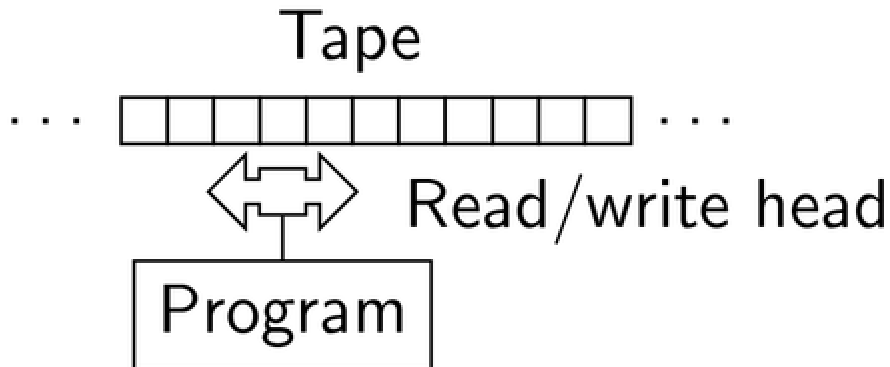
O podstawach Informatyki konkretnie:
od Maszyny Turinga do współczesnych funkcyjnych języków programowania

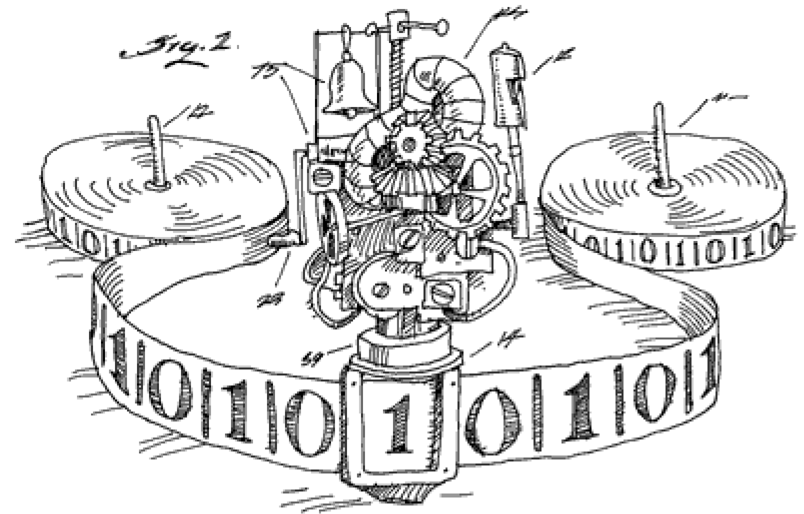
Stanisław Ambroszkiewicz

IPI PAN

18 stycznia 2016

Maszyna Turinga (1936)

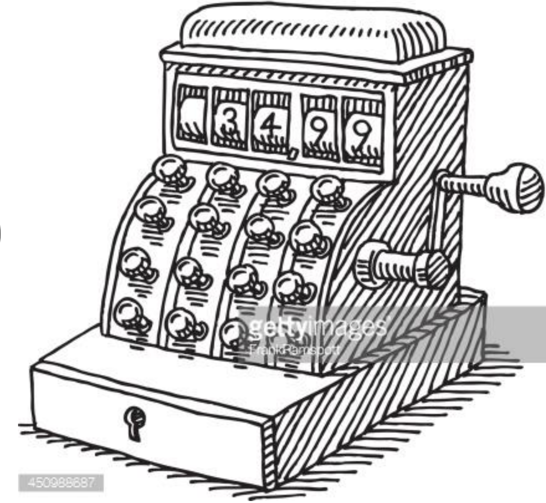




- http://www.felienne.com/archives/2974

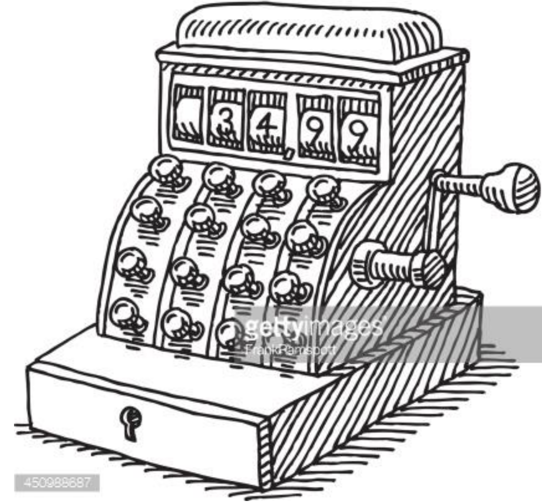
- składa się z nieskończonej taśmy podzielonej na komórki oraz głowicy, która czyta z pojedynczej komórki symbol tam zawarty (tj. 0, 1 lub blank) oraz zapisuje w komórce symbole 0 lub 1. Głowicą steruje program (funkcja przejścia), który na podstawie stanu maszyny (zbiór stanów jest skończony) oraz bieżącego odczytu głowicy wyznacza jaki symbol ma być zapisany na taśmie oraz czy głowica ma zostać przesunięta w prawo, lewo, czy pozostać na miejscu. Osiągnięcie specjalnego stanu **Halt** oznacza zatrzymanie działania i tym samym zakończenie obliczenia.
- Początkowo na taśmie jest skończony ciąg zero-jedynkowy jako input, po zakończeniu obliczeń, wynikiem obliczeń (output) jest ciąg zero-jedynkowy taśmie.
- Jeśli kto uważa, że Maszyny Turinga to dobry model obliczalności, to niech spróbuje skonstruować taką maszynę dla prostej funkcji dodającej kolejne liczby naturalne, od 1 do n.

Maszyna rejestrowa
register machine 1950/1960



- nieograniczoną liczbę rejestrów **R_n , n:N**
- **R_n** zawiera liczbę naturalną **r_n** (nieograniczony pamięciowo)
- Program **P** zawiera skończoną liczbę instrukcji na bazie 4 podstawowych instrukcji:
  1. zero **Z(n)** (zamień **r_n** na 0)
  2. następnik **S(n)** // zamień **r_n** na **$r_n + 1$**
  3. transfer **T(m; n)** // zamień **r_n** na **r_m**
  4. skok **J(m; n; q)** // jeśli **$r_m = r_n$**, to przejdź do instrukcji o numerze **q** w **P**, w przeciwnym przypadku przejdź do następnej instrukcji w **P**

Maszyna rejestrowa register machine 1950/1960



- Program realizujący funkcję **f** (zliczającą kolejne liczby naturalne) jest prosty
- wprowadźmy dodatkową instrukcję dla **plus(m,n,k)** dodawania zawartości dwóch rejestrów (**r_m, r_n**) a wynik wpisywany jest do rejestru **r_k**
- Początkowo w rejestrze **r_1** jest dana wejściowa, czyli **n**. W rejestrach **r_2, r_3**, oraz **r_4** jest **0**
- Program **P**
  1. **S(2)**
  2. **S(4)**
  3. **plus(3,2,3)**
  4. **J(1;2;6)**
  5. **J(4;2;1)** // warunek skoku zawsze spełniony
  6. **STOP** // wynik jest w **r_3**

Co to są obliczenia?

- operacje wykonywane na liczbach, a dokładniej sekwencje (warunkowe) takich operacji. Czyli algorytmy (programy) składające się z takich operacji

- Co to są operacje na liczbach? Czym są liczby?

- Liczby naturalne to abstrakcja liczenia. Typ liczb naturalnych ma swój operator, jest to pierwotna operacja następnika, obiekt pierwotny, czyli 1, oraz operację pierwotną poprzednika i pierwotne relacje równości, większości i mniejszości.
- Jeśli dodamy jeszcze parę aksjomatów, w tym schemat pierwotnej rekursji, to (plus logika predykatów) mamy Arytmetykę Peano pierwszego rzędu, jako formalną teorię.

Proste obliczenia

- Przykład. **$f: N \rightarrow N$**, symbol **N** oznacza liczby naturalne
- **$f(n) = 1+2+ \ldots +(n-1)+n$** %% to nie jest ani definicja ani konstrukcja
- **$f(n) = f(n-1)+n$** oraz **$f(1) = 1$** % to jest defiicja rekurencyjna
- obliczenie **$f(5) =$**

  $f(4) + 5 =$

  $f(3) + 4 + 5 =$

  $f(2) + 3 + 4 + 5 =$

  $f(1) + 2 + 3 + 4 + 5 =$

  $1 + 2 + 3 + 4 + 5 = 15$
- dla uproszczenia dodawanie jest pierwotne

- **$f(n) = n(n+1)/2$** %% ale to już jest twierdzenie

Funkcje pierwotnie rekurencyjne

- W Arytmetyce Peano występuje schemat rekursji do definicji nowych funkcji (z liczb naturalnych w liczby naturalne) na podstawie poprzednio zdefiniowanych oraz postulowanych (jako pierwotne): funkcji następnika, funkcji stałych, projekcji. Jest jeszcze kompozycja funkcji.
- **Schemat pierwotnej rekursji.** Jeśli funkcje **h** (k - argumentowa) oraz **g** (k+2 argumentowa) są już zdefiniowane, to nowa funkcja **f** (k+1 argumentowa) jest definiowana poprzez równości
- $x = (x_1, \dots, x_k)$
- $f(1,x) = h(x)$
- $f(n+1, x) = g(n, f(n, x), x)$
- Obliczanie wartości funkcji **f** dla zadanego argumentu **n** sprowadza się do rozpisania według równań rekurencyjnych schodząc z **n** co **1** aż do **1**

Funkcje rekurencyjne

- Funkcje obliczalne według Tezy Chucha-Tuiringa mogą być częściowe, tj. nie dla wszystkich argumentów określone

- Funkcje rekurencyjne i definicje za pomocą równości. Pierwotna rekursja jest przykładem takiej definiującej równości. Funkcje rekurencyjne (wg. Herbrand-Gödel) są definiowane poprzez (niesprzeczne) równości i dowód (w Arytmetyce), że jest to funkcja globalna, tj., że jest określona dla wszystkich argumentów ze swojej dziedziny.

Proste funkcjonały

- Sam schemat rekursji pierwotnej to funkcjonał wyższego rzędu **F**. Bierze jako input dwie funkcje (**h** i **g**) i produkuje jako output inną funkcję **f**

  (**h** , **g**) **→ f**

  F(h, g)(n, x) = f(n, x) // równośc definiująca
- **Fukcjonał F jako obiekt, który mozna używać do obliczeń?**
- Jak?

  F(h, g)(1, x) = h(x)

  F(h, g)(n+1, x) = g(n, F(h, g)(n, x) ,x)

- Są to obliczeniea symboliczne poprez przepisywanie symboli (term rewriting) zgodnie z równaniami definiującymi

Term rewriting systems

- Systemy przepisywania nazw (terminów).
- Ogólnie i najbardziej abstrakcyjnie: zbiór **A** i relacja binarna **->** na tym zbiorze.
- Elementarne obliczenie w tym systemie, to przejście **a1->a2**.
- **Przykład.** **A** składa się z wszystkich słów skończonych nad alfabetem: **a, b i c**. Reguły przepisywania (redukcji) :

  ab ->c, ca -> b, stosowane do pod-słów (pod-termów)
- term **aabca**
  - **aabca -> acca -> acb**
  - **aabca -> aabb -> acb** // **acb** to postać normalna dla tego termu
- term **cab**
  - **cab -> bb**
  - **cab -> cc** // brak postaci normalnej

Term rewriting systems

- Termy to słowa (napisy) dla ustalonego alfabetu według ustalonej składni. Alfabet to zmienne, stałe, symbole funkcyjne, i operatory do konstrukcji termów, np. :
  - aplikacja **()**, np. **f(x)**
  - lambda abstrakcja **λ**, np. **λx.f(x)**
- Każdy term jest jakiegoś typu
- Symbole również dla oznaczania typów
- Podstawianie w termie, tj. wszystkie występowanie ustalonej zmiennej wolnej w tym termie na inny term (w typowanej wersji typ wstawianego termu musi być taki sam jak typ tej zmiennej)
- W rachunku lambda, reguły redukcji termów to **alfa, beta, eta**
- Elementarne obliczenie to przejście z termu początkowego do następnego za pomocą jednej z reguł redukcji
- Co jeśli brak postaci normalnej? Obliczenie bez sensu?

Term rewriting systems

- Ten rodzaj obliczeń można nazwać symbolicznym. Odpowiada to obliczeniom na kartce papieru za pomocą ołówka według reguł przepisywania. Tak jest z obliczeniami liczbowymi i obliczeniami algebraicznymi na symbolach.

- Można używać równości skierowanych (od lewej strony do prawej) jako dodatkowych reguły przepisywania. Wtedy schemat rekursji to także reguły redukcji termów. Ogólnie wszystkie powyższe techniki obliczeniowe dają się sprowadzić do przepisywania termów a więc do **obliczeń symbolicznych**

Termy i funkcjonały

A i **B** to symbole oznaczające typy. Typ funcyjny jest oznaczany poprzez

A → B

- **f** typu **A → B**, oznaczenie **f: A → B**
- stała **a:A** oraz zmienna **x:A**
- termy **f(a), f(x)** są typu **B**
- term **λx.f(x)** jest typu **A → B**, oznacza funkcję
- **y** jest typu **A → B** , zaś **z** jest typu **B → C**
- **λy.λz.z(y(x))** jest typu **(A→B) →** (**B→C) → (A→C)** , oznacza funkcjonał kompozycji dwóch funkcji **f: A→B** oraz **g:B→C** , w funkcję **h:A→C** , taką że **h(x) = g(f(x))**

- Czym jest składanie (kompozycja) funkcji? Intuicyjnie proste i oczywiste!

Termy i funkcjonały

Typy jako obiekty, zmienna **X** oznacza „dowolny” typ

- **F** to symbol funkcjonału operującego na typach
- **Przykład.** Oznaczmy „→” przez **F**, wtedy **F(A,B) = (A→B).**

  Lambda abstrakcja **Λ,** np. **ΛY.ΛX. F(X,Y)** oznacza funcjonał, który bierze dwa typy i zwraca typ funkcyjny

Zmienne dla typów

- zmienna x^A jest typu **A**, zaś x^U jest typu **U**
- term $\lambda y^{U\to V}\lambda z^{V\to W}\ \lambda x^U.(y^{U\to V}(x^U))$ jest typu **(U→V) → (V→W) → (U→W)** oznaczmy ten term przez **Comp(U,V,W),** jest to **λy.λz.z(y(x))** , tj. taki sam term z poprzedniego slajdu
- **ΛU. ΛV.ΛW. Comp(U,V,W)** -- cóż ten term oznacza?
- Funcjonał, który dla trzech konkretnych typów (**A**,**B** i **C**) zwraca funkcjonał komponujący dwie funkcje (**f** i **g**) odpowiednich typów, taki że

  ΛU. ΛV.ΛW. Comp(U,V,W)(A)(B)(C)(f)(g)(x) = g(f(x))

Intuicyjnie ten funkcjonał jest prosty i oczywisty

Po co to wszystko? No to jeszcze jeden przykład

Prof. Andrzej Grzegorczyk (1922-2014). Rekursja na wyższych skończonych typach i funkcjonał nazwany Rekursorem Grzegorczyka.

- Notacja dla aplikacji **()**: **f(a)(c)(n)** jest to samo co **((f(a)) (c))(n)**
- nowy symbol funkcyjny R^A typu
- **N → (N → (A→A)) → (A → A)**; **N** to typ liczb naturalnych

term $R^A(n)(c)(a)$ **i** jego definicja

- dla zmiennych **a:A**, **c: N → (A→A),** oraz **n:N**

  $R^A(1)(c)(a) = a$

  $R^A(n+1)(c)(a) = c(n)(R^A(n)(c)(a))$

- **Interpretacja:** R^A bierze jalo input liczbę naturalną **n+1**, a następnie **c** , (tj. ciąg funkcji z **A** do **A**) i zwraca jako wynik kompozycję pierwszych **n** funkcji w tym ciągu. Dla **n=1** wynik jest funkcją stałą z **A** do **A**
- Intuicyjnie proste i oczywiste.

Po co to wszystko? No to jeszcze jeden przykład

Z termu **$R^A(n)(c)(a)$** robimy term poprzez lambda abstrakcję oznaczający funkcjonał

$\lambda n.\ \Lambda c.\ \Lambda a.\ R^A(n)(c)(a)$

onaczmy ten term przez $\underline{R}^A$,

Lambda abstrakcja: **$\Lambda X.\ \underline{R}^X$** jest to term polimorficzny Jaki jest jego typ?

$\Lambda X.\ \underline{R}^X(A)$ to to samo co $\underline{R}^A$

Jaki funkcjonał ten term oznacza? - intuicyjnie jest to proste i oczywiste

- Jak wykonywać obliczenia na takich termach? Poprzez przepisywania; w tym przypadku równania rekurencyjne (prawostronnie skierowane) są regułami przepisywania. W tym celu term (napis) ma kontekst definicyjny (Environment), który wyznacza jego semantykę obliczeniową. Bez tego kontekstu to tylko nic nie znaczący napis.
- Zarządzanie i operowanie takim temami i ich kontekstami nie jest proste.

Po co to wszystko?

- Czy funkcja, funkcjonał jest nazwą plus pewne (specyficzne dla niego) metody przepisywania jako obliczenia na nim?
- Czy raczej term i jego kontekst tylko opisują funkcjonał ?

- Pojęcie obiektu (funkcjonału) wyższego rzędu zawiera w sobie następujące pojęcia, które są TYLKO opisywane przez termy:
  - typy i obiekty tych typów
  - typ funkcyjny i związane z nim input, output
  - konstrukcja funkcjonału: wejście (input), wyjście (output) i ciało
  - aplikacja argumentu do funkcjonału: wrowadzenie argumntu do wejścia
  - wynik aplikacji (jako output)
  - kompozycja dwóch (lub więcej) funkcjonałów jako połącenie wyjścia (output) jednego funkcjonału z wejściem (input) drugiego funkcjonału
- **Czy są to konkretnie (fizycznie) realizowane pojęcia?**

Po co to wszystko?

- **Teoria.** Czym są funkcjonały (obiekty wyższych rzędów) i jak ich używać w obliczeniach?
- (Banach-Mazur, Rosa Peter, Kurt Godel i Andrzej Grzegorczyk, Kleene, Kreisler, Scott, Platek, Girard, Reynolds, P. Martin-Lof, ...)

- **Programowanie.** W językach funkcyjnych operowanie na obiektach wyższych rzędów, tj. funkcjach i funkcjonałach, sprowadza się do redukcji termów (tzw. lazy evaluation) Systemie F, Martin-Löf Type Theory i rachunku Lambda. Obliczenia na obiektach wyższych rzędów jest symboliczne, tj. polega na operowaniu na symbolach.

- **Paradygmat I. Obliczenia na obiektach (funkcjonałach) wyższych rzędów mogą być tylko symbolioczne poprzez przepisywanie termów oznaczającym funkcjonały.**

Von Neumann computer architecture

Genialne prosta

Central Processing Unit

Control Unit

Arithmetic/Logic Unit

Input Device

Output Device

von Neumann bottleneck

Memory Unit

Programowanie w komputerze von Neumnna

```
begin
    int n, i, x;
    input(n);
    i:=1;
    x:=0;
    while (n > i) do ( i++; x:= x + i);
    output (x);
end
```

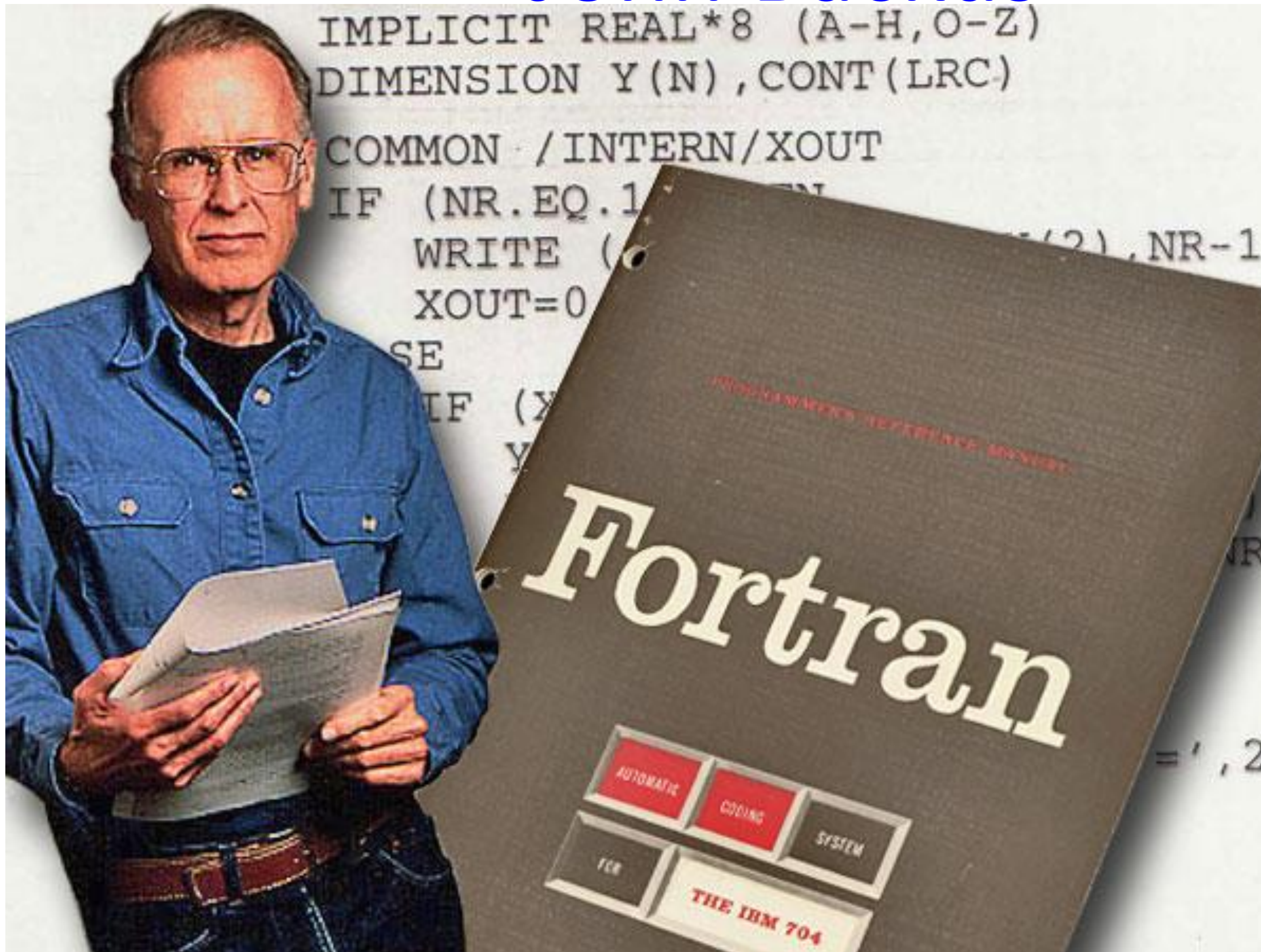
- w asemblerze (kodzie maszynowym procesora) pętla ``while'' jest trochę bardziej złożona. Ale typ int to (w realizacji, tj. kompilacji) skończona i ograniczona reprezentacja liczb
- Zmienna to miejsca w pamięci adresowane jej nazwą. W asemblerze nazwa zmiennej to jej adres; są tam instrukcje warunkowego skoku (w zależności od stanu flag). W zasadzie kod jest bardzo podobny (powiedzmy, że izomorficzny) tylko dłuższy. Bo przecież każdy program jest kompilowany do kodu maszynowego.

Czy możliwe są inne architektury ?

- **TAK.** Umysł człowieka nie jest komputerem von Neuamnna (patrz *)
- Czy zostały skonstuowane do tej pory istotnie różne (non-von Neumann) komputery? **NIE**
- Czy możliwa jest obliczalność niesymboliczna? Owszem, jest to obliczalność analogowa jako przeciwieństwo do obliczeń tzw. cyfrowych.

- *(*) John von Neumann, The Computer and the Brain. Silliman Memorial Lectures, New Haven: Yale University Press 1958*

John Backus



Backus-Naur form (BNF), a widely used notation to define formal language syntax

Von Neumann bottleneck

Wąskie gardło – ograniczona przepustowość pomiędzy CPU (procesorem) a pamięcią w porównaniu z rozmiarem pamięci.

CPU jest zmuszony czekać (3/4 swojego czasu lub więcej) na dane z pamięci /zapis danych w pamięci.

Jest to ograniczenie widoczne zwłaszcza jeśli duża porcja danych ma być przetworzona w stosunkowo prosty i szybki sposób przez CPU.

Coraz bardziej zwiększa się szybkość CPU (GHz) oraz wielkość pamięci (TB). Przepustowość pomiędzy CPU a pamięcią nie za bardzo.

Delegowanie części przetwarzania do karty graficznej, czy urządzeń INPUT-OUTPUT niewiele daje.

Problem z wąskim gardłem (von Neumann bottleneck) staje się coraz bardziej poważny.

Architektura hardware staje się coraz bardziej złożona.

Von Neumann language

Architektura komputera wg. von Neumanna jest dominująca (i jedyna) od samego początku czyli od lat 40-tych XX wieku.

Język programowania von Neumanna to taki, który jest izomorficzny (na wysokom poziomie abstrakcji) z architekturą komputera von Neumanna.

Izomorfizm:

- program variables ↔ computer storage cells
- control statements ↔ computer test-and-jump instructions
- assignment statements ↔ fetching, storing instructions
- expressions ↔ memory reference and arithmetic instructions.

Von Neumann language

metaphor from John Backus:

- ➢ assignment statements in von Neumann languages split programming into two worlds
- ➢The first world consists of *expressions*, an orderly mathematical space with potentially useful algebraic properties: most computation takes place here.
- ➢The second world consists of assignment *statements*

Von Neumann language

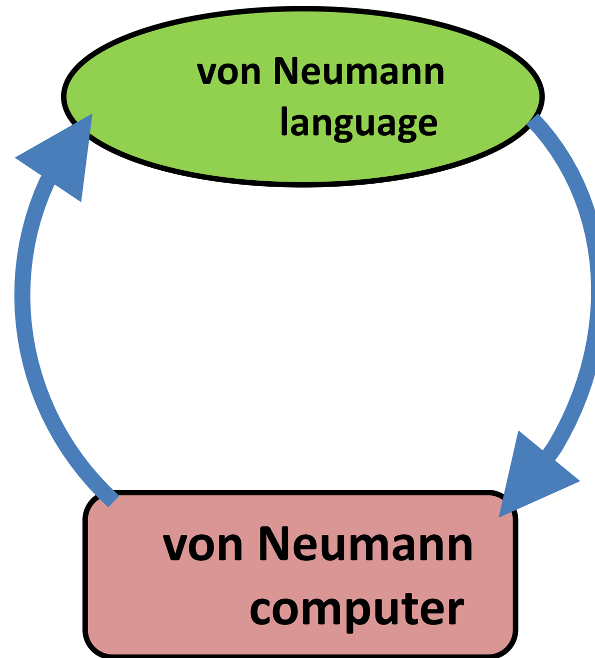
The von Neumann bottleneck was described by John Backus in his 1977 ACM Turing Award lecture. According to Backus:

- Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic

An intellectual bottleneck:

- that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself, but where to find it.

Von Neumann vicious cycle



Paradigm.

Most of the current programming languages are high-level abstract isomorphic copies of von Neumann computer architectures.
There is a vicious cycle caused by this isomorphism.
Non-von Neumann computer architectures cannot be developed because of the lack of widely available and effective non-von Neumann languages. New languages cannot be created because of lack of conceptual foundations for non-von Neumann architectures.

Von Neumann language

Value-level programs

- are those that describe how to combine various *values* (i.e., numbers, symbols, strings, etc.) to form other values until the final *result values* are obtained. New values are constructed from existing ones by the application of various value-to-value functions, such as addition, concatenation, matrix inversion, and so on.

Conventional, von Neumann programs are value-level: expressions on the right side of assignment statements are exclusively concerned with building a value that is then to be stored.

Non-von Neumann language

John Backus: function-level programming

- programs as mathematical objects
- program is built directly from programs that are given at the outset, by combining them with program-forming operations or functionals.

Backus' function-level approach is different than so called functional programming based on lambda calucus and combinatory logic

Haskell, F#, etc.

Według Johna Backusa, współczesne języki programowania oparte na rachunku lambda i symbolicznym obliczaniem na obiektach wyższych rzędów są nadal von Neumann.

Języki funkcyjne a funkcjonały

Jak są kompilowane i wykonywane termy (wyrażenia) odpowiadające funkcjonałom wyższego rzędu?

Poprzez tzw. ``lazy evaluation'', tj. dynamiczne przepisywanie termów i modyfikacja kodu ale tylko wtedy kiedy jest to konieczne. A konieczne jest wtedy kiedy w przypisaniu do zmiennej typu prostego występuje ten term, tzn. w komórce pamięci odpowiadającej tej zmiennej ma być zapisana konkretna wartość (konkretne bajty).

Te przypisane może pojawić się dopiero przy wykonywaniu programu. Co więcej czasami (jeśli używane są termy polimorficzne wyższego rzędu) dopiero przy wykonywaniu można sprawdzić zgodność typów (? tak czy nie, jest to złożony problem).

Języki funkcyjne a funkcjonały

Same lazy evaluation (LE) to generalnie przetwarzanie obiektów typu string (jakkolwiek mogą to być struktury drzewiaste). Wykonywane jest to w sposób imperatywny, tzn. musi być odpowiedni osobny kod do realizacji LE.

Modyfikacja kodu powoduje, że wyrażenie w przepisaniu składa się już z symboli bezpośrednio kompilowanych.

Ta modyfikacja wygląda na pojedyncze przypisanie (zapis w pamięci) ale po kompilacji do kodu maszynowego jest to ogromna ilość przypisań, co jest typowe w architekturze von Neumanna.

Języki funkcyjne a funkcjonały

Na czym polegają obliczenia na funkcjonałach w typowym funkcyjnym języku programowania.

Przykład. Term `ΛX. R^X` jest w kodzie zadeklarowany (jest to tzw. Environment)

int to term oznaczający typ liczb całkowitych

w trakcie wykonania programu zmieniają się fragmenty kodu i zostanie obliczony symbolicznie term
`c` typu `int → (int →int)` , w taki sposób, że spełnione są równania
`c(i) == f` dla każdego `i int` zaś
term (zadeklarowany wcześniej) `f` jest typu `int → int`, oraz `f(n)`
to suma liczb od **1** do **n**

Języki funkcyjne a funkcjonały

w trakcie wykonania programu (z dynamiczne modyfikowanym kodem i powtórną kompilacją) pojawia się (ewentualnie) przypisanie

```
int y = ΛX. R^X (int)(2)(c)(5);
```

co się wtedy dzieje? Jak ten term jest przepisany na elementarne komendy w asemblerze (kodzie maszynowym)? Każda komenda w asemblerze to albo przypisanie do pamięci, rejestru, do input lub output albo komendy kontrolne.
Jaki jest wynik, tj. co jest zapisane w pamięci pod adresem `y`?

Na koniec drobna uwaga. A co jeśli nastąpi przepełnienie (overflow) na zmiennych typu `int`? Tj. przekroczenie zakresu typu `int`.

Stan obecny

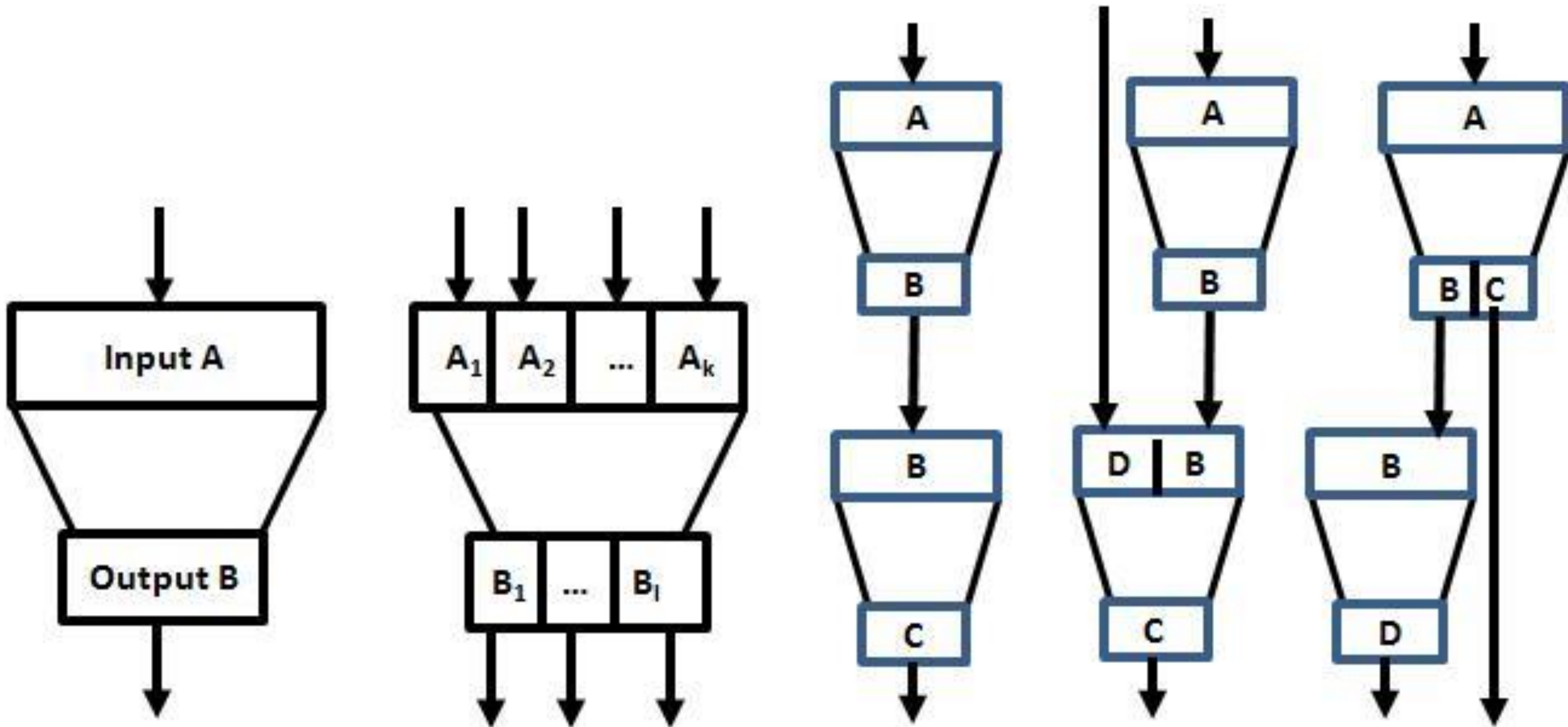
Paradygmat II. W programowaniu obliczenia na obiektach wyższych rzędów mogą być tyko symboliczne, tzn. poprzez lazy evaluation

Przełamanie tego paradygmatu, to jednocześnie nowe non-von Neumann języki programowania i nowa non-von Neumann architektura komputera

Jest o co powalczyć!

Trochę intuicji

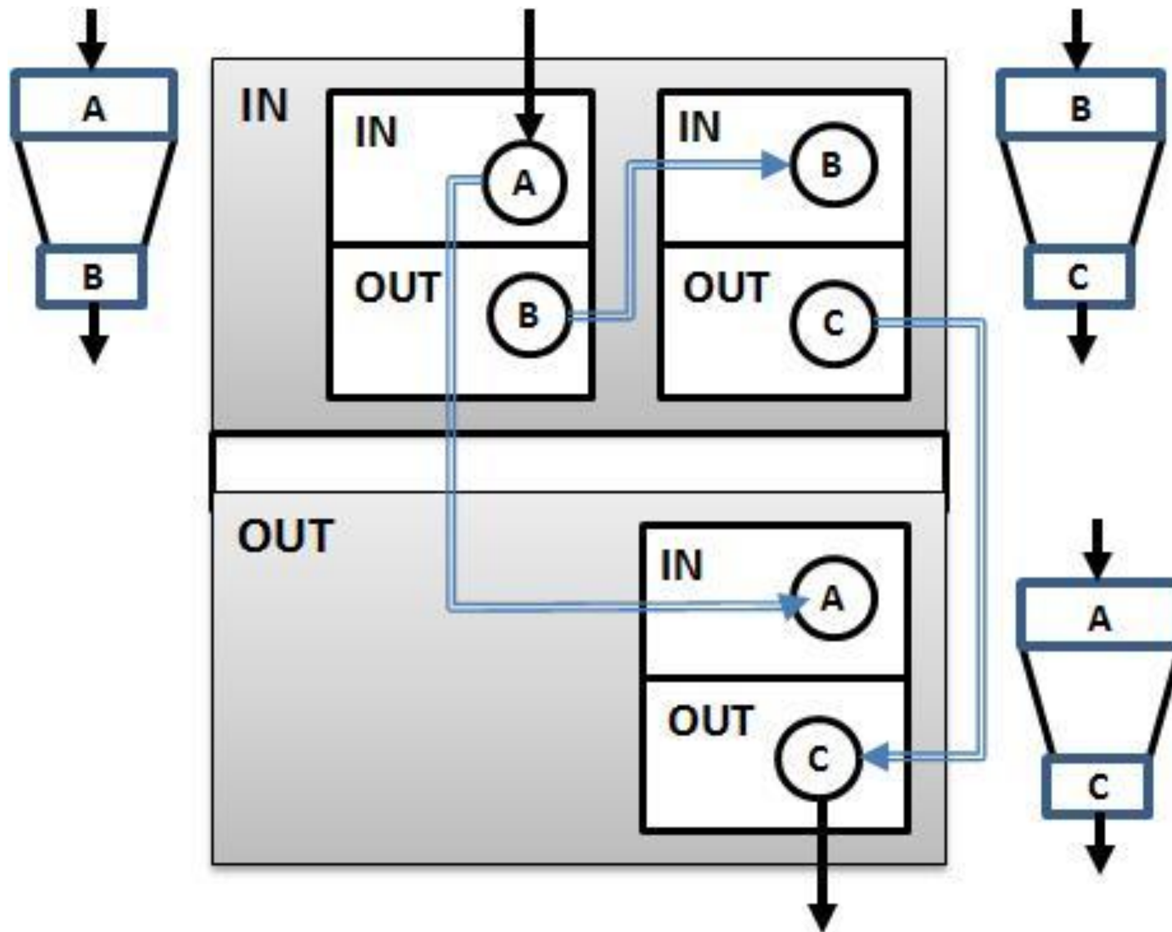
Jak można inaczej.
Typy jako gniazda. Funkcja(funkcjonał) jako gniazdo input, ciało oraz gniazdo output.
Kompozycja jako zestawienie łącza pomiędzy output a input



Trochę intuicji

Kompozycja jako funkcjonał

$$compose_{A,B,C} : ((A \to B); (B \to C)) \to (A \to C)$$



Trochę intuicji

Obiekt to układ gniazd z ustalonymi łączami.
Obliczenia na tych obiektach (funkcjonałach wyższych rzędów) to dynamiczne zestawianie łącz pomiędzy gniazdami