

Sztuczna inteligencja

```
166
167 (defn graph-search
168   "Finds a state that satisfies goal? predicate. Starts with states and
169   proceeds according to successors-generator and combiner. It never considers
170   the same state twice, keeping the set of already examined states."
171   ([states goal? successors-generator combiner]
172    (graph-search states goal? successors-generator combiner #{}))
173   ([states goal? successors-generator combiner examined-states]
174    (log/debug "Search: %s\n" (str (seq states)))
175    (cond (nil? states) nil
176          (goal? (first states)) (first states)
177          :else (recur
178                 (combiner (filter (fn [state]
179                                     ;; new state is valid only if not present
180                                     ;; in states nor in examined-states
181                                     (not (or (member? state states)
182                                              (examed-states state)))))
183                 (successors-generator (first states)))
184          (rest states))
185          goal?
186          successors-generator
187          combiner
188          (union #{(first states)} examined-states))))
189
```

Literatura

1. Russel, S. J., Norvig, P.: Artificial Intelligence A Modern Approach Second Edition. Pearson Education Inc., Upper Saddle River, New Jersey 07458 (2003)
2. Norvig, P.: Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp. Morgan Kaufmann (1991)
3. Abelson, H., Sussman, G. J.: Structure and Interpretation of Computer Programs. ISBN 0-262-01077-1. MIT Press (1984)
4. Żurada J., Barski M., Jędruch W.: Sztuczne sieci neuronowe, Wydawnictwo Naukowe PWN, Warszawa (1996)
5. Tadeusiewicz R., Sieci neuronowe, Akademicka Biblioteka Cyfrowa AGH, <http://winntbg.bg.agh.edu.pl/skrypt/0001/>

Czy rzeczywistość jest opisywalna ?

- **David Hilbert** (ur. 23 stycznia 1862 w Królewcu (Prusy Wschodnie) - zm. 14 lutego 1943 w Getyndze) - matematyk niemiecki. Sformułował tzw. Entscheidungsproblem (*circa* 1900) – pytanie o istnienie mechanicznej procedury zdolnej do udowodnienia dowolnego twierdzenia.
- **Kurt Gödel** (1906-1978) – austriacki logik i matematyk; autor ważnych twierdzeń z zakresu logiki matematycznej, najważniejsze – Twierdzenie Gödla o niezupełności systemów logicznych (1931). Wprowadził do matematyki pojęcie nierozstrzygalności.
- **Alan Mathison Turing** (ur. 23 czerwca 1912 w Londynie - zm. 7 czerwca 1954 w Wilmslow) - angielski matematyk, twórca uniwersalnej maszyny Turinga i jeden z twórców informatyki. Dowiódł, że nawet najbardziej uniwersalny automat nie jest zdolny do rozstrzygnięcia o prawdzie lub fałszu dowolnego twierdzenia w matematyce (1936).

Źródło: <http://en.wikipedia.org>
<http://pl.wikipedia.org>

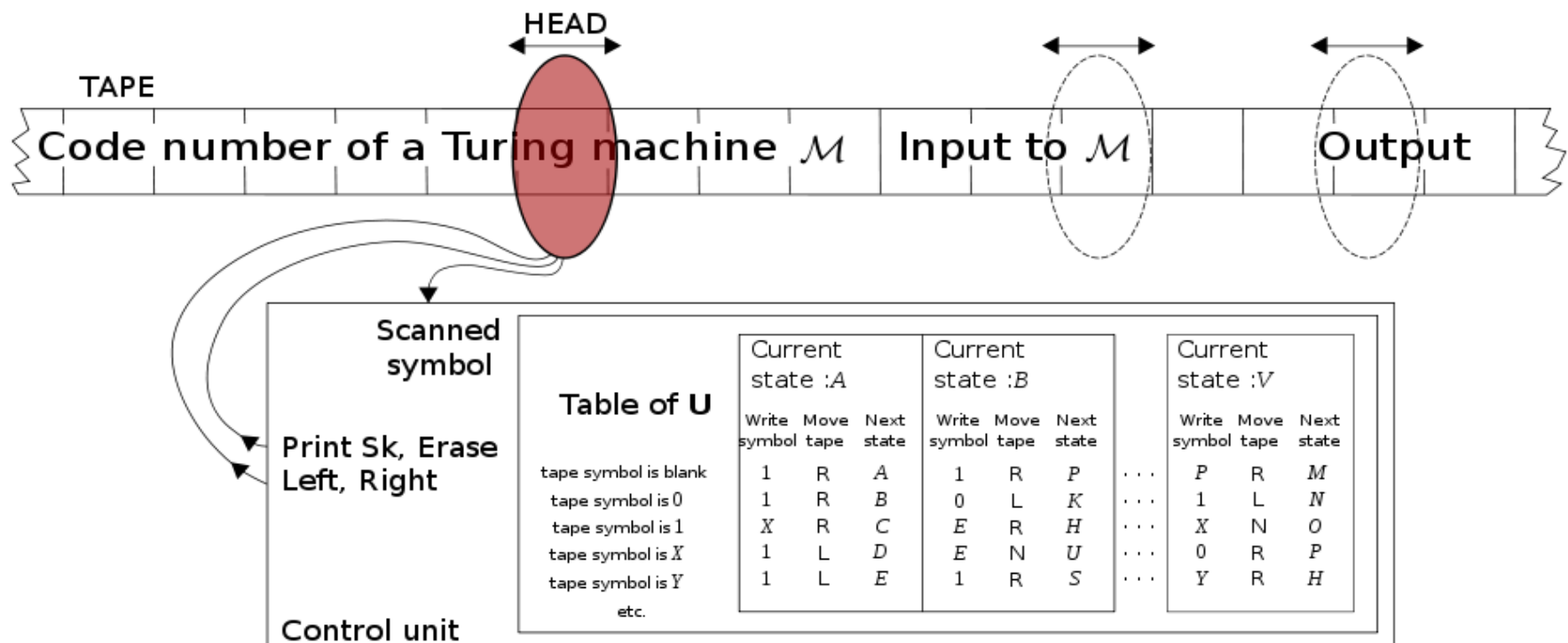
Uniwersalna maszyna Turinga

Maszyna Turinga - abstrakcyjny model komputera służący do wykonywania algorytmów.

Maszyna Turinga składa się z nieskończenie długiej taśmy podzielonej na pola. Taśma może być nieskończona jednostronnie lub obustronnie. Każde pole może znajdować się w jednym z M stanów. Maszyna zawsze jest ustawiona nad jednym z pól i znajduje się w jednym z N stanów. Zależnie od kombinacji stanu maszyny i pola maszyna zapisuje nową wartość w polu, zmienia stan i przesuwa się o jedno pole w prawo, w lewo lub pozostaje na miejscu. Taka operacja nazywana jest rozkazem. Maszyna Turinga jest sterowana listą zawierającą dowolną ilość takich rozkazów. Liczby N i M mogą być dowolne, byle skończone. Czasem dopuszcza się też stan $(N+1)$, który oznacza zakończenie pracy maszyny.

Źródło: <http://en.wikipedia.org>
<http://pl.wikipedia.org>

Uniwersalna maszyna Turinga - schemat



Źródło:

http://upload.wikimedia.org/wikipedia/commons/thumb/4/43/Universal_Turing_machine.svg/1000px-Universal_Turing_machine.svg.png

John von Neumann

John von Neumann (ur. 28 grudnia 1903 w Budapeszcie, zm. 8 lutego 1957 w Waszyngtonie) – matematyk, inżynier chemik, fizyk i informatyk. Wniósł znaczący wkład do wielu dziedzin matematyki - w szczególności był głównym twórcą teorii gier, teorii automatów komórkowych (w które znaczący wkład miał także **Stanisław Ulam**), i stworzył formalizm matematyczny mechaniki kwantowej. Uczestniczył w projekcie Manhattan. Przyczynił się do rozwoju numerycznych prognoz pogody.

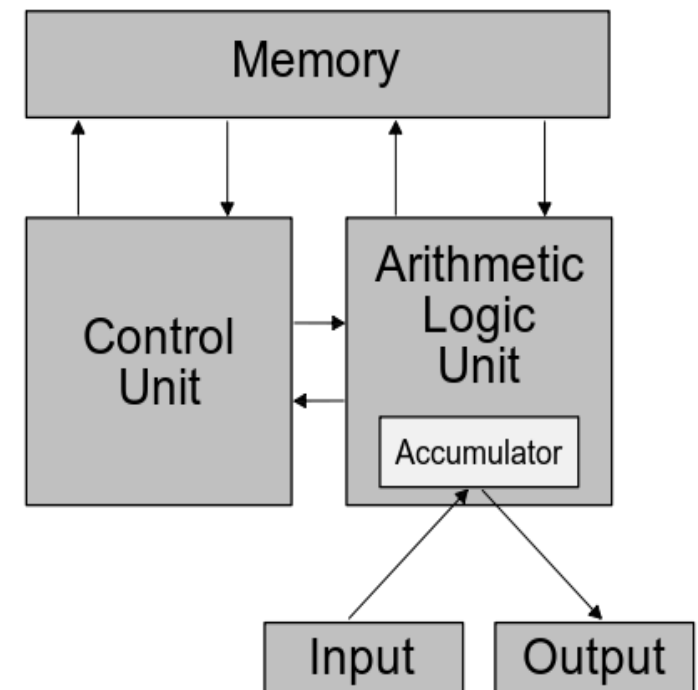
Architektura komputera w-g von Neumanna

Opracowana na projekcie EDVAC (Electronic Discrete Variable Automatic Computer), będącym następcą projektu ENIAC. Przedstawiona po raz pierwszy w roku 1945 w artykule *First Draft of a Report on the EDVAC*.

John von Neumann oparł się na pomysłe Turinga.

Jest to architektura, która – z pewnymi modyfikacjami – jest architekturą obecnych komputerów.

Fundamentalną cechą komputera von Neumanna jest fakt **modyfikacji** pamięci. Jest to pochodna cechy maszyny Turinga – głowica mogła **modyfikować stan taśmy**.



Czy komputer von Neumanna potrafi myśleć ?

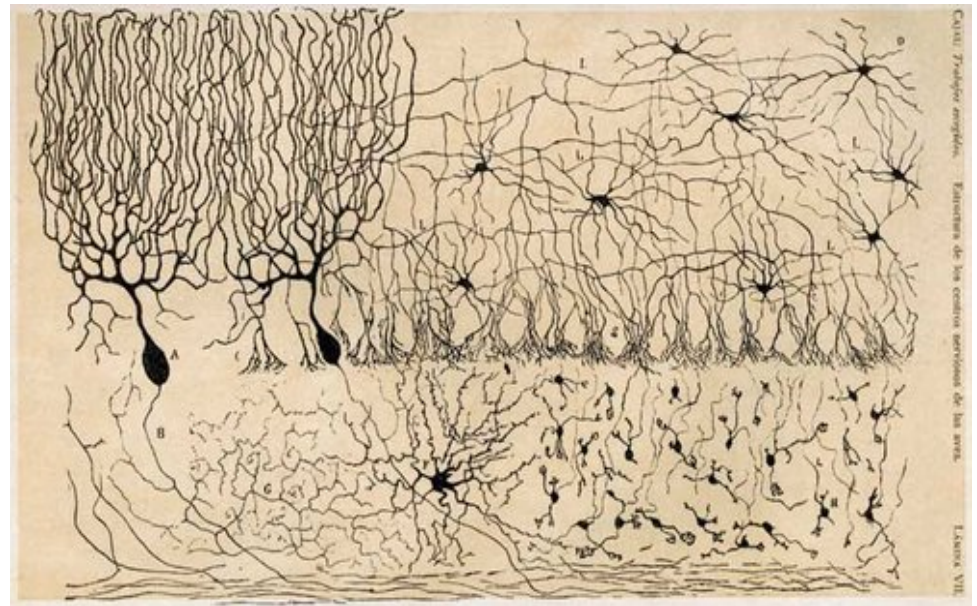
Problem ten nurtował naukowców od momentu powstania nowoczesnego komputera.

Podejmował go sam Turing. Oraz wielu innych wybitnych myślicieli XX w. (Wittgenstein, Haldane, Minsky, Papert).

Na drodze do rozwiązania tego problemu stoi fakt, że związek pomiędzy działaniem sieci neuronów a tym, co nazywamy (samo)świadomością, nie jest znany. Zdanie to jest prawdą również dzisiaj – w początkach XXI stulecia.

Neurobiologia – początki

Santiago Ramón y Cajal (ur. 1 maja 1852 w Petilla de Aragón, Nawarra, zm. 17 października 1934 w Madrycie) – hiszpański histolog, neuroanatom, prekursor neurobiologii. Otrzymał Nagrodę Nobla w dziedzinie medycyny (razem z Camillo Golgi) za badania nad strukturą systemu nerwowego.



Komórki mózdzku. Rysunek z roku 1905

Źródło: <http://neuroimages.tumblr.com/post/178000476>

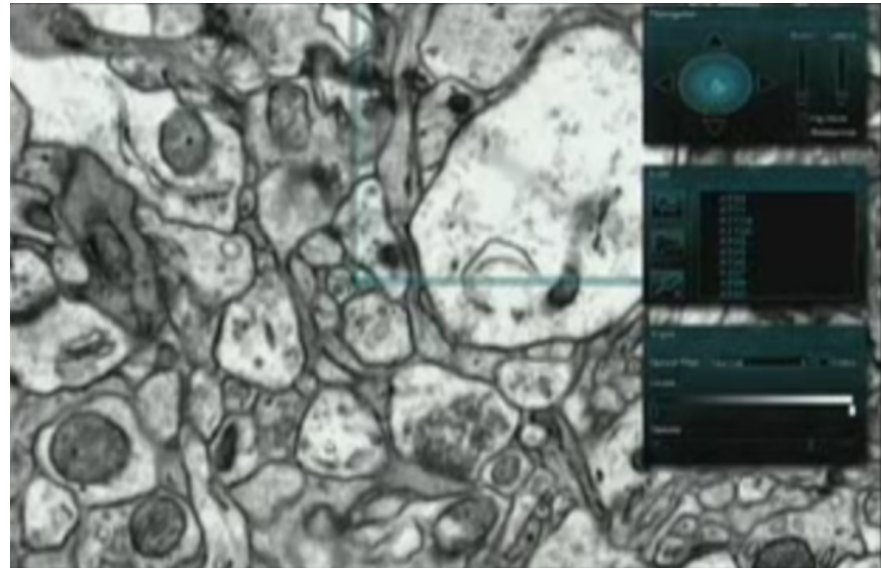
Źródło: http://pl.wikipedia.org/wiki/Santiago_Ramon_y_Cajal

Neurobiologia – stan obecny

„O mózgu wiemy dzisiaj mniej, niż o jakimkolwiek innym organie naszego ciała. Poszedłbym nawet dalej. Wiemy o mózgu mniej niż o czymkolwiek, co jest dostępne ludzkiemu poznaniu. [...] Do cyfrowego odwzorowania sieci połączeń nerwowych w mózgu potrzeba przestrzeni dyskowej o rozmiarach rzędu **1 miliona petabajtów.**”

Jeff Lichtman,
Professor of Molecular and
Cellular Biology
Harvard University

1 PB = 1000 TB = 1 mln GB



Komórki mózgu myszy. Wizualizacja z wykorzystaniem systemu firmy Microsoft.

Źródło: <http://www.youtube.com/watch?v=6b6DEUM5Gbw>

Dartmouth College

Prywatny uniwersytet założony w roku 1769, zlokalizowany w mieście Hanover w stanie New Hampshire (USA). Należy do tzw. Ligi Bluszczowej (ang. *Ivy League*). Strona oficjalna: <http://www.dartmouth.edu/>



Biblioteka Baker Memorial. Mieści 240 tys. woluminów.

Źródło: http://en.wikipedia.org/wiki/File:Dartmouth_College_Baker_building.jpg

Konferencja w Dartmouth College (1956)

Dartmouth Summer Research Conference on Artificial Intelligence. Opłacona przez Fundację Rockefellera (7 mln \$). Warsztaty naukowe poświęcone rozwiązywaniu trudnych problemów przez maszyny cyfrowe. Miały one decydujące znaczenie z punktu widzenia powstania i rozwoju dziedziny nazwanej *sztuczną inteligencją*.

W swoim artykule (napisanym jeszcze w roku 1955) nadesłanym na tę konferencję John McCarthy po raz pierwszy użył pojęcia „**sztuczna inteligencja**”.

Uczestnicy Konferencji w Dartmouth

1. **John McCarthy** – twórca języka Lisp, autor pojęcia *sztuczna inteligencja*, pracownik naukowy Dartmouth, MIT, Princeton.
2. **Claude Shannon** – amerykański matematyk i inżynier, twórca *teorii informacji*
3. **Marvin Minsky** – badacz AI, założyciel MIT AI Lab., Nagroda Turinga w roku 1969, współautor książki „Perceptrony” (wraz z Seymourem Papertem) otwierającej rozwój sztucznych sieci neuronowych.
4. **Nathaniel Rochester** – stworzył komputer IBM 701, napisał pierwszy assembler.
5. **Herbert Simon** – amerykański politolog, ekonomista i psycholog, laureat Nagrody Nobla w dziedzinie ekonomii (1978) oraz Nagrody Turinga (1975), współtwórca jednego z pierwszych ważnych programów sztucznej inteligencji – *GPS (General Problem Solver)* – wraz z Allenem Newellem.
6. **Ray Solomonoff** – matematyk, badacz w dziedzinie informatyki, twórca pojęcia *prawdopodobieństwa algorytmicznego* łączącego teorię informacji ze złożonością Kolmogorowa.
7. **Oliver Selfridge** – brytyjski naukowiec zajmujący się sztucznymi sieciami neuronowymi (perceptron), pracownik Laboratorium Lincolna w MIT.
8. **Arthur Samuel** – pionier inteligentnych gier komputerowych, napisał pierwszy w historii uczący się program grający (w warcaby).
9. **Allen Newell** – współtwórca programu GPS, laureat Nagrody Turinga (1975), pracownik Carnegie Mellon Univ. Promotorem jego doktoratu był Herbert Simon.

Źródło: http://en.wikipedia.org/wiki/Dartmouth_Conference

Rozwój w latach 1956 - 1974

Simon, Newell: „W ciągu 10 lat komputer będzie mistrzem szachowym” (1958)

Minsky: „W czasie od 3 do 8 lat będziemy dysponować maszyną o inteligencji przeciętnej istoty ludzkiej” (1970)

Finansowanie badań przez DARPA (*Defense Advanced Research Projects Agency*). Najważniejsze ośrodki i badania:

- ♦ MIT – projekt MAC (*Project on Mathematics and Computation*), ok. 2-3 mln \$ rocznie przez całą dekadę lat 70-tych XX w. Minsky, McCarthy
- ♦ Carnegie Mellon University, kilka mln \$ rocznie, Simon, Newell
- ♦ Stanford University AI Project, McCarthy
- ♦ Edinburgh University, Donald Michie

Zima sztucznej inteligencji

Brak widocznego progresu po pierwszych zachęcających wynikach.

Powody:

- ***Eksplozja kombinatoryczna***, złożoność wykładnicza lub gorsza większości algorytmów.
- Ograniczona moc obliczeniowa komputerów.
- Brak wglądu w działanie umysłu.

Okres 1974 – 1980 – ograniczenie lub wstrzymanie dotacji na badania.

Okres 1980 – 1987 – rozwój systemów ekspertowych, nowe ciekawe wyniki dotyczące sieci neuronowych.

Okres 1993 do dziś – rozwój mechanizmów reprezentacji wiedzy, składowania informacji (Google, Amazon), zwycięstwo Deep Blue nad Kasparowem (1997), większość ważnych problemów nie znalazła swojego definitywnego rozwiązania. Z dużą dozą prawdopodobieństwa można powiedzieć, że $P \neq NP$.

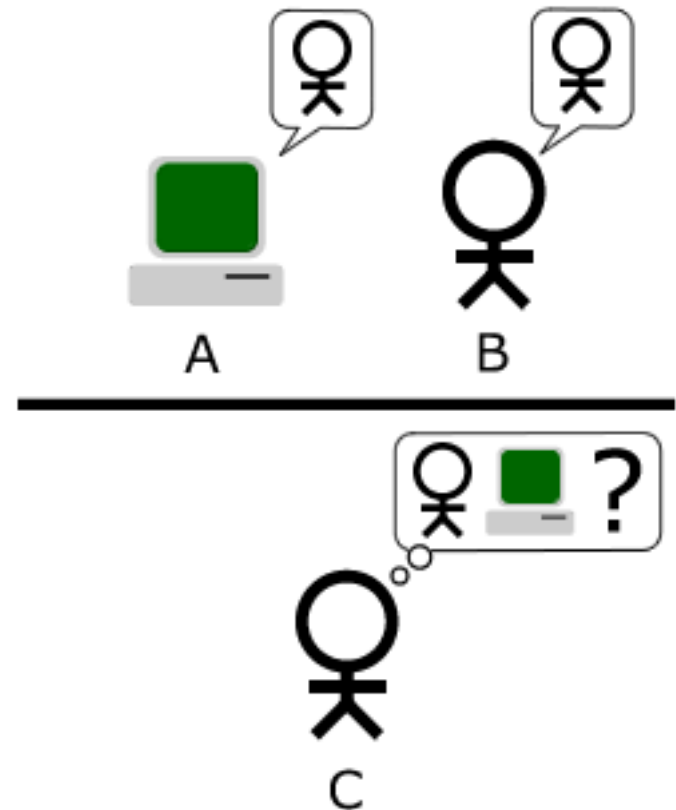
Źródło: http://en.wikipedia.org/wiki/History_of_artificial_intelligence

Sztuczna inteligencja - definicja

<i>Systemy, które myślą jak ludzie</i>	<i>Systemy, które myślą racjonalnie</i>
<p>Badania w zakresie kognitywistyki.</p> <p>Sztuczna świadomość (?)</p> <p>Czym jest myślenie, jaki jest związek pomiędzy działaniem mózgu a myśleniem ?</p>	<p>Arystotelesowskie próby ujmowania rzeczywistości i wnioskowania o niej w reguły formalne (logika).</p> <p>Problemy z reprezentacją wiedzy.</p>
<i>Systemy, które działają jak ludzie</i>	<i>Systemy, które działają racjonalnie</i>
<p>Równoczesne:</p> <ul style="list-style-type: none">- przetwarzanie języka naturalnego- reprezentacja wiedzy- wnioskowanie- uczenie się <p>Test Turinga i argument Chińskiego Pokoju Searle'a.</p>	<p>Systemy agentowe. Agent to dowolny mechanizm zdolny do działania zgodnego z oczekiwaniami jego twórców. Posiadający:</p> <ul style="list-style-type: none">- autonomiczne sterowanie- możliwości percepcji otoczenia- trwałość w czasie- zdolność do adaptacji (nauki)

Test Turinga (1950)

Człowiek C zadaje pytania kolejno obiektom A (komputerowi) oraz B (innej osobie), których nie widzi. Jeśli na podstawie uzyskanych odpowiedzi C nie jest w stanie rozpoznać, który z nich jest maszyną, a który człowiekiem, oznacza to, że obiekt A (komputer wraz z uruchomionym programem) przeszedł test - można go więc nazwać programem inteligentnym.



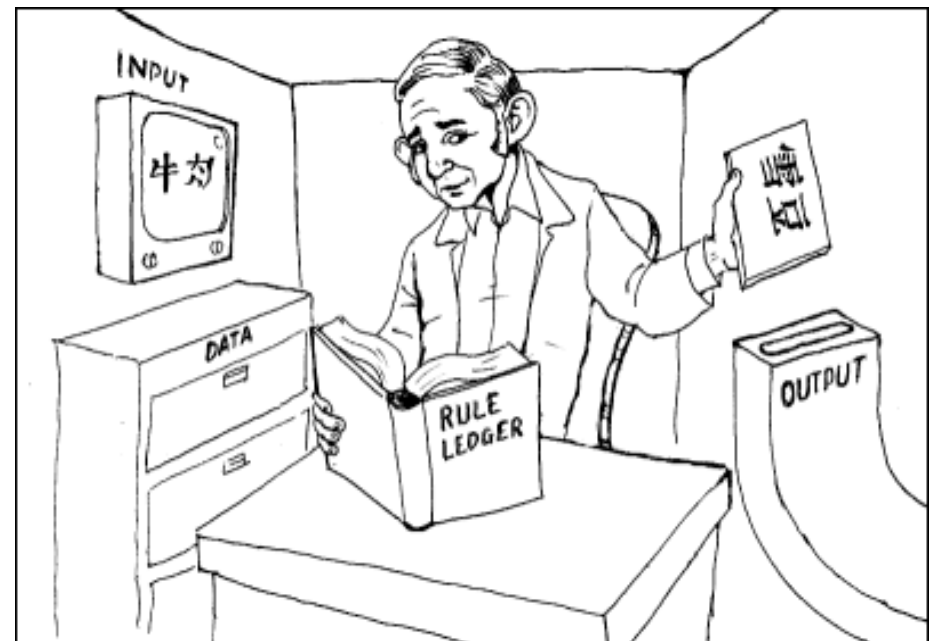
Chiński pokój Searle'a (1980)

John Rogers Searle (ur. 31 lipca 1932) – amerykański filozof zajmujący się filozofią umysłu.

Osoba umieszczona w pokoju dysponuje zbiorem reguł „tłumaczenia” chińskich znaków. Każda reguła stwierdza, że określonej ciągowi znaków odpowiada inny określony ciąg. Lokator pokoju przyjmuje „pytania” i generuje „odpowiedzi”. Searle stawia tezę, że w optymalnych warunkach taki „automat” byłby zdolny do przejścia testu Turinga. Jego działanie nie ma jednak nic wspólnego z rozumieniem (języka chińskiego w tym wypadku).

Jest to argument przeciwko tzw. mocnej sztucznej inteligencji – automat ze swej natury nigdy nie będzie myślał tak, jak robi to człowiek.

http://en.wikipedia.org/wiki/Chinese_room
<http://subcortex.com/pictures/c-room.gif>



Co to jest abstrakcja ?

Abstrakcja (łac. *abstractio* - *oderwanie*) – w sztukach plastycznych: taka realizacja dzieła, w której jest ono pozbawione wszelkich cech ilustracyjności, a artysta nie stara się naśladować natury. Autorzy stosują różne środki wyrazu, dzięki którym "coś przedstawiają".

Kompozycja VII (1913).
Wasilij Wasiljewicz
Kandinski (ur. 4 grudnia 1866 w Moskwie, zm. 13 grudnia 1944 w Neuilly-sur-Seine) – rosyjski malarz, grafik i teoretyk sztuki. Współtwórca i jeden z najważniejszych przedstawicieli abstrakcjonizmu.



[http://pl.wikipedia.org/wiki/Abstrakcja_\(sztuka\)](http://pl.wikipedia.org/wiki/Abstrakcja_(sztuka))
http://en.wikipedia.org/wiki/Wassily_Kandinsky

Abstrakcja w filozofii, matematyce i psychologii

1. (łac. ***abstractio*** – ***oderwanie***). Proces budowania pojęć, w którym od rzeczy jednostkowych (konkretnych) dochodzi się do pojęcia ogólnego poprzez wyłanianie tego, co dla tych rzeczy jest wspólne. ***Uogólnianie***.

2. Potężny element naszego procesu myślowego pozwalający na niwelowanie ograniczeń naszego umysłu. Polega na ***pomijaniu*** (abstrahowaniu od) ***szczegółów pojęć***, którymi się posługujemy w ***procesie wnioskowania***.

Funkcja jako pojęcie abstrakcyjne

Funkcja - relacja określona na zbiorach A (dziedzinie) i B (przeciwdziedzinie), w której elementowi ze zbioru A odpowiada jeden i tylko jeden element ze zbioru B. Właściwość ta zachodzi **zawsze**. W matematyce **czas nie istnieje**.

1. \sin , \cos , $x \rightarrow 2x + 3$ to niewątpliwie osobne byty. Wszystkie one posiadają pewne wspólne cechy. Cechy te są opisane powyżej. Funkcja jest abstrakcją tych trzech (i nieprzeliczalnie wielu innych, podobnych) bytów.

Funkcja = tablica o potencjalnie nieskończonych (nieprzeliczalnych) rozmiarach (zależnie od właściwości dziedziny).

$1 \rightarrow 5$

$2 \rightarrow 7$

$3 \rightarrow 9$

$\cdot \rightarrow \cdot$

2. Abstrakcyjne ujęcie funkcji – receptura (przepis) na generowanie kolejnych elementów tablicy

$f: x \rightarrow 2x + 3$

Interpretacja

Proces polegający na zmianie stanu automatu pod wpływem podawanych na jego wejście zdań języka.

<i>Język</i>	<i>Interpreter</i>
polski (naturalny)	Umysł człowieka
maszynowy	Procesor
Python	python
kod bajtowy	java (JVM – ang. <i>Java Virtual Machine</i>)

Kompilacja

Tłumaczenie zdań jednego języka na zdania innego języka.

<i>Języki</i>	<i>Kompilator</i>
polski (naturalny) → angielski (naturalny)	Umysł człowieka
assembler → maszynowy	asm, tasm, masm, ...
C → maszynowy	gcc, cc,
Java → kod bajtowy	javac

Jak działa interpreter

Searle pokazał, że automat nie myśli w takim sensie, w jakim myślimy my, ludzie. Z perspektywy niskiego poziomu abstrakcji – maszyny Turinga – *interpretacja* jest *zmianą stanu* tejże maszyny *pod wpływem zdań* języka.

Komputer (ang. *compute* – obliczyć) = „obliczacz”

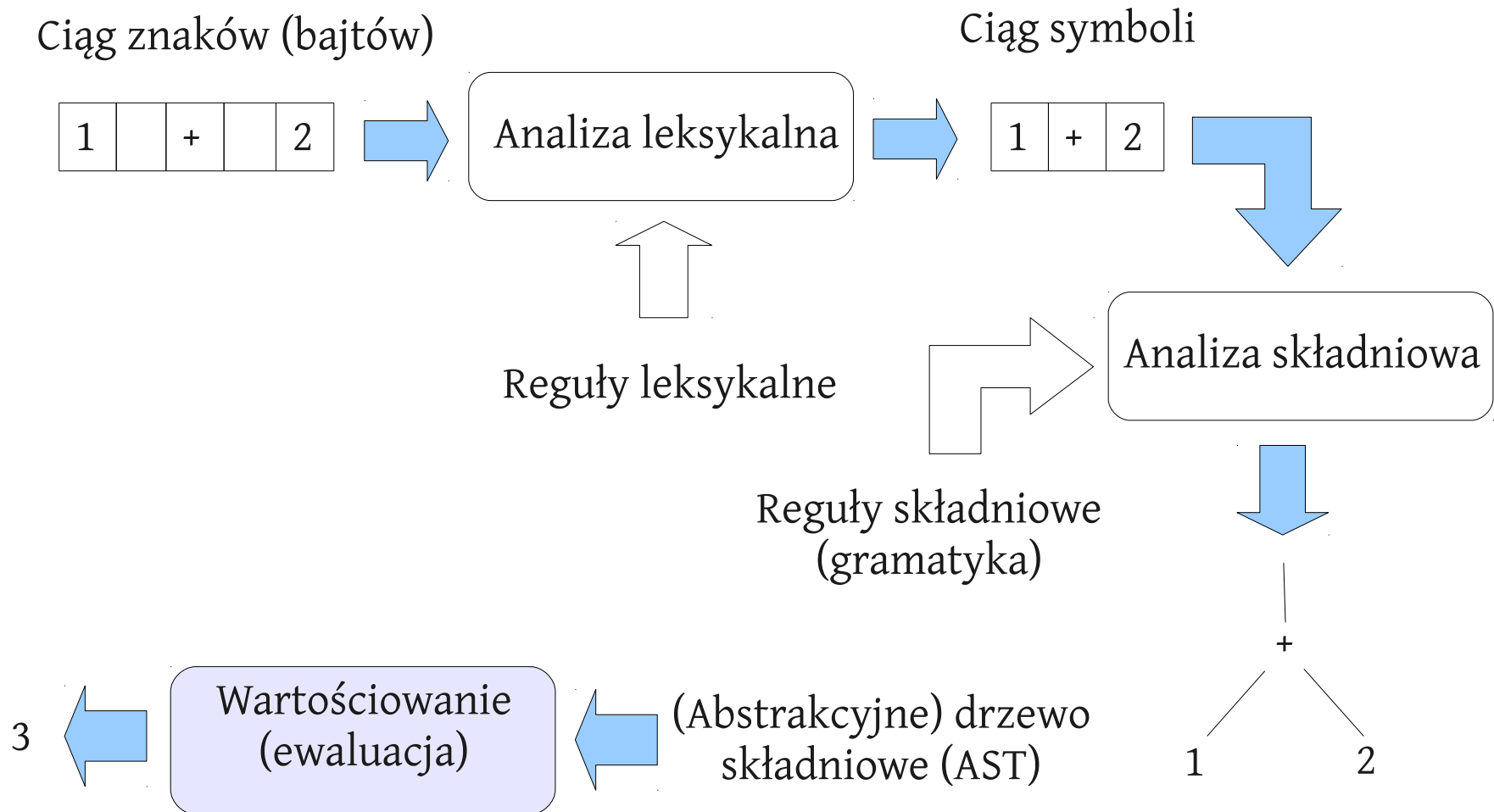
Z perspektywy wysokiego poziomu – języka do komunikacji z automatem – *interpretacja* polega na *obliczaniu wartości zdań* sformułowanych w tymże języku i podawanych na wejście maszyny.

Dwa centralne zagadnienia z tym związane to:

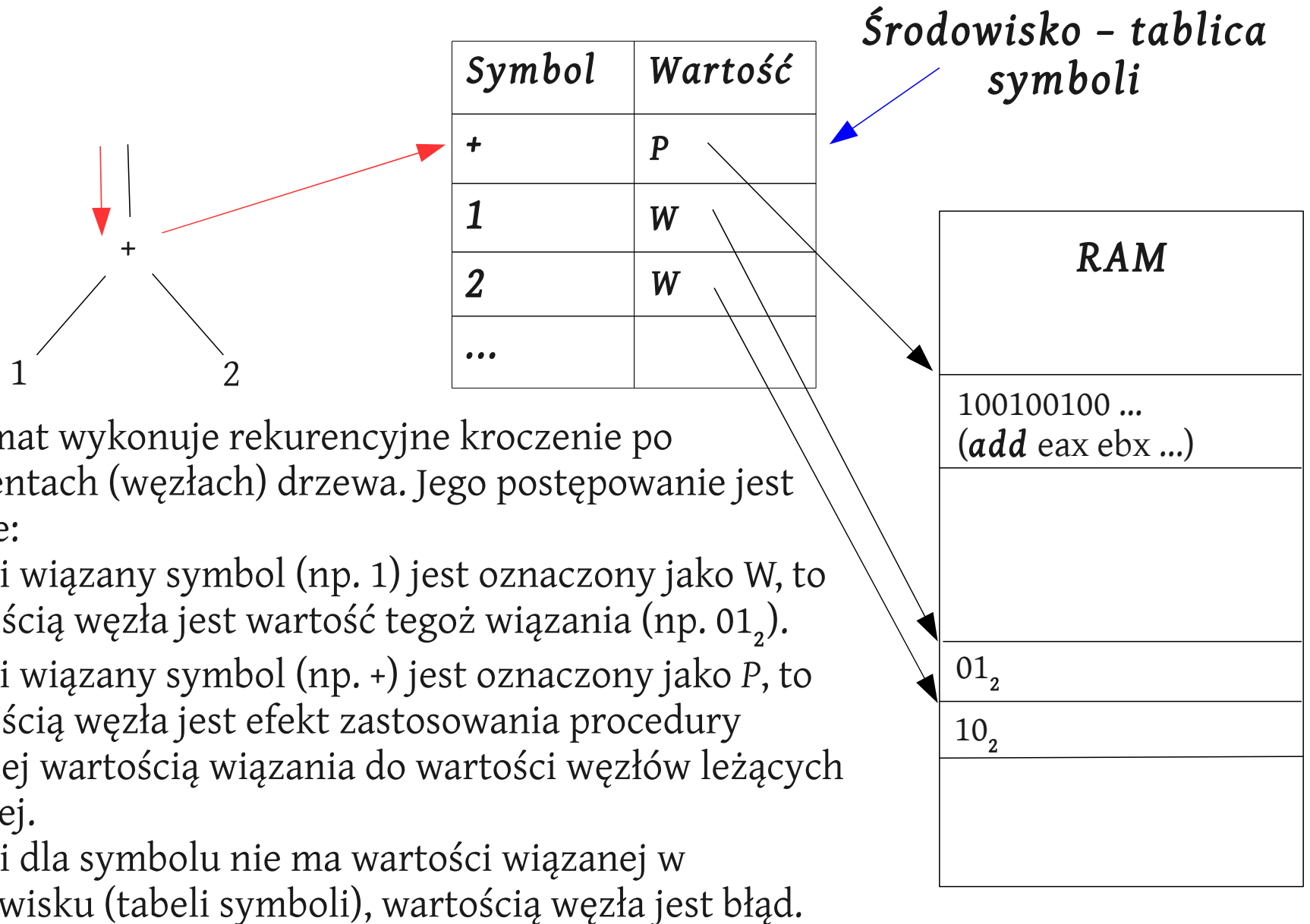
1. Zamiana zdań języka na postać dogodną dla maszyny, w praktyce – na *hierarchiczną strukturę danych*.
2. *Obliczanie wartości* zdania zamienionego na postać hierarchiczną.

Analiza zdania w języku formalnym

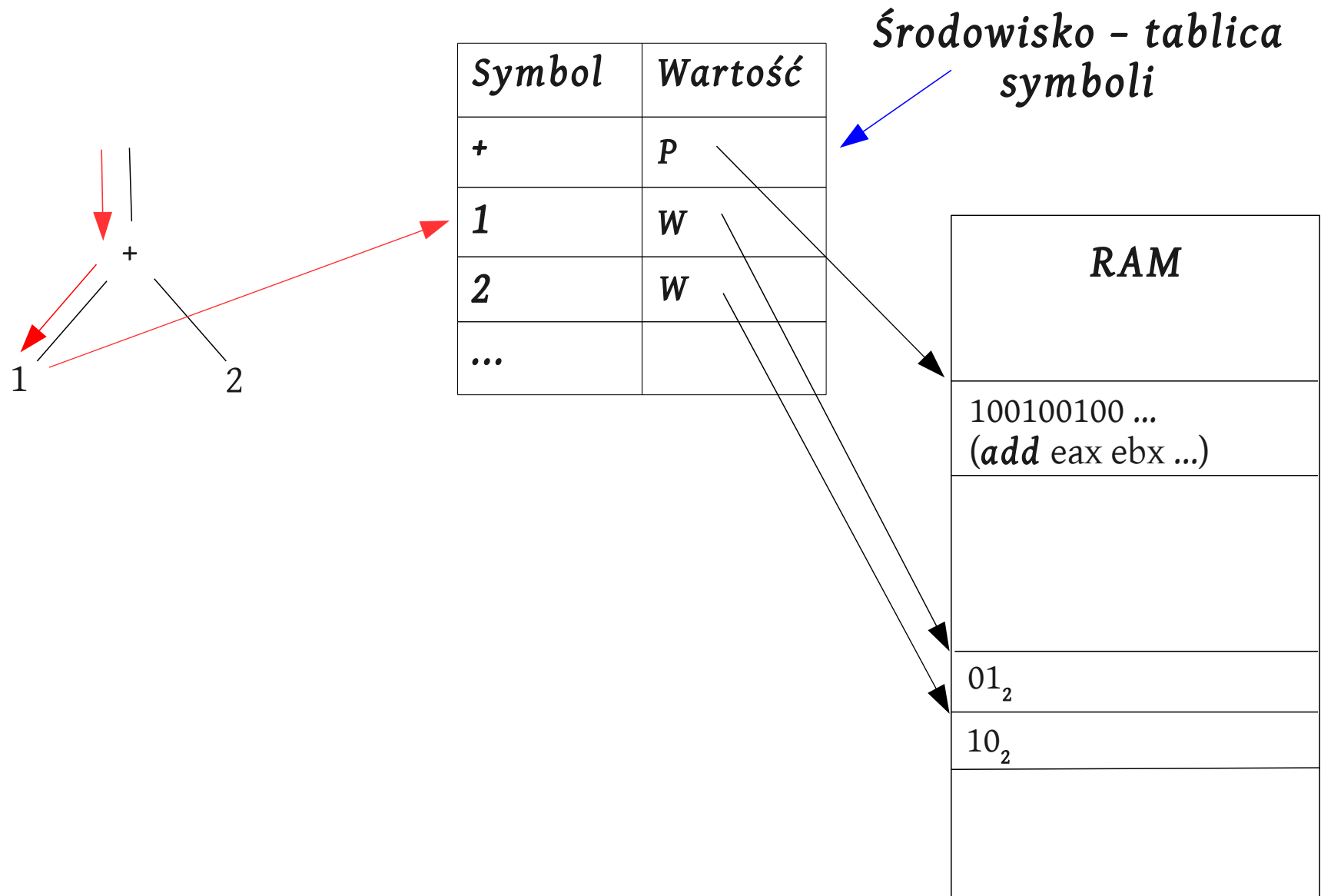
Wyobraźmy sobie, że mamy do czynienia ze zdaniem wyrażonym w języku Python: $1 + 2$



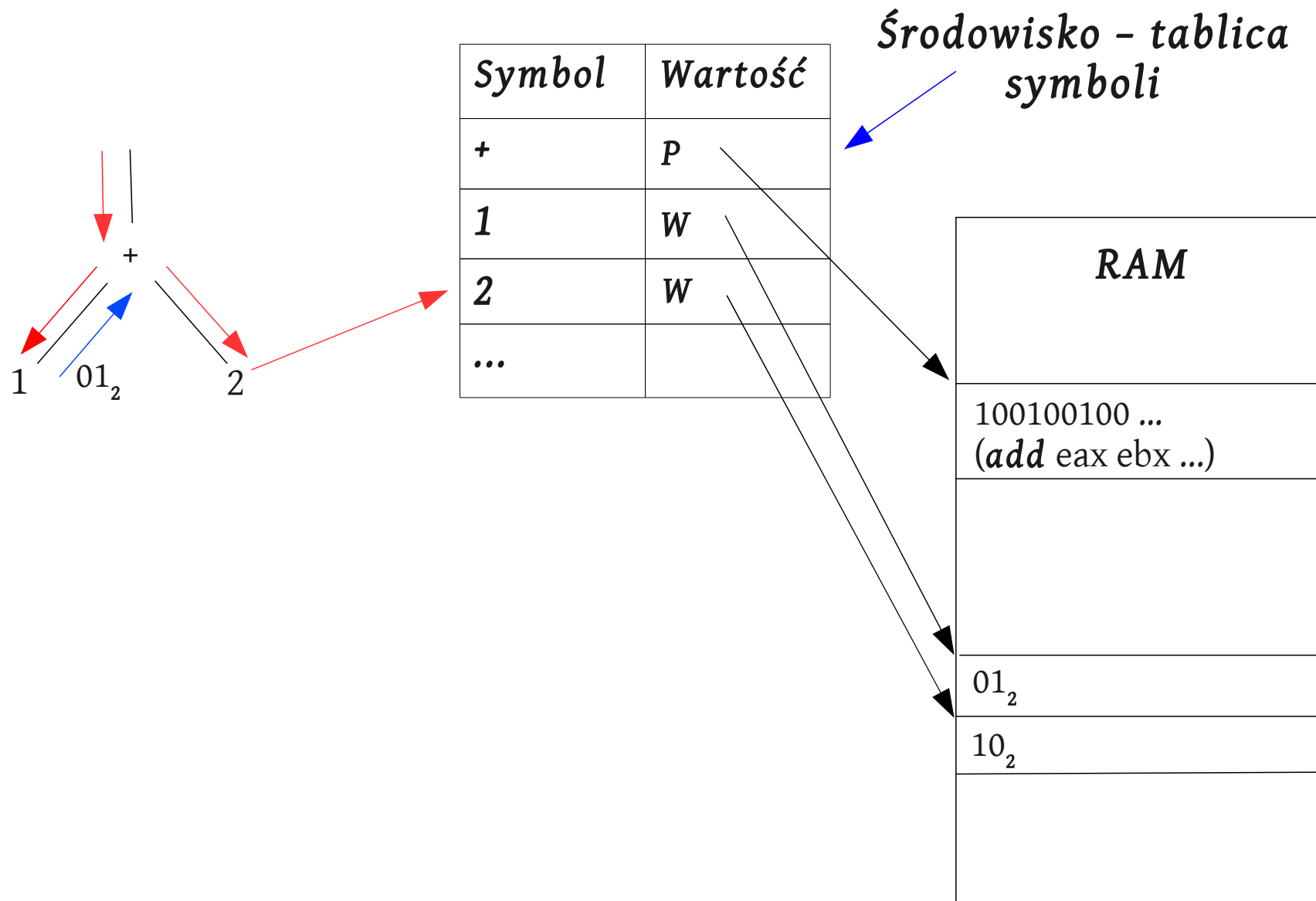
Wartościowanie drzewa składniowego



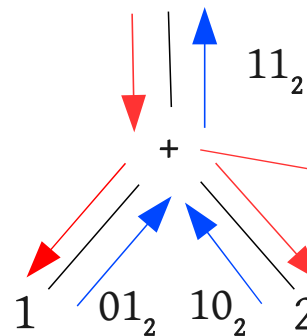
Wartościowanie drzewa składniowego



Wartościowanie drzewa składniowego



Wartościowanie drzewa składniowego



<i>Symbol</i>	<i>Wartość</i>
+	<i>P</i>
1	<i>W</i>
2	<i>W</i>
...	

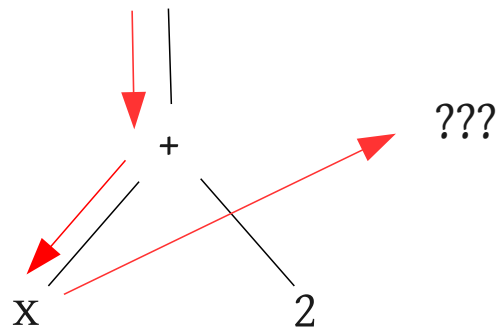
Środowisko - tablica symboli

Możemy teraz powiedzieć, że wartością wyrażenia jest 3 (11_2). Zapiszemy to w postaci:

$$1 + 2 ==> 3$$

RAM
100100100 ... (<i>add</i> eax ebx ...)
01_2
10_2

Wartościowanie wyrażenia $x + 2$



<i>Symbol</i>	<i>Wartość</i>
$+$	P
1	W
2	W
\dots	

Środowisko - tablica symboli

RAM

100100100 ...
(*add* eax ebx ...)

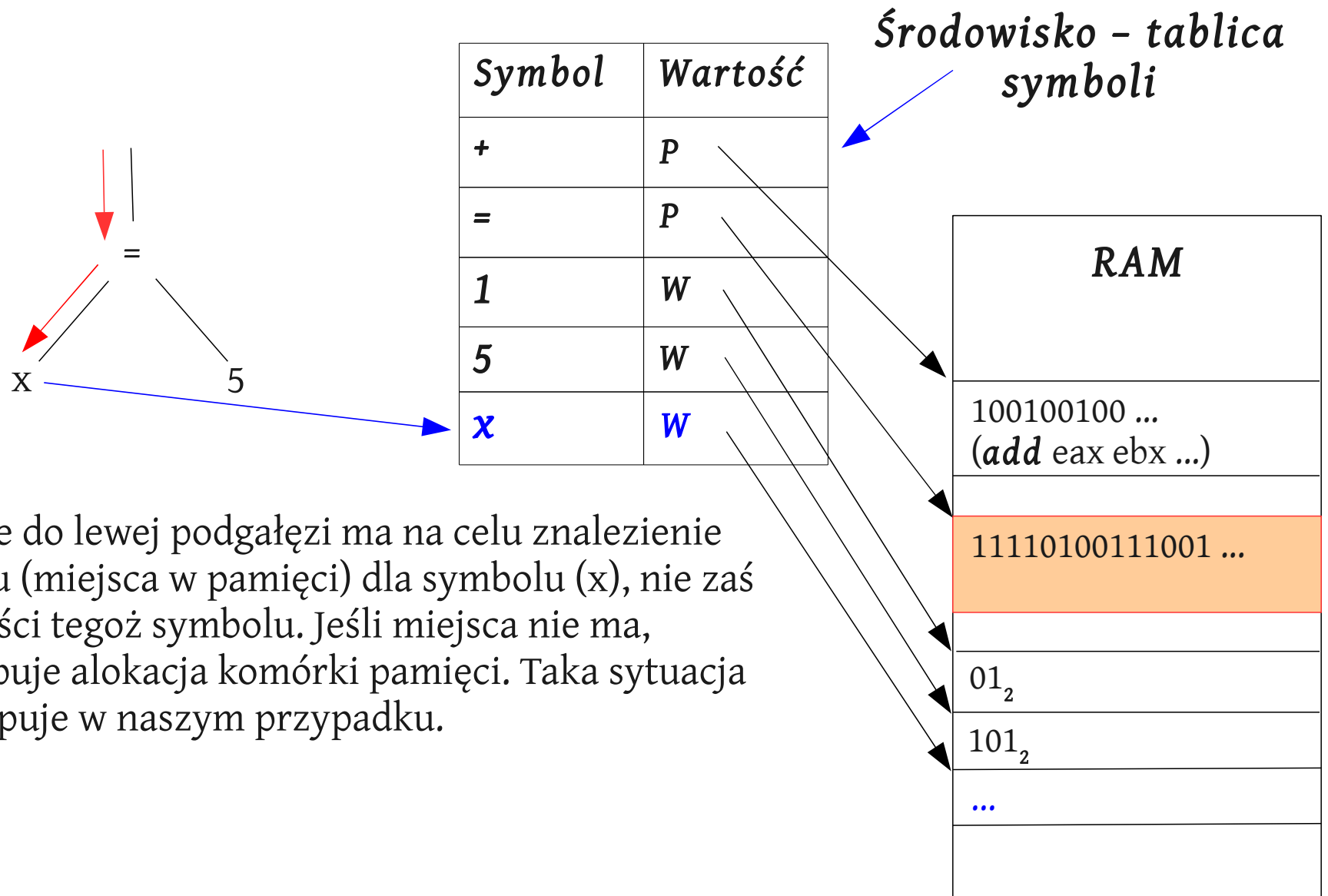
01_2

10_2

W tej sytuacji wartością wyrażenia $x + 2$ jest błąd.

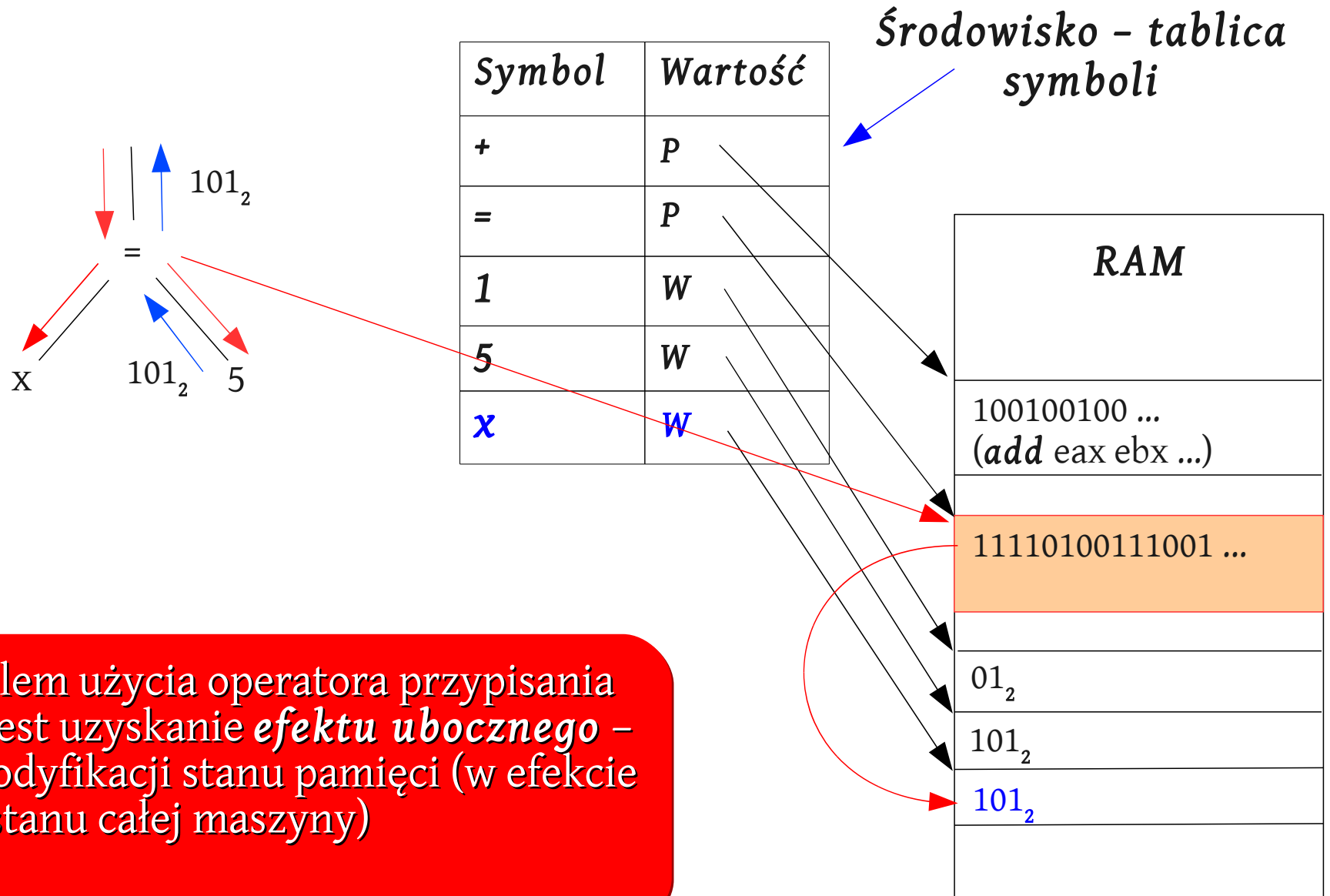
$x + 2 \Rightarrow$ błąd

Operator przypisania $x = 5$



Zejście do lewej podgałęzi ma na celu znalezienie adresu (miejsca w pamięci) dla symbolu (x), nie zaś wartości tegoż symbolu. Jeśli miejsca nie ma, następuje alokacja komórki pamięci. Taka sytuacja występuje w naszym przypadku.

Operator przypisania $x = 5$



- Celem użycia operatora przypisania = jest uzyskanie *efektu ubocznego* – modyfikacji stanu pamięci (w efekcie – stanu całej maszyny)

Operator przypisania

Większość popularnych języków programowania, określanych jako *imperatywne*, „z lubością” wykorzystuje operator przypisania. Niestety, przekłada się to na obserwowalną jakość oprogramowania tworzonego przez programistów.

Użycie operatora przypisania wprowadza do języków programowania i tworzonych z ich wykorzystaniem programów (co najmniej) dwa natychmiastowe *FATALNE* efekty.

Trudny do ogarnięcia stan

Wyobraźmy sobie komputer, który ma pamięć złożoną z N komórek, w których zapisujemy wartości należące do zbioru $\{0, 1\}$. Ilość możliwych stanów takiego układu wynosi 2^N .

Ile wynosi N w przypadku przeciętnego komputera ?

$$\begin{aligned} N &= 2\text{GB} \\ &= 2 * 1024\text{MB} \\ &= 2 * 1024 * 1024 \text{ kB} \\ &= 2 * 1024 * 1024 * 1024 \text{ B} \\ &= 2 * 1024 * 1024 * 1024 * 8 = 2 * 1024 * 1024 * 1024 * 8 \text{ bitów} \\ &= 17179869184 \text{ bitów} \end{aligned}$$

W takim razie ilość stanów, w jakich może znaleźć się nasza maszyna wynosi $S = 2^{17179869184}$

Które z tych stanów można uznać za prawidłowe ? Jak to ocenić ?

Jest nawet gorzej. Istota dowodu Turinga polegała na stwierdzeniu, że jest to w ogólności niemożliwe w sensie matematycznym, przy założeniu nieskończonej mocy obliczeniowej i nieskończonego czasu.

Czas

W matematyce czas nie istnieje. Wyrażenia $\sin 0$, $\cos \pi$, $x \rightarrow 2x + 3$ zawsze mają tę samą wartość. Popatrzmy, co powoduje użycie „niewinnego” operatora $=$.

t_1	$x + 2 \Rightarrow \text{błąd}$
t_2	$x = 5 \Rightarrow 5$
t_3	$x + 2 \Rightarrow 7$
t_4	$x = 3 \Rightarrow 3$
t_5	$x + 2 \Rightarrow 5$
t	

W trzech różnych chwilach t_1, t_3, t_5 *to samo wyrażenie przyjmuje różne wartości.*

Czy posługiwanie się formalizmem podlegającym takim *anomaliom* może być skutecznym sposobem opisywania złożonych problemów świata? Czy postawilibyśmy stopę na Księżycu, gdyby *matematyka* zachowywała się w ten sposób.

Kilka innych przydatnych definicji

1. **Zmienna** – symbol, który charakteryzuje się tym, że wartość jego wiązania (w tabeli symboli) może zmieniać się przy zastosowaniu operatora przypisania =.
2. **Obiekt** – „**Byt** posiadający **tożsamość**, to znaczy – **jednoznacznie odróżnialny** od wszystkich innych bytów.”

Na podstawie: Harold Abelson, Gerald J. Sussman, Struktura i interpretacja programów komputerowych, WNT 2002

3. **Typ**:

- a. **Zbiór więzów i ograniczeń** nałożonych na **dane (obiekty)** w celu **zagwarantowania** ich **poprawnego użycia**. Chodzi o użycie w zestawieniu z funkcjami (operatorami) w zdaniach pewnego języka (zwykle formalnego).
- b. **Zbiór danych (obiektów)** charakteryzujących się tym, że wszystkie one **spełniają** ten sam zestaw **więzów i ograniczeń**.

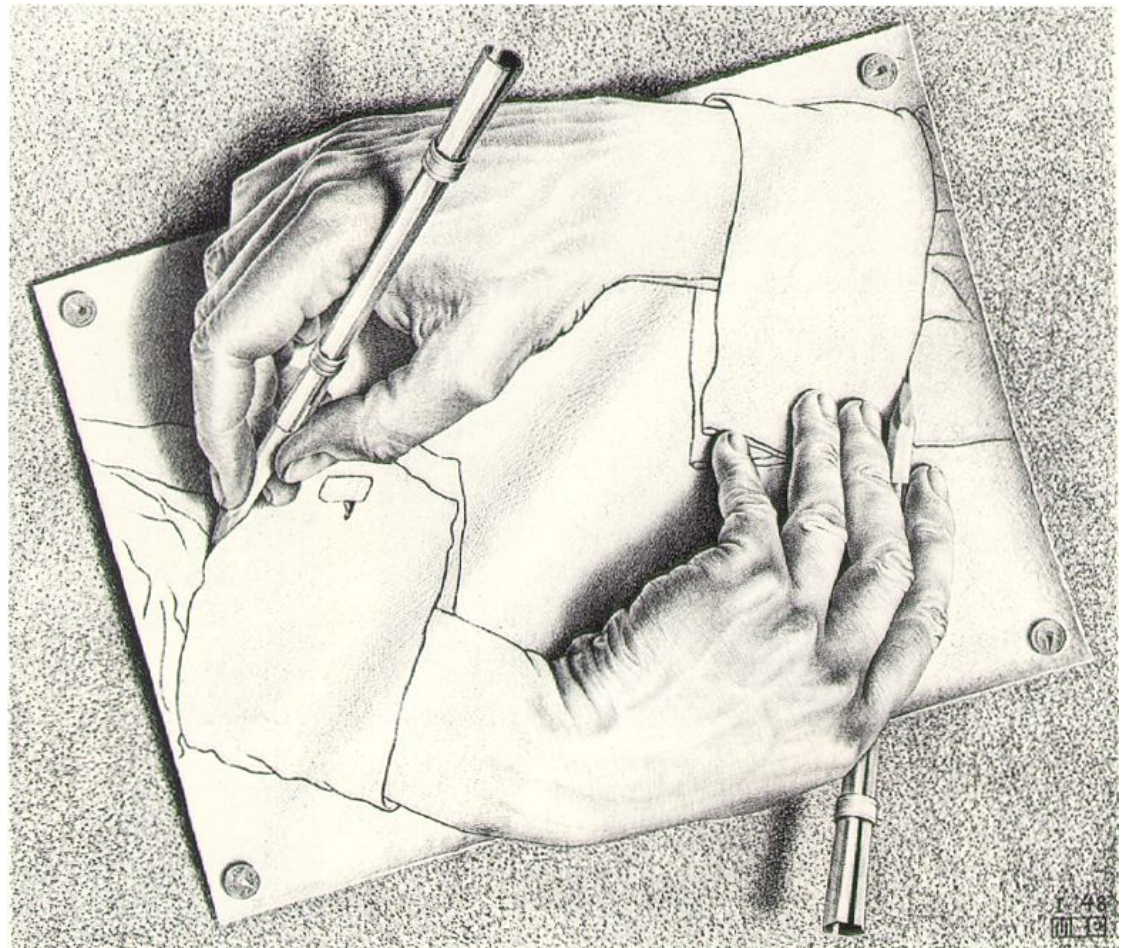
Powyższe komplementarne definicje typu zostały wypracowane przez znanych matematyków amerykańskich: **Danę Scotta i Christophera Strachey**.

Rekurencja i iteracja

„Iteracja jest rzeczą ludzką, rekurencja – boską.” - *L (Lawrence) Peter Deutsch*, hacker, twórca Ghostscripta (interpreter/przeglądarka plików Postscript i PDF).

Drawing Hands, litografia (1948).

Maurits Cornelis Escher (1898 – 1972), holenderski grafik znany ze swoich matematycznych inklinacji.



Rekurencja

(łac. *recurrere* – przybiec z powrotem). Zjawisko występujące, gdy **definicja** pewnego bytu **odwołuje się do** tegoż **definiowanego bytu**. Rodzaj intelektualnego „zapętlenia”. W praktyce przyjmuje ono następujące kształty:

1. W matematyce i informatyce (ang. *computer science*) rekurencja polega na **wywoływaniu funkcji** (procedury) **przez samą siebie**.
2. Rekurencja może również polegać na tym, że złożony typ danych (struktura) T określa jedną ze swoich składowych jako odwołanie (referencję, wskaźnik) do bytu (obiektu, wartości) o typie T. Mamy tutaj do czynienia z **rekurencyjnymi typami danych**.

Silnia

Funkcja określona na zbiorze liczb naturalnych N , zdefiniowana następująco:


$$n! = 1 \quad \text{dla } n = 0$$

$$n! = n (n - 1)! \quad \text{dla } n > 0$$

Wykorzystywana szeroko w m. in. kombinatoryce (i innych dyscyplinach wchodzących w skład dziedziny dyskretnej) oraz teorii liczb.

Silnia – implementacja

```
static int silnia(int n) {  
    if (n == 0) {  
        return 1;  
    }  
    else {  
        return n * silnia (n - 1);  
    }  
}
```




```
static void main(String[] args) {  
    System.out.println("Silnia z " + 0 + " wynosi " + silnia(0));  
    System.out.println("Silnia z " + 3 + " wynosi " + silnia(3));  
}
```

Silnia – implementacja – narzędzia

```
static void wypisz_silnie(int n) {  
    System.out.println("Silnia z " + n + " wynosi " + silnia(n));  
}
```

```
static void wypisz_silnie_z_zakresu(int start,  
                                     int koniec) {  
    if(start <= koniec) {  
        wypisz_silnie(start);  
        wypisz_silnie_z_zakresu(start+1, koniec);  
    }  
}
```



```
static void main(String[] args)  
{  
    wypisz_silnie_z_zakresu(0, 10);  
}
```

```
Silnia z 0 wynosi 1  
Silnia z 1 wynosi 1  
Silnia z 2 wynosi 2  
Silnia z 3 wynosi 6  
Silnia z 4 wynosi 24  
Silnia z 5 wynosi 120  
Silnia z 6 wynosi 720  
Silnia z 7 wynosi 5040  
Silnia z 8 wynosi 40320  
Silnia z 9 wynosi 362880  
Silnia z 10 wynosi 3628800
```

Silnia – jak to działa

`silnia(5) =`

`5 * silnia(4) =`

`5 * 4 * silnia(3) =`

`5 * 4 * 3 * silnia(2) =`

`5 * 4 * 3 * 2 * silnia(1) =`

`5 * 4 * 3 * 2 * 1 * silnia(0) =`

`5 * 4 * 3 * 2 * 1 * 1 = 120`

Powyższe rozumowanie „przeprowadziło” nas od *rekurencyjnej* definicji funkcji $n!$ oraz procedury *silnia* do wersji *jawnej*.

$$n! = \prod_{k=1}^n k \quad \forall n \in \mathbb{N}$$

Zachodzi to przy koniecznym i przyjętym w matematyce założeniu, że iloczyn zera liczb (pusty) jest równy 1.

Iteracja

(łac. *iteratio*) – powtarzanie operacji, szczególnie z każdorazowym przekształceniem w analogiczny sposób.

Procedura

```
void wypisz_silnie_z_zakresu(int start,  
                             int koniec) { ... }
```

jest przykładem *algorytmu iteracyjnego*, ponieważ jego celem jest **wielokrotne powtórzenie** operacji wypisywania wartości procedury *silnia* na ekran, dla wartości parametru formalnego *n* należących do podanego zakresu.

Procedura ta jest **zrealizowana** z wykorzystaniem **rekurencji**.

Iteracja

- Wielu programistów milcząco przyjmuje, że *iteracja* jest równoznaczna z realizacją powtarzania operacji z wykorzystaniem *instrukcji skoku (pętli) i zmiany stanu*. Vide L Peter Deutsch.
- Jest to rozumowanie *nieściste* ! Wielokrotne powtarzanie (iterowanie) może być zrealizowane z użyciem zarówno instrukcji skoku, jak i *rekurencji*.

Z uwagi jednak na rozpowszechnienie tego poglądu, wygodnie jest przyjąć rozpowszechnioną choć nieścistą terminologię. Od tej pory mówiąc „*iteracyjny*” mamy na myśli – *zrealizowany z wykorzystaniem zmiany stanu i instrukcji skoku* (pętli). Chyba, że zostanie to jawnie określone inaczej.

Problemy rekurencji

Są dwa zasadnicze:

1. Rekurencyjna postać funkcji może być mało czytelna dla człowieka nie przyzwyczajonego do myślenia w ten sposób o problemach. Fakt ten może utrudnić analizę algorytmu, ocenę jego poprawności. Bardzo częstym błędem popełnianym przy tworzeniu procedur rekurencyjnych jest pominięcie warunku prowadzącego do zatrzymania procesu obliczeniowego (**warunek „stopu”**) lub jego nieumiejętne użycie. W wyniku tego proces obliczeniowy może „wpaść” w nieskończoną iterację. Objawi się to komunikatem **STACK-OVERFLOW-ERROR** z uwagi na zjawisko opisane w punkcie 2.
2. **PRYMITYWNA** realizacja rekurencji w obecnie stosowanych językach programowania – oparta na **skończonym stosie** – powoduje, że zwykle mamy do czynienia z ograniczeniem głębokości drzewa wywołań rekurencyjnych do kilkuset lub kilku tysięcy. Iteracja zrealizowana **poprawnie** z wykorzystaniem rekurencji jest więc narażona na **STACK-OVERFLOW-ERROR** dla nietrywialnych rozmiarów problemu, który adresuje.

Dojrzała realizacja rekurencji

Dojrzałe języki programowania oferują przeźroczystą (ang. *transparent*) konwersję wywołań rekurencyjnych na postać, która nie wykorzystuje fatalnego stosu.

Technika, o której mowa, stosowana jest na etapie kompilacji i nosi nazwę **optymalizacji rekurencji krańcowej** (dosł. „ogonowej” z ang. *tail recursion optimization*). Języki, które tę technikę oferują:

§ Dialekty języka **LISP**, w szczególności

- Scheme – zawsze, domyślnie
- Common Lisp – przy założeniu niektórych ustawień optymalizacji
- Clojure – poprzez jawne wywołanie (**recur** ...) pomimo braku tej techniki w maszynie wirtualnej Javy (JVM)

§ Niektóre „pochodne” języka ML, np. **Ocaml**

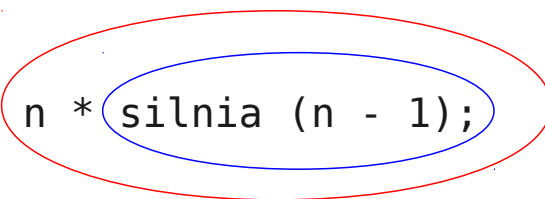
Doskonały język funkcyjny **Haskell** jest językiem wyposażonym w techniki **odraczania wykonania** (ewaluacji) **wyrażeń** (ang. *lazy evaluation*). Z tego powodu opisywane problemy z rekurencją w ogóle go nie dotyczą, bo stos nie jest wykorzystywany w sposób tradycyjny. W **Clojure** późne wykonanie jest również dostępne, ale jego zastosowanie musi być jawnie zadekretowane, poprzez użycie odpowiednich form składniowych.

Optymalizacja rekurencji krańcowej

Warunkiem koniecznym zastosowania tej techniki przez kompilator jest taka postać procedury rekurencyjnej, w której **wartość wywołania rekurencyjnego** jest jednocześnie **wartością procedury**. Innymi słowy – po obliczeniu wartości wywołania rekurencyjnego wartość ta nie jest dalej używana do obliczeń, lecz jest natychmiast zwracana jako wartość procedury. Stąd nazwa „krańcowa” – nic więcej nie robimy, **zejście (wywołanie) rekurencyjne** jest **ostatnim etapem** wyznaczania wartości wywołania procedury.

Silnia w postaci

```
int silnia(int n)
{
    if (n == 0) {
        return 1;
    }
    else {
        return n * silnia (n - 1);
    }
}
```



nie posiada rekurencji krańcowej. Wartość wyrażenia `silnia (n - 1)` jest użyta do wyznaczenia wartości wyrażenia `n * silnia (n - 1)`.

Silnia z rekurencją krańcową

```
int silnia(int n, int wartosc) {  
    if(n == 0) {  
        return wartosc;  
    }  
    else {  
        return silnia(n-1, wartosc * n);  
    }  
}
```

`int x = silnia(10, 1);` ==> 3628800 (10!), proszę zwrócić uwagę na dodatkowy konieczny parametr równy zawsze 1.

Teraz wartość wywołania rekurencyjnego `silnia(n-1, wartosc * n)` jest wartością wywołania procedury `silnia`. Wywołanie `silnia(n-1, wartosc * n)` jest krańcowe, po nim nie ma już żadnych innych obliczeń, jego wartość jest natychmiast zwracana.

Kompilator języka Java (`javac`) nie posiada optymalizacji rekurencji krańcowej. Technika ta była dotychczas odrzucana przez twórców kompilatorów języków Java, C, C++, Ada z uwagi na brak automatycznego zarządzania pamięcią (szczególnie przestrzenią stosu) oraz ogólny, niskopoziomowy charakter tych języków.

Co zrobiłby dojrzały kompilator javac ?

```
static int silnia(int n, int wartosc) {  
    int tmp;  
    START:  
    if(n == 0) {  
        return wartosc;  
    }  
    else {  
        tmp = n;  
        n = n - 1;  
        wartosc = wartosc * tmp;  
        goto START;  
    }  
}
```

Postać ta opiera się na zastosowaniu instrukcji skoku (**goto**) oraz na **modyfikacji stanu** procesu obliczeniowego przy użyciu **operatora przypisania** (=) przykładanego do symboli **zmiennych** *n*, *wartosc* i *tmp*.

GOTO Considered Harmful ...

„Nieostrożne użycie polecenia goto skutkuje natychmiastowymi konsekwencjami; szalenie trudne staje się ustalenie zbioru koordynatów, przy użyciu których można byłoby określić kierunek zmian procesu (obliczeniowego). [...] Polecenie goto jest zbyt prymitywne, jest zaproszeniem do bałaganienia w kodzie programu.”

Edsger Dijkstra (March 1968). „Go To Statement Considered Harmful”. Communications of the ACM 11 (3): 147–148. doi:10.1145/362929.362947

```
static int silnia(int n,  
                 int wartosc) {  
    while(true) {  
        if(n == 0) {  
            return wartosc;  
        }  
        else {  
            wartosc = wartosc * n;  
            n = n - 1;  
        }  
    }  
}
```

Edsger Dijkstra (ur. 11 maja 1930 w Rotterdamie, zm. 6 sierpnia 2002 w Neunen). Matematyk, jeden z pionierów informatyki, twórca algorytmu znajdującego najkrótszą ścieżkę pomiędzy węzłami grafu (**algorytm Dijkstry**), wynalazca **semafora** w prog. współbieżnym, laureat Nagrody Turinga (1972).

Dwa style programowania

1. **Imperatywny.** Polega na wykorzystaniu *zmiennych, modyfikacji stanu* procesu obliczeniowego w sposób *jawny* przy użyciu *operatora przypisania* oraz na wykorzystaniu *instrukcji skoku* (w tym również pętli).

Język maszynowy, assembler, Fortran, Algol, C, Ada, C++, Java, C#, Python, Ruby, ...

2. **Deklaratywny.** Polega na posługiwaniu się wyrażeniami, które posiadają *wartość*. Jawna modyfikacja stanu nie występuje. Deklaratywny język programowania służy do wyrażania raczej tego, co chcemy, aby maszyna dla nas wykonała, nie zaś do jawnego przechodzenia pomiędzy stanami.

HTML, SQL, PostScript i PDF, TeX, Lisp, Haskell, Erlang, ML i Ocaml

Programowanie funkcyjne

Jest to **wariant stylu deklaratywnego**. W odróżnieniu od takich języków deklaratywnych jak HTML lub SQL, które są językami specjalizowanymi, dziedzinowymi, języki funkcyjne służą do budowy różnorodnych rozwiązań algorytmicznych. Są to języki **ogólnego przeznaczenia**.

Lisp: Common Lisp, Clojure, Scheme
Erlang
Haskell
ML, Ocaml

Programowanie funkcyjne polega na jednoczesnym zastosowaniu następujących programistycznych „środków wyrazu”:

1. **Rekurencja**, w tym rekurencja krańcowa.
2. Instrukcje warunkowe – **if**, **case** (**cond**).
3. **Procedury** pozbawione efektów ubocznych (modelujące funkcje w matematyce) w szczególności **domknięcia** (ang. *closure*) jako obiekty pierwszego rzędu (ang. *first-class objects*).

Co to są obiekty pierwszego rzędu ?

Kiedy obiekty pewnego typu (rodzaju) podlegają wszystkim standardowym operacjom:

- tworzeniu,
- usuwaniu,
- przekazywaniu do procedur jako wartości parametrów formalnych (operandy),
- zwracaniu jako wartości wywołań procedur

wówczas o obiektach takich mówimy, że są to **obiekty pierwszego rzędu** (ang. *first-class objects*). Przykład: liczby całkowite, łańcuchy znaków (String).

```
int dodaj(int a, int b) {  
    return a + b;  
}
```

```
public String toString() {  
    return "[" + this.x + ", " + this.y + "];  
}
```

Powstaje nowy obiekt
zwracany przez procedurę

Procedury jako obiekty pierwszego rzędu

W wielu językach programowania procedury są obiektami pierwszego rzędu. Do języków tych zaliczyć można wszystkie wymienione wcześniej języki funkcyjne. Niektóre języki imperatywne, np. Python, Ruby również posiadają tę cechę.

W Javie cecha ta jest dostępna w oparciu o anonimowe klasy wewnętrzne. Gdyby jednak była wspierana w nieco bardziej naturalny sposób, mogłoby to wyglądać następująco:

```
Function<int, int> utworzSumator(final int n) {
```

```
    int sumator(int x) {  
        return x + n;  
    }
```

Procedura jest tworzona i
zwracana

```
    return sumator;  
}
```

```
Function<int, int> dodajTrzy = utworzSumator(3);
```

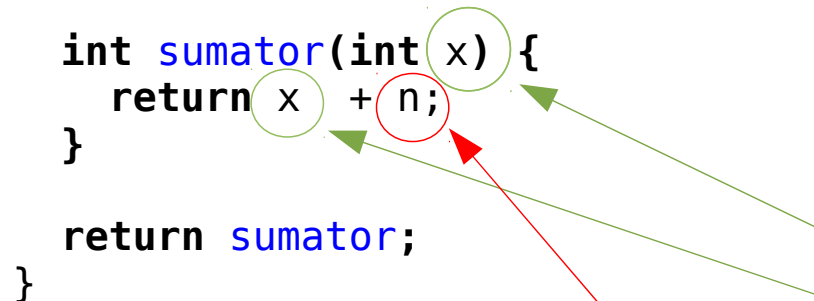
```
int wynik = dodajTrzy(10); ==> 13
```

Symbole związane i wolne

Przyjrzyjmy się raz jeszcze problemowi:

```
Function<int, int> utworzSumator(final int n) {
```

```
    int sumator(int x) {  
        return x + n;  
    }  
    return sumator;  
}
```



x jest **związany** wewnątrz procedury sumator.

n jest **wolny** wewnątrz Procedury sumator.

Zarówno n jak i x są **związane** wewnątrz procedury utworzSumator.

Symbol związany – taki, który można zmienić w danym kontekście (najczęściej w procedurze) nadając mu inną nazwę w miejscach wszystkich wystąpień nie zmieniając znaczenia wyrażień.

- Symbol wolny – takiego w danym kontekście nie można zmienić, ponieważ spowodowałoby to zmianę znaczenia wyrażień, w szczególności – powstanie błędów.

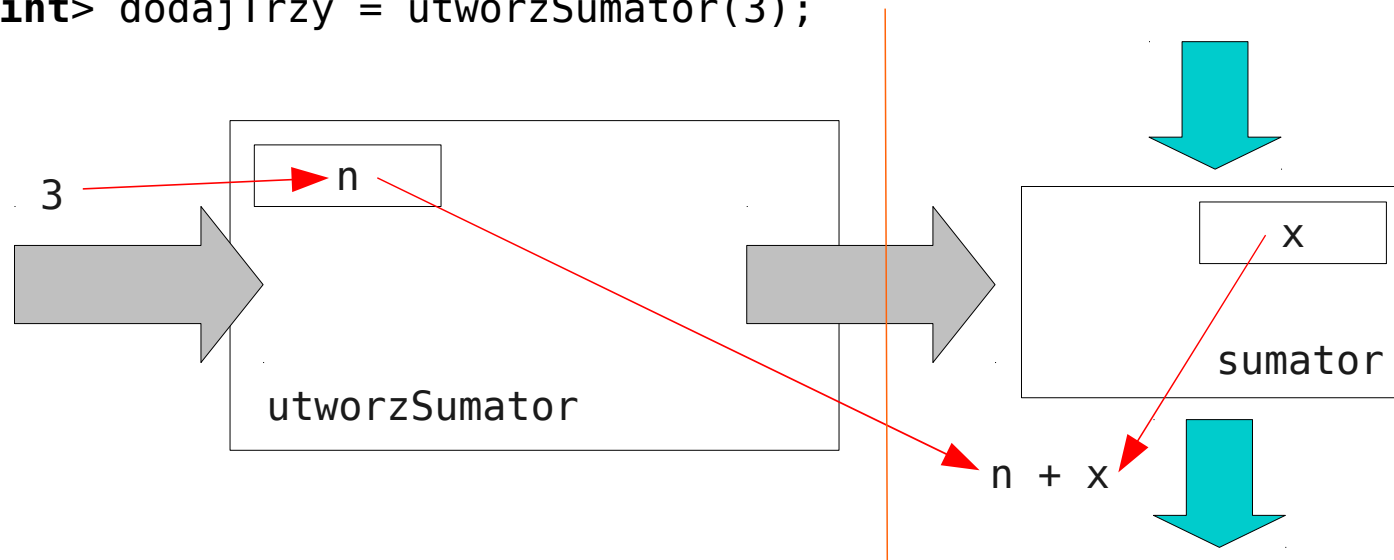
Domknięcie (ang. *closure*)

Procedura pierwszego rzędu (ang. *first-class procedure*), wewnątrz której znajdują się **wolne symbole**, nazywana jest **domknięciem**. Co takiego „domyka”? Domyka i „zabiera ze sobą” środowisko zewnętrzne, w którym została utworzona, reprezentowane przez tabelę wolnych symboli.

```
Function<int, int> utworzSumator(final int n) {  
  int sumator(int x) {  
    return x + n;  
  }  
  return sumator;  
}
```

Utworzona procedura sumator istnieje poza procedurą utworzSumator jako osobny byt.

```
Function<int, int> dodajTrzy = utworzSumator(3);
```



Styl funkcyjny w Javie - ograniczenia

Programiści chcący programować funkcyjnie z wykorzystaniem języka Java muszą uwzględnić fakt, że w języku tym **procedury** (wł. *metody*) **nie są obiektami pierwszego rzędu**.

Możliwe jest natomiast zamodelowanie tego brakującego elementu poprzez użycie typów (interfejsów lub klas abstrakcyjnych) których obiekty będą reprezentantami procedur. O obiektach takich mówimy, że są to **funktory**.

```
Runnable obj = new Runnable() {  
    @Override  
    public void run() {  
        Kod-do-wykonania ...  
    }  
};
```

```
Function<void, void> obj = fn() {  
    Kod-do-wykonania ...  
};
```



Tak mogłoby to wyglądać w **lepszej** Javie.

Runnable jest **typem** (dokładnie – *interfejsem*) należącym do biblioteki standardowej języka Java. Obiekt reprezentowany przez symbol `obj` jest natomiast **funktorem**. Reprezentuje kod wykonywalny znajdujący się w **przesłoniętej** (stąd adnotacja **@Override**) **metodzie** `run`. Reprezentuje niemal dokładnie tak, jak reprezentowałaby to procedura pierwszego rzędu, gdyby możliwość definiowania takowych procedur była w Javie dostępna.

Funktory w Javie

Przyjrzyjmy się temu zagadnieniu przez pryzmat realizacji przykładu z sumatorem z jednego z poprzednich slajdów:

```
public class Program {  
  
    public static Functor.Unary<Integer, Integer> utworzSumator(final int n) {  
        return new Functor.Unary<Integer, Integer>() {  
            @Override  
            public Integer call(Integer x) {  
                return x + n;  
            }  
        };  
    }  
  
    public static void main(String[] args) {  
        Functor.Unary<Integer, Integer> dodajTrzy = utworzSumator(3);  
        int x = dodajTrzy.call(10);  
        System.out.println("Wynik dodawania to " + x);  
    }  
}
```

Integer jest synonimem typu *int*.
Ze względów implementacyjnych,
a więc mało interesujących,
wprowadzono dwie nazwy.

Przesłonięta metoda *call* jest „sercem” rozwiązania – jest kodem, który zostanie „uruchomiony” w momencie wywołania funktora. Proszę zwrócić uwagę na fakt, że nasz funktor jest bezstanowy – jest modelem funkcji matematycznej $x \rightarrow x + n$.

Ale co to jest Functor.Unary ?

Konstrukcje z poprzedniego slajdu powstały w wyniku zastosowania metody „pobożnych życzeń” (ang. *wishful thinking*). Polega ona na podejmowaniu wczesnych decyzji dotyczących tego, co chcę mieć i późnych – jak mam to wykonać.

```
public class Functor {  
    public interface Unary<S, T> {  
        T call(S arg);  
    }  
    ...  
}
```

Funktor jednoargumentowy jest typem (a dokładniej – interfejsem) generycznym, tj. parametryzowanym symbolami S i T. S oznacza typ argumentu, T – typ zwracanej wartości. W przypadku typu sumatora (`Functor.Unary<Integer, Integer>`) symbole S i T przyjmują tę samą wartość: *Integer*.

Jedyna metoda tego interfejsu – *call* – jest reprezentantem zarówno ciała procedury (funktora), jak i komunikatem, dzięki któremu świat zewnętrzny może wywoływać tę procedurę (funktor).

Pozostałe funktory

```
public class Functor {  
    ...  
    public interface Binary<S, R, T> {  
        T call(S arg1, R arg2);  
    }  
    public interface Ternary<S, R, V, T> {  
        T call(S arg1, R arg2, V arg3);  
    }  
    public interface Multi<S, T> {  
        T call(S... args);  
    }  
    private Functor() {  
        ;  
    }  
}
```

*Prywatny konstruktor zagwarantuje nam, że klasa **Functor** nie będzie miała ani żadnych obiektów ani nie będzie możliwe jej odziedziczenie.*

Jest tutaj wyłącznie przestrzeń nazw.

Kompozycja (składanie) funkcji

Jedną z najważniejszych technik w programowaniu funkcyjnym pozwalającą na budowanie efektywnych i poprawnych programów jest kompozycja funkcji.

Definicja 1. Złożeniem funkcji $f: X \rightarrow Y$ oraz $g: Y \rightarrow Z$ jest funkcja $h: X \rightarrow Z$ taka, że $h(x) = g(f(x))$ dla każdego x należącego do zbioru X .

Zapisujemy $h = g \circ f$ i dalej możemy zapisać $h(x)$ albo $(g \circ f)(x)$.

Zadanie. Zaimplementować złożenia funkcji w języku Java z wykorzystaniem biblioteki funktorów.

Operator *compose* – implementacja

Na wstępie poczynimy pewne założenia:

- Nasz operator zostanie zdefiniowany jako procedura (statyczna) wewnątrz przestrzeni nazw wyznaczonej przez klasę *Functor*.
- Pierwsza wersja będzie dotyczyła składania funkcji jednoargumentowych (unarnych).

```
public class Functor {  
  
    static <X, Y, Z> Functor.Unary<X, Z> compose(final Functor.Unary<Y, Z> g,  
                                                final Functor.Unary<X, Y> f)  
    {  
        return new Functor.Unary<X, Z>() {  
            @Override  
            public Z call(X arg) {  
                return g.call(f.call(arg));  
            }  
        };  
    }  
}
```

Operator *compose* – przykład użycia

Modelujemy prostą sytuację, gdy $f: \text{double} \rightarrow \text{double}$ oraz $g: \text{double} \rightarrow \text{double}$, a więc $h: \text{double} \rightarrow \text{double}$ oraz zakładamy, że

$$f = x \rightarrow 2x + 3$$

$g = x \rightarrow x + 1$ (operator inkrementacji, czasem zwany po prostu *inc*).

$$h = g \circ f$$

W takim razie $h(x) = g(f(x)) = (2x + 3) + 1$

```
Functor.Unary<Double, Double> f = new Functor.Unary<Double, Double>() {  
    @Override  
    public Double call(Double x) {  
        return 2 * x + 3;  
    }  
};
```

```
Functor.Unary<Double, Double> g = new Functor.Unary<Double, Double>() {  
    @Override  
    public Double call(Double x) {  
        return x + 1;  
    }  
};
```

```
Functor.Unary<Double, Double> h = Functor.compose(g, f);  
System.out.println(h.call(4.0)); // ==> 12.0
```

Zadanie

Zaimplementować przeciążenie procedury ***compose*** dla złożień funktorów:

- unarnego z binarnym
- binarnego z unarnym
- binarnego z binarnym

Umiejętnie określić typy parametrów. Przetestować !!!.

Operator complement

W programach (nie tylko) sztucznej inteligencji mamy do czynienia z predykatami, tj. wyrażeniami przyjmującymi wartość logiczną: **prawda**, **fałsz**. Co za tym idzie, pojawiają się w sposób naturalny funkcje, które wywoływane pełnią rolę predykatów w różnych miejscach algorytmów.

W języku Java typem danych służącym do oznaczania wartości logicznych jest typ **boolean** (od nazwiska **George Boole (1815 - 1864)**, wybitnego matematyka brytyjskiego, twórcy słynnej **algebry Boole'a** – stanowiącej podstawę działania elektronicznych układów cyfrowych, w tym komputerów).

Niech będzie dany predykat $p: X \rightarrow \text{boolean}$

Operator **complement** definiowany tutaj to operator dopełnienia logicznego. Dla podanej funkcji – predykatu $p: X \rightarrow \text{boolean}$ zwraca on funkcję $p': X \rightarrow \text{boolean}$, taką, że

$p'(x) = \sim p(x)$ dla każdego x należącego do X .

Operator *complement* – implementacja

Podobnie, jak miało to miejsce w przypadku składania funkcji, operator dopełnienia zostanie zdefiniowany w klasie – przestrzeni nazw *Functor*:

```
public class Functor {  
  
    static <X> Functor.Unary<X, Boolean>  
        complement(final Functor.Unary<X, Boolean> p) {  
  
        return new Functor.Unary<X, Boolean>() {  
            @Override  
            public Boolean call(X x) {  
                return !p.call(x);  
            }  
        };  
    }  
}
```

Operator *complement* – przykład użycia

Modelujemy prostą sytuację, gdy predykat p odpowiada na pytanie, czy podana wartość typu całkowitego jest parzysta lub nie:

$p: \text{int} \rightarrow \text{boolean}$ oraz $p = x \rightarrow \text{czy-parzysta?}(x)$

W takim razie

$\text{complement}(p): \text{int} \rightarrow \text{boolean}$ oraz
 $(\text{complement}(p))(x) = x \rightarrow \sim(\text{czy-parzysta?}(x))$

```
Functor.Unary<Integer, Boolean> p = new Functor.Unary<Integer, Boolean>() {  
    @Override  
    public Boolean call(Integer x) {  
        return x % 2 == 0;  
    }  
};
```

```
Functor.Unary<Integer, Boolean> p1 = complement(p);  
System.out.println(p1.call(256)); // ==> false
```

Zadanie

Zaimplementować przeciążenie procedury ***complement*** dla funktorów:

- binarnego
- trójargumentowego
- wieloargumentowego

Umiejętnie określić typy parametrów. Przetestować !!!