zamierzał
- algorytmiczne – programista używa błędnego algorytmu

Trzy pierwsze kategorie błędów może znaleźć kompilator/interpreter. Dwie ostatnie kategorie są trudne do wykrycia.

Automaty ze stosem:

- Stos w AZS pełni funkcję pamięci lub licznika.
- Akceptacja poprzez stany akceptujące.
- Akceptacja poprzez opróżnienie stosu.
- AZS są równoważne GBK.

Każdy AZS może być zdefiniowany jako automat akceptujący poprzez stany końcowe akceptujące lub jako automat akceptujący poprzez opróżnienie stosu. Istnieją schematy zamiany jednego typu w drugi.

Zmienne, typy i wiązania.

Wiązanie odbywa się na wszystkich poziomach, od projektu języka po uruchomienie programu. Może być:

- statyczne,
- dynamiczne,
- sprzętowe.

Typowanie zmiennych może być statyczne lub dynamiczne, a dodatkowo silne lub słabe. Dokonuje się:

- jawnie,
- niejawnie,
- kontekstowo.

Niektóre mechanizmy wymagają interpretacji a nie kompilacji programu.



Programowanie deklaratywne nie wymaga znajomości algorytmu w tradycyjnym znaczeniu tego słowa.

- Programowanie imperatywne: dane + algorytm.
- Programowanie deklaratywne: dane + reguły + cel.

Podstawę teoretyczną tworzą

- dla Haskell'a rachunek lambda:
  - ▶  $\alpha$ -konwersja czyli zamiana etykiet,
  - ▶  $\beta$ -redukcja czyli wyliczenie wartości funkcji.
- dla Prologu rachunek logiki:
  - ▶ algebra Boole'a,
  - ▶ klauzule Horna.

Oba podejścia dają możliwość pełnego programowania.

Programowanie deklaratywne ma problem z wykonywaniem kilku operacji w ramach jednego wywołania. Stosuje się m.in.:

- efekty uboczne działania,
- monady.

**DZIĘKUJĘ ZA UWAGĘ!  
PYTANIA?**