Grzegorz Jakacki

Codility Ltd.

Języki – komunikacja z człowiekiem i maszynami

Często nie zdajemy sobie sprawy, że otaczają nas nie tylko języki naturalne (polski, angielski), ale także wiele języków sztucznych, którymi bezwiednie posługujemy się na co dzień w komunikacji z ludźmi i maszynami. Przyciski pilota od telewizora, liczby rzymskie, oznaczenia na mapach, a nawet zielone światło na skrzyżowaniu to elementy języków sztucznych. Szczególnie ciekawa grupa to języki programowania, czyli języki służące do precyzyjnego instruowania maszyn.

Od lat 50. XX wieku trwają prace nad językami do komunikacji z maszyną. Pierwsze języki były szorstkie i niewygodne do stosowania. Komunikacja była trudna, trzeba było wkładać dużo pracy w to, żeby się dobrze wysłowić. Łatwo było się pomylić, zostać zrozumianym opacznie. Z biegiem czasu powstawały języki coraz doskonalsze. Rozwinięto metody opisu języków, pomagające zachować precyzję i poprawność wypowiedzi. Stworzono narzędzia pomagające tłumaczyć języki wygodniejsze dla człowieka na języki bardziej pasujące do maszyn. Zaprojektowano mechanizmy językowe, dzięki którym maszyna potrafi zrozumieć i wyłapać niektóre pomyłki rozmówcy. Zbadano style komunikacji z maszyną i stworzono języki, ułatwiające spójne formułowanie wypowiedzi, a utrudniające bałaganiarstwo.

W tym rozdziale pokazujemy przykłady różnych języków sztucznych, demonstrujemy kilka prostych metod projektowania takich języków oraz opowiadamy historię rozwoju komputerów z punktu widzenia komunikacji programisty z maszyną.

1. Prolog

Wiosną 1965 roku na biurku Martina Minsky'ego, kierownika projektu Sztucznej Inteligencji na Politechnice Massachusetts (Massachusetts Institute of Technology), odezwał się bakelitowy telefon. Dzwonił Stanley Kubrick, reżyser i producent filmowy, który od ponad roku pracował nad scenariuszem do nowego filmu science-fiction dla wytwórni Metro-Goldwyn-Mayer. Kubricka interesowały perspektywy rozwoju sztucznej inteligencji, a to akurat była domena Minskiego. Nie tylko badał sztuczną inteligencję, ale też filozofował na pokrewne tematy. W filmie Kubricka komputer HAL 9000, potężna maszyna obdarzona zdolnością odczuwania emocji i komunikacji za pomocą mowy, zainstalowany jako komputer pokładowy na statku kosmicznym, popada w wewnętrzny konflikt moralny spowodowany sprzecznymi rozkazami i, by go uniknąć, po kolei morduje członków załogi. Ostatni pozostały przy życiu astronauta z trudem dostaje się do wnętrza maszyny i, nie mając innych możliwości zatrzymania komputera, rozpoczyna sukcesywny demontaż modułów pamięci. Komputer próbuje negocjować z astronautą, a czując nadciągający koniec, zaczyna się bać. Astronauta demontuje kolejne moduły pracującej maszyny, wciąż prowadząc z nią rozmowę. HAL stopniowo popada w otępienie, a na koniec mentalnej agonii głosem przechodzącym w powolny bełkot, śpiewa piosenkę *Daisy Bell*, której nauczono go na wczesnym etapie programowania. Film *Odyseja Kosmiczna 2001* miał premierę 2 kwietnia 1968 roku, a mówiący komputer HAL 9000 zawładnął masową wyobraźnią.

Komunikacja z maszyną za pomocą mowy to marzenie, które pojawia się od zarania komputerów. Póki co komputery mają problem z reagowaniem na ludzką mowę. Przez ostatnie 50 lat to raczej ludzie uczyli się mówić tak, żeby komputer rozumiał, o co im chodzi.

2. Moje pierwsze sztuczne języki

2.1. Jak babcia pokazała mi język

Pierwszego sztucznego języka do komunikacji z maszyną nauczyłem się w wieku trzech lat, gdy babcia pokazała mi sygnalizator na przejściu dla pieszych. Maszyna komunikowała się ze mną bardzo prostym językiem: czerwony ludzik, zielony ludzik, który czasami migał. Ten język ma proste zasady:

(1) czerwony i zielony ludzik nigdy nie świeciły się jednocześnie,
(2) jeśli zielony ludzik migał, to znaczyło, że zaraz pokaże się czerwony.

Ten język służy do przekazywania informacji, czy można bezpiecznie przejść przez jezdnię. Ja też komunikowałem coś sygnalizatorowi. Akurat koło domu mojej babci na słupku sygnalizatora zainstalowano małe blaszane pudełko z gumowym przyciskiem. Mogłem nacisnąć gumowy przycisk, co w języku sygnalizacji oznaczało „chcę przejść, daj mi zielone światło".

Wiele lat później odkryłem, że sygnalizatory w innych krajach (np. w USA) porozumiewają się z przechodniami podobnie, chociaż język jest nieco inny. Był tam biały idący ludzik oraz pomarańczowy napis STOP i, by zasygnalizować, że faza bezpiecznego przejścia dobiega końca, sygnalizator mrugał napisem STOP, odmiennie od polskich sygnalizatorów, posługujących się wówczas migającym zielonym sygnałem.

Moja znajomość języka sygnalizacji świetlnej pogłębiła się, kiedy zdawałem egzamin na prawo jazdy. Poznałem wtedy język sygnalizatorów ulicznych przeznaczonych dla samochodów. Ten język jest dużo bogatszy od języka komunikacji z pieszymi. Ma więcej znaków, np. światło żółte, małą zieloną strzałkę pod sygnalizatorem, sygnały kierunkowe. Znaki można łączyć, np. światło czerwone z małą zieloną strzałką oznacza „nie wolno jechać prosto przez skrzyżowanie, ale można skręcić, ustępując pierwszeństwa". Pewne kombinacje znaków mają inne znaczenie od tych samych znaków występujących osobno, np. światło czerwone z żółtym oznacza co innego niż światło czerwone i co innego niż światło żółte. Pewne kombinacje sygnałów są niepoprawne, np. jednoczesne światło czerwone i zielone.

2.2. Język liczb arabskich

Drugim sztucznym językiem, którego się nauczyłem jeszcze w przedszkolu, był dziesiętny pozycyjny system zapisu liczb, który znają wszyscy, więc nie jest czymś nadzwyczajnym[1]. Ten język ma dziesięć **symboli-cyfr** (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), które można łączyć zapisując je poziomo, jeden obok drugiego w jeden napis zwany **liczbą**. Narzuca on niewiele ograniczeń. W zasadzie tylko takie, że cyfry należy pisać jedna obok drugiej poziomo, a nie np. po skosie albo pionowo. O ile w języku sygnalizatorów ulicznych pewne kombinacje symboli są zakazane, o tyle w przypadku zapisu pozycyjnego każdy ciąg cyfr jest liczbą. Może jedynie nie należy pisać zer z lewej strony liczby (np. 0005), ale nawet jeżeli ktoś to zrobi, to wciąż można zrozumieć, o co chodzi (0005 to po prostu 5).

2.3. Język liczb rzymskich

W moim domu rodzinnym wisiał zegar z kukułką. Majstrowałem przy nim czasami, zmuszając kukułkę do nadmiarowego kukania, skutkiem czego cykl

[1] O pochodzeniu systemów liczenia jest mowa w rozdziale *Historia rachowania – ludzie, idee, maszyny*.

kukułki rozjeżdżał się z tarczą i kukułka kukała np. 12 razy o godzinie szóstej. Zegar miał tarczę, a na niej wyrażenia innego sztucznego języka. Podstaw tego języka nauczyła mnie mama, a szczegóły poznałem w pierwszej klasie podstawówki. Ten język ma siedem symboli I, V, X, L, C, D i M, które można zapisywać w ciągach, oznaczających liczby. Oznaczają one:

I = 1 V = 5 X = 10 L = 50 C = 100 D = 500 M = 1000

Kolejne godziny na naszym zegarze z kukułką oznaczone były słowami I, II, III, IIII, V, VI, VII, VIII, IX, X, XI, XII. Zegar z kukułką zawisł w moim rodzinnym domu w roku 1982, czyli MCMLXXXII.

Język liczb rzymskich ma bardziej złożone zasady tworzenia poprawnych liczb niż dziesiętny system pozycyjny. Na przykład, w liczbie MCMLXXXII mamy M = 1000, CM = 900, LXXX = 80 i II = 2, co razem daje 1982. „Cyfry" w tym systemie (np. L = 50) można modyfikować, dopisując z prawej lub lewej strony symbole oznaczające mniejsze wartości. Dopisanie z prawej powiększa wartość, a dopisanie z lewej – zmniejsza[2]. Na przykład LX to „pięćdziesiąt zwiększone o dziesięć", czyli 60, LXXX to „pięćdziesiąt trzy razy zwiększone o 10", czyli 80, CM to „tysiąc zmniejszone o sto", czyli 900. Zawsze mniejsza liczba modyfikuje większą, a więc CM to „tysiąc zmniejszone o sto", a nie „sto zwiększone o tysiąc".

Reguły języka liczb rzymskich powodują powstawanie niejednoznaczności. Po pierwsze tę samą liczbę można wyrazić na wiele sposobów, np. 4 to IIII, ale również IV. Po drugie trudno powiedzieć, czy MXXL to M XXL, MX XL, czy MXX L (czyli odpowiednio 1030, 1050 czy 1070). Takie niejednoznaczności istnieją w wielu językach. Niejednoznaczność typu IIII/IV pojawia się nawet w dziesiętnym systemie pozycyjnym, np. 0005 i 5. Od takich niejednoznaczności roi się też w językach naturalnych, np. w języku polskim „auto" i „samochód" oznaczają to samo. Niejednoznaczność w interpretacji słowa MXXL wydaje się jednak poważniejszym problemem.

Niejednoznaczność w języku polskim ilustruje historia o informatyku wysłanym na zakupy:
– Kup serdelki, a jeśli będą jajka, to kup dwa.
– OK.
Wraca po chwili.
– Kupiłem dwa serdelki.
– A jajka?
– Były.

[2] Taki system liczbowy nazywa się **addytwnym**.

Język polski jest wieloznaczny i dopuszcza dwie różne interpretacje: „Kup serdelki. Jeśli będą jajka, to kup dwa serdelki" oraz „Kup serdelki. Jeśli będą jajka, to kup dwa jajka".

W języku liczb rzymskich istnieją dodatkowe reguły, które zapobiegają niejednoznacznościom w rodzaju MXXL, na przykład:

(1) liczba powinna zostać tak zapisana, aby osobno były zapisane tysiące, osobno setki, osobno dziesiątki i osobno jedności,
(2) każdy z symboli I, X, C i M może wystąpić co najwyżej trzy razy pod rząd (co przesądza na rzecz IV, a przeciwko IIII),
(3) każdy z symboli D, L i V może wystąpić co najwyżej raz pod rząd,
(4) I można odjąć wyłącznie od L lub X,
(5) X można odjąć wyłącznie od L lub C,
(6) C można odjąć wyłącznie od D lub M,
(7) nie wolno odejmować V, L ani D,
(8) można odjąć co najwyżej jeden symbol.

Interpretacja liczby MXX L jako 1020 + 50 = 1070 narusza regułę (1) – liczbę 1070 należy zapisać jako MLXX. Interpretacja liczby MX XL jako 1010 + 40 = 1050 narusza regułę (1) – liczbę 1050 należy zapisać jako ML. Interpretacja liczby M XXL jako 1000 + 30 = 1030 narusza regułę (8) – liczbę 1030 należy zapisać jako MXXX. Możliwe są jeszcze inne interpretacje zapisu MXXL, np. M X XL, czyli 1000 + 10 + 40, ale jest on też zabroniony przez powyższe reguły.

Język liczb rzymskich ma proste reguły dotyczące tworzenia nowych wyrażeń (dopisywanie z lewej i z prawej), ale ma złożone reguły dotyczące tego, które z wyrażeń są poprawne. W tym języku pojawia się również zjawisko kontekstowości, np. znak X może mieć różne znaczenie, w zależności od tego, gdzie występuje i co jest dookoła niego. Może oznaczać 10, ale też może oznaczać „dodaj 10 od budowanej liczby" albo „odejmij 10 od budowanej liczby". Choć liczby rzymskie znałem od pierwszej klasy, w praktyce po raz pierwszy zetknąłem się z kontekstowością nieco później.

2.4. Konwersacje z telewizorem

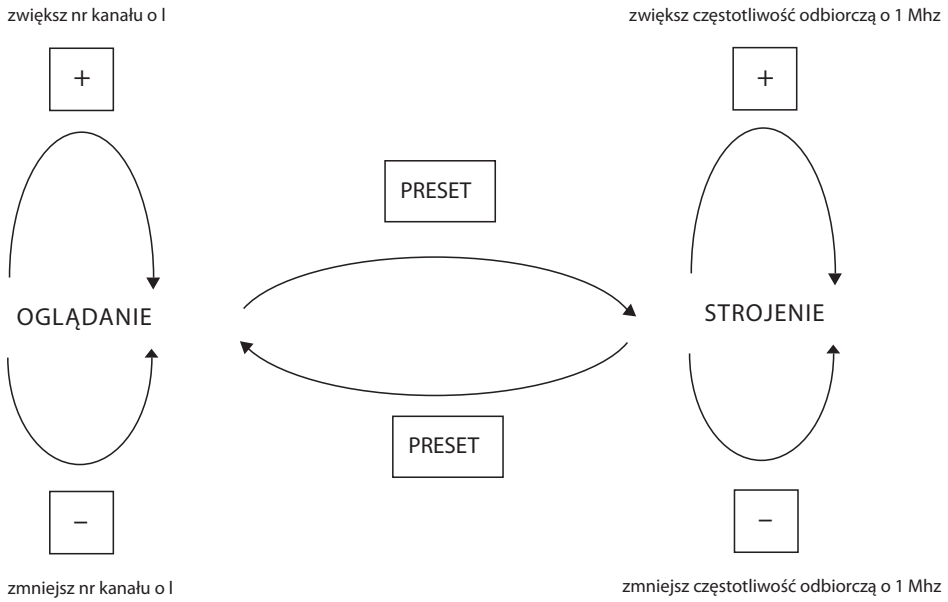
Kiedy miałem 12 lat w naszym domu pojawił się pierwszy japoński telewizor marki Otake. Z pilotem! Poczuliśmy powiew wielkiego świata, ale skokiem technologicznym okazało się dopiero strojenia kanałów. Poprzednik Otake, radziecki Elektron, miał sześć przycisków służących do wyboru kanałów. Naciśnięcie przycisku oznaczało wybór kanału. To i tak było więcej niż potrzeba, bo Telewizja Polska nadawała wówczas tylko dwa programy. Obok każdego przycisku było kółeczko do strojenia częstotliwości odbiorczej dla danego kanału.

Obrót w lewo oznaczał mniejszą częstotliwość, a obrót w prawo – większą. To było proste i intuicyjne.

Telewizor Otake komunikował się jednak za pomocą innego języka. Ukryty pod klapką panel zawierał kilka przycisków: dwa przyciski oznaczone + i –, przycisk PRESET oraz przycisk SHIFT. Wszystkie służyły do strojenia, ale ich znaczenie zmieniało się w zależności od tego, co naciśnięto wcześniej. To była ta kontekstowość. Telewizor miał wyświetlacz pokazujący numer kanału z zakresu 00-99. Naciśnięcie i przytrzymanie przycisku PRESET przez trzy sekundy powodowało wejście telewizora w tryb strojenia. Numer kanału zaczynał migać, natomiast przyciski + i – zmieniały swoje znaczenie. Zamiast przełączać kanał, powodowały zmniejszenie lub zwiększenie częstotliwości odbiorczej danego kanału. Naciśnięcie przycisku PRESET powodowało powrót do normalnego trybu. Dodatkowo przytrzymanie przycisku SHIFT powodowało modyfikację funkcji przycisków + i –. W normalnym trybie pracy zaczynały przełączać numer kanału o dziesięć, natomiast w trybie strojenia zmieniały częstotliwość odbiorczą o większy skok. Z kolei przycisk SHIFT nie miał żadnej samodzielnej funkcji, służył tylko do modyfikacji znaczenia innych przycisków. Taki modyfikator występuje nie tylko w języku telewizora Otake. Mała zielona strzałka w prawo na sygnalizatorze ulicznym jest takim modyfikatorem. W języku zapisu liczb całkowitych takim modyfikatorem jest np. znak minus stawiany przed liczbą. Sam nie ma znaczenia, natomiast modyfikuje znaczenie stojących za nim symboli. W języku polskim podobną funkcję mają np. przedrostki takie jak niby-, anty-, vice-.

Nastroiłem dwa kanały na program pierwszy i drugi TVP. Telewizor wciąż mnie intrygował, więc nastawiłem te same programy na kolejnych kanałach – program pierwszy na parzystych, program drugi na nieparzystych. To nie zaspokoiło mojej ciekawości, więc przeczytałem instrukcję. Zauważyłem, że instrukcja nie precyzuje, co się zdarzy, jeśli jednocześnie nacisnę + i –. Postanowiłem to sprawdzić empirycznie, ale o moich eksperymentach dowiedzieli się rodzice i dostałem szlaban nie tylko na zabawę telewizorem, ale i na telewizję. Na szczęście wkrótce nasz sąsiad nabył taki sam telewizor i wezwał mnie jako specjalistę od komunikacji z maszyną. Mogłem więc kontynuować eksperymenty w języku telewizora Otake. Jednoczesne naciśnięcie + i – powodowało pojawianie się na wyświetlaczu dziwnych znaków, wcale nieprzypominających cyfr. Doszedłem do wniosku, że tego typu kombinacje powinny być w języku telewizora Otake zabronione. Akurat tak się złożyło, że kolejni znajomi kupowali japońskie telewizory, miałem więc jeszcze kilka okazji komunikowania się z odbiornikami marek Otake, Hitachi, Fujitsu i JVC. Na języki tych telewizorów składały się naciśnięcia przycisków. Te naciśnięcia można było łączyć na dwa sposoby: albo naciskać przyciski jeden po drugim, albo jednocześnie (np. SHIFT równocześnie z +). Pewne kombinacje były dopuszczalne, pewne nie,

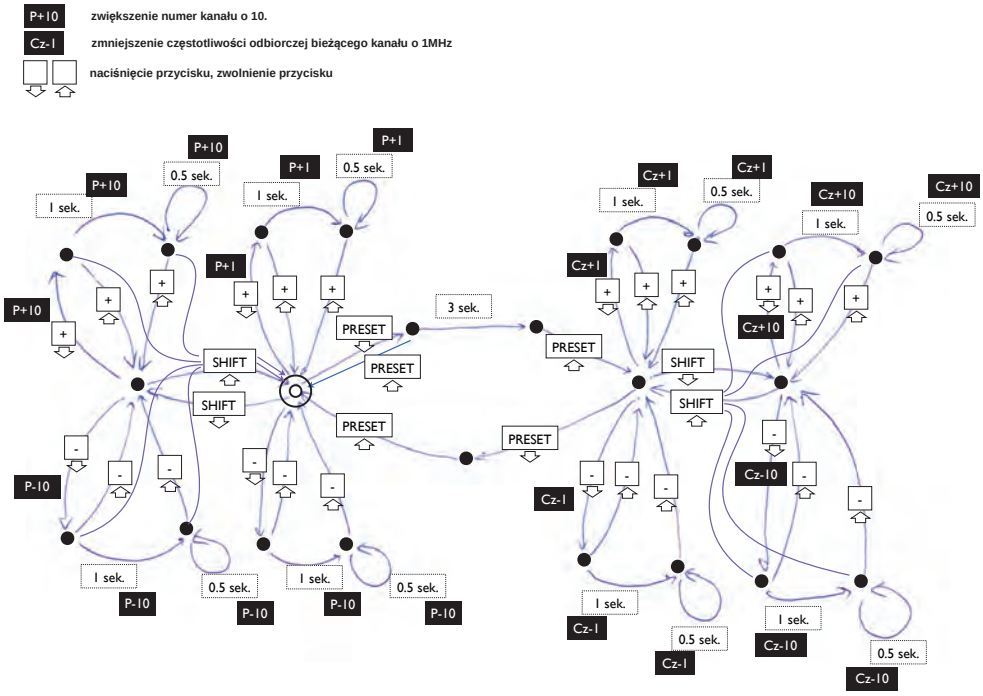
np. jednoczesne naciskanie + oraz – wyraźnie nie wychodziło na zdrowie sterownikowi telewizora Otake. Z nudów zacząłem rysować schematy opisujące języki telewizorów, patrz. rys. 1.



Rysunek 1. Opis przejść pomiędzy stanami telewizora Otake, wersja pierwsza

Nie wiedziałem wtedy jeszcze, że badam języki formalne, a te rysunki to schematy tzw. automatów skończonych opisujących języki. Wiedziałem natomiast, że mój rysunek jest niewystarczający. Nie potrafiłem znaleźć sposobu, by umieścić na nim SHIFT, albo żeby przekazać informację, że PRESET trzeba przytrzymać przez 3 sekundy, a + wystarczy nacisnąć bardzo krótko. Telewizory miały też taką dodatkową funkcję, że gdy przycisk + lub – przytrzymało się przez dłuższą chwilę, to jego działanie zaczynało się powtarzać, np. telewizor przełączał kolejne programy lub przeglądał coraz większe częstotliwości odbioru.

Wpadłem wtedy na pomysł, żeby spojrzeć na język telewizora Otake z innej perspektywy. Początkowo przyjąłem, że podstawowym jego elementem jest wciśnięcie przycisku. Do opisania pewnych cech języka to nie wystarczało. Naciśnięcie przycisku SHIFT było zazwyczaj rozciągnięte w czasie, od naciśnięcia przycisku do jego zwolnienia mogło upłynąć wiele sekund. Postanowiłem spojrzeć na ten język tak, jakby naciśnięcia i zwolnienia przycisków były odrębnymi zdarzeniami. Naciśnięcie oznaczyłem strzałką w dół, zwolnienie strzałką w górę, patrz rys. 2.



Rysunek 2. Opis przejść pomiędzy stanami telewizora Otake, wersja druga

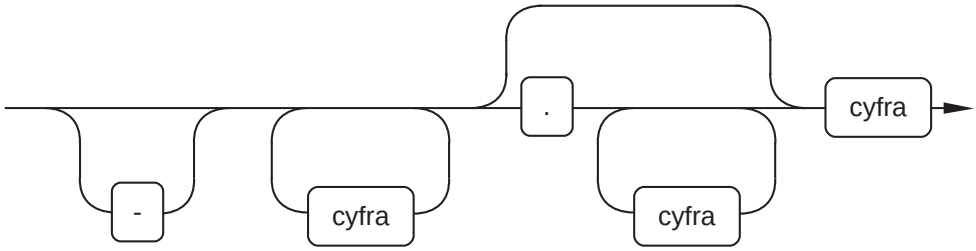
W trakcie rysowania stwierdziłem, że oprócz naciśnięcia i zwolnienia przycisku jest jeszcze jeden podstawowy element języka: upływ czasu. Jeżeli nacisnąłem PRESET i przytrzymałem go, to po upływie trzech sekund coś się działo (telewizor przełączał się w tryb strojenia), nawet jeśli nie zwolniłem ani nie nacisnąłem żadnego przycisku.

Z diagramami opisującymi języki spotkałem się ponownie parę lat później. W szkole średniej sięgnąłem po książkę *Algorytmy + Struktury Danych = Programy* Niklausa Wirtha [4]. Niewiele z niej rozumiałem, ale bardzo podobał mi się Dodatek B, zawierający całą masę diagramów opisujących różne fragmenty **języka Pascal**, m.in. diagram podobny do pokazanego na rysunku 3[3]. Jest to diagram opisujący język zapisu liczb. Dopuszczalnymi wyrażeniami tego języka są np. **5, –64, .01, –3.14**.

Dopiero po latach skonstatowałem, że rysunki w Dodatku B wspomnianej książki, a także diagramy opisujące sterownik telewizora Otake, to opisy języków.

[3] W istocie połączyłem na tym jednym diagramie dwa diagramy występujące we wspomnianej książce, ale nie ma to tutaj większego znaczenia.

Te opisy są sformułowane w podobny sposób. Składają się z prostych symboli: kropek i etykietowanych strzałek. Te elementy można łączyć według ustalonych reguł (np. strzałka musi zaczynać i kończyć się na kropce). Tak narysowany diagram niesie pewną treść – opis zachowania sterownika. Zaciekawiło mnie, że te kropki i etykietowane strzałki to też język. Za pomocą jednego sztucznego języka opisywałem inne.



Rysunek 3. Diagram opisujący popularną notację liczb wymiernych

3. Opisywanie języków

Człowiek zajmuje się badaniami języków naturalnych i sztucznych od tysiącleci. Można to robić na wiele różnych sposobów, w tym rozdziale skupiamy się na podejściu, w którym wyróżnia się:

- podstawowe symbole języka, czyli **leksykę**,
- zasady łączenia tych symboli w wyrażenia, czyli **gramatykę**,
- znaczenie wyrażeń, czyli **semantykę**.

Tabela 1 zawiera opisy leksyki, gramatyki i semantyki przykładowych języków.

Tabela 1.
Opisy przykładowych języków pojawiających się w tym rozdziale

Język	Leksyka	Gramatyka	Semantyka
Sygnalizator uliczny	czerwony ludzik, zielony ludzik	czerwony i zielony nie występują razem	zielony – droga wolna, zielony migający – zaraz będzie czerwony, czerwony – stój
Sterownik telewizora	kropki i etykietowane strzałki	strzałka musi zaczynać się i kończyć w kropce	etykiety na strzałkach oznaczają przyciski telewizora, kropki oznaczają stany telewizora; naciśnięcie przycisku odpowiada przejściu po strzałce

Tabela 1 (cd.)

Język	Leksyka	Gramatyka	Semantyka
Liczby arabskie	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	ciągi symboli (w linii)	wartość liczby uzyskujemy mnożąc wartość cyfry przez 10 do potęgi *k*, gdzie *k* oznacza numer pozycji licząc od prawej i zaczynając od zera
Liczby rzymskie	I, V, X, L, C, D, M	ciągi symboli (w linii), każdy z symboli co najwyżej 3 razy pod rząd, od V i X można odejmować wyłącznie I...	Przyjmij wartość zero i powtarzaj następującą operację: odetnij najdłuższy możliwy z poniższych przedrostków i do wartości dodaj odpowiadającą mu liczbę: I → 1 IL → 49 IX → 9 X → 10 XL → 40 XC → 90 C → 100 CD → 400 CM → 900 M → 1000 *Uwaga: ten sposób interpretacji wartości zakłada, że liczba jest poprawnie zapisana.*
Proste wyrażenia algebraiczne	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, *, (,)	gramatyka jest opisana na rysunku 11	Zinterpretuj wyrażenie według gramatyki opisanej na rysunku 11 i wykonaj działania arytmetyczne w kolejności wyznaczonej przez rozkład gramatyczny.
Język maszynowy	Liczby naturalne	ciągi liczb naturalnych, zapisanych według specyfikacji procesora	powodują modyfikacje pamięci przez kopiowanie wartości, operacje arytmetyczne i skoki *Semantykę szczegółowo opisuje dokumentacja techniczna procesora.*
Język mnemoniczny	Rozkazy: MOV, SUB, ADD, JMP, CMP...	ciągi rozkazów z jednym lub dwoma argumentami	rozkazy należy przetłumaczyć na kod maszynowy według opisu języka mnemonicznego i interpretować jako kod procesora

Tabela 1 (cd.)

Język	Leksyka	Gramatyka	Semantyka
Język C	liczby, słowa, znaki przestankowe, operatory +, – ..., nawiasy	gramatyka opisana w postaci klauzul podobnych do BNF w dokumencie standaryzacyjnym ISO/IEC JTC1/ SC22/WG14	semantyka opisana za pomocą prozy technicznej w dokumencie standaryzacyjnym ISO/IEC JTC1/SC22/WG14

Pierwsze poważne i udane podejście do precyzyjnego opisu gramatyki języka wykonał hinduski lingwista Pānini (पाणिनि) w VI wieku p.n.e. Opracował on precyzyjny system opisu gramatyki i sformułował w nim 3959 reguł opisujących gramatykę języka Sanskryt. Na następne równie udane podejście trzeba było czekać dosyć długo. Powszechnie uważa się, że dopiero prace Noama Chomskiego, Johna Backusa oraz Petera Naura na przełomie lat 50. i 60. XX wieku posunęły tę dziedzinę do przodu. W roku 1959 John Backus, projektując dla firmy IBM język IAL (znany obecnie pod nazwą Algol 58), zaproponował ścisły sposób opisu gramatyki pewnego sztucznego języka. Ale o tym za chwilę.

Komputer HAL 9000 z *Odysei Kosmicznej 2001* porozumiewał się z astronautami po angielsku. Była to pociągająca wizja, daleka jednak od rzeczywistości lat 60. Język komunikacji człowieka z maszyną był w tamtych czasach wyjątkowo surowy i zdecydowanie był to język wygodny dla maszyny, ale niekoniecznie dla człowieka.

Może trudno w to uwierzyć, ale od końca lat 40. komputery w zasadzie buduje się podobnie. Komputer składa się z pamięci oraz procesora. Pamięć to przechowalnia liczb. Komórki pamięci, ponumerowane kolejnymi liczbami całkowitymi, mogą przechowywać liczby całkowite z niewielkiego zakresu (współcześnie zazwyczaj 0-255). Procesor pobiera wartości z kolejnych komórek pamięci i interpretuje je jako rozkazy powodujące zmianę wartości pewnych komórek pamięci lub wysyłanie sygnałów elektrycznych na wyprowadzenia procesora. To wszystko. Część komórek pamięci zawiera wartości przeznaczone do sterowania zachowaniem procesora (tzw. **program**), część zawiera wartości, na których procesor wykonuje operacje (tzw. dane). Na przykład, współczesny procesor firmy Intel zinterpretuje ciąg liczb 49, 237, 103, 138, 69, 0, 103, 2, 69, 1, 103, 136, 69, 2 jako program powodujący dodanie wartości z komórki pamięci o adresie 0 do komórki pamięci o adresie 1 i zapisanie wyniku w komórce pamięci o adresie 2.

Liczby w pamięci składające się na program dla procesora można traktować jak wyrażenia języka. Jest to bardzo surowy język komunikacji z maszyną. Stąd też nazwa – **język maszynowy**. Symbolami języka maszynowego są liczby natu-

ralne zapisane w pamięci. Znaczenie tych liczb opisuje dokumentacja techniczna procesora.

W czasach, gdy powstawał film o mówiącym komputerze HAL 9000, komunikacja z maszynami odbywała się wciąż na tym najniższym poziomie. Programiści wprowadzali do komputerów kod maszynowy za pomocą specjalnych kart dziurkowanych, a w starszych modelach po prostu za pomocą przełączników i kabli elektrycznych. Był to proces żmudny i podatny na błędy. Ludzki mózg, przyzwyczajony do języków, w których występują czasowniki i rzeczowniki, niekoniecznie płynnie porusza się w języku maszynowym, w którym występują tylko liczby. O ile wygodniej człowiekowi posługiwać się frazą np. „wyzeruj rejestr EBP" zamiast „49, 237". Wszystkie kody języka maszynowego można ponazywać takimi mnemonicznymi nazwami, tworząc język podobny do języka maszynowego, jednak o wiele bardziej czytelny. Na przykład, programy dodawania dwóch liczb w obu językach i w języku wyrażeń algebraicznych mają postać:

Kod maszynowy:	Kod mnemoniczny (język procesora i686)	Język wyrażeń algebraicznych
49, 237	XOR %EBP, %EBP	
103, 138, 69, 0	MOV 0(%EBP), %AL	x + y
103, 2, 69, 1	ADD 1(%EBP), %AL	
103, 136, 69, 2	MOV %AL, 2(%EBP)	

Rysunek 4. Programy, które służą do obliczania wartości x + y

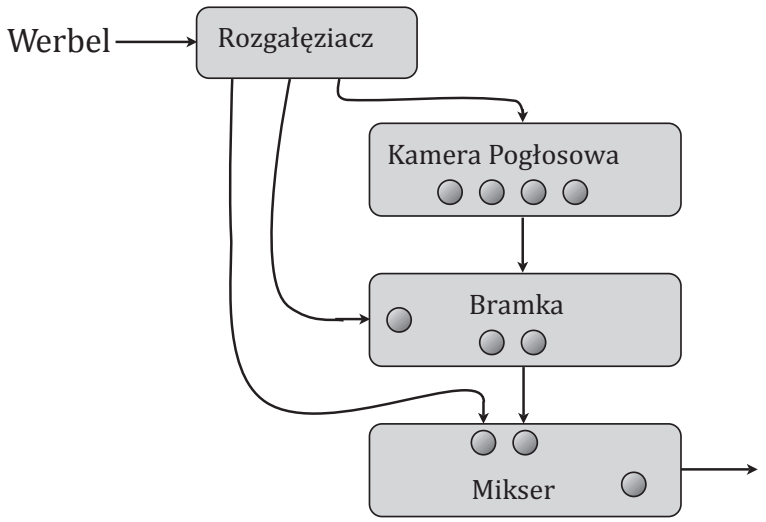
Dźwięk perkusji w utworze *In The Air Tonight* Philla Collinsa jest tak niesamowity, że kopiowali go wszyscy muzycy rockowi przez całe lata 80. XX wieku. Dźwięk werbla jest zaskakująco bogaty, jednocześnie krótki i spójny. Ten niezwykły efekt dźwiękowy zaprojektował Hugh Padgham, pracujący z Collinsem nad płytą Face Value. Konfiguracja przetworników sygnału wykorzystanych do stworzenia tego efektu jest pokazana na rysunku 5. Sygnał z mikrofonu nagrywającego perkusję rozdziela się na trzy linie. Jedna z nich jest podłączona do wejścia kamery pogłosowej. Kamera dodaje do sygnału pogłos, sprawiający wrażenie, że perkusista gra w wielkiej sali, w której dźwięk odbija się od ścian. Pogłos bardzo wzbogaca dźwięk, sprawia jednak, że krótkie, zdecydowane uderzenia (np. uderzenia werbla) stają się rozlazłe i rozwleczone w czasie – jeszcze długo po wybrzmieniu oryginalnego dźwięku słychać cichnące pogłosy i echa. Bramka przepuszcza sygnał tylko wtedy, kiedy poziom głośności sygnału na wejściu sterującym jest odpowiednio wysoki. Podanie oryginalnego sygnału na wejście steru-

jące powoduje, że dźwięk wzbogacony o pogłosy jest ucinany zaraz po wybrzmieniu oryginalnego, krótkiego uderzenia werbla. Najlepsze efekty uzyskuje się miksując tak opracowany sygnał z odrobiną pierwotnego sygnału z mikrofonu. Całość należy dostroić „na ucho", np. dobierając odpowiednie opóźnienia w kamerze pogłosowej, czas otwarcia bramki po wybrzmieniu sygnału sterującego, czy też poziom sygnału sterującego, poniżej którego bramka się zamyka.

W podobny sposób – przez strojenie i łączenie modułów kablami – programowało się pierwszy komputer elektroniczny – ENIAC. Program zapisywano najpierw na papierze, po czym zespół programistek mozolnie przenosił program na maszynę, wpinając kable i przełączając przełączniki. Dane do przetwarzania wprowadzano za pomocą czytnika kart perforowanych, służącego do wczytywania ciągów liczb wydziurkowanych na tekturowych kartach. Wyniki były dziurkowane przez maszynę na takich samych, czystych kartach.

Programowanie maszyny ENIAC było uciążliwe i często trwało tygodniami. Był to jeden z problemów, który próbowano rozwiązać w powstającym projekcie nowego komputera EDVAC. Udział w tych pracach brał John von Neumann, matematyk z Uniwersytetu Princeton, zaangażowany do Projektu Manhattan, dotyczącego broni masowego rażenia. Von Neumann korzystał z maszyny ENIAC do wykonywania obliczeń dotyczących bomby wodorowej. W notatkach z czerwca 1945 roku proponował konstrukcję, polegającą na wprowadzaniu do maszyny programu w tej samej postaci, co dane, tj. za pomocą ciągu liczb zapisanego na kartach perforowanych. Liczby miały oznaczać kolejne rozkazy dla maszyny, a maszyna miała te rozkazy po kolei wykonywać. Było to podejście zupełnie odmienne, ograniczało m.in. możliwość równoległego wykonywania operacji (w każdym momencie maszyna miała wykonywać tylko jedną operację z listy, podczas gdy w komputerze ENIAC możliwe było np. podłączenie równoległe dwóch modułów sumujących liczby jednocześnie), jednakże rozwiązywało o problem kłopotliwej i czasochłonnej rekonfiguracji kabli i przełączników przy każdorazowej zmianie programu. Pomysł został zrealizowany w roku 1948 po modyfikacji komputera ENIAC, jeszcze przed ukończeniem komputera EDVAC. Początkowy ciąg liczb wczytanych z kart perforowanych był interpretowany jako lista rozkazów, czyli program, a reszta stanowiła dane dla programu.

Przy takim podejściu przetwarzanie danych trwało sześciokrotnie dłużej, natomiast maszynę dawało się przeprogramować w ciągu jednego dnia, zamiast kilku tygodni – wystarczyło wczytać nowy zestaw kart perforowanych. Dodatkowo okazało się, że w przypadku większości obliczeń o wiele więcej czasu schodzi na wczytanie danych z kart perforowanych, niż na wykonanie samych obliczeń, więc spowolnienie nie miało znaczenia. Olbrzymia większość współczesnych komputerów, w tym komputery osobiste IBM PC, jest zbudowana według architektury von Neumanna – programy, tak jak dane, są przechowywane i przetwarzane w postaci ciągów liczb (lub symboli). Nie powinno więc nas dziwić, że programy są przechowywane na dysku, w tym samym miejscu, gdzie trzymamy zdjęcia, piosenki lub film.



Rysunek 5. Połączenia realizujące efekt dźwiękowy Gated Reverb

4. Nie możesz zrozumieć, czego nie możesz nazwać

Język mnemoniczny jest łatwiejszy dla człowieka, ponieważ przypomina trochę język naturalny. Każdy wiersz programu (patrz rys. 4) to fraza rozkazująca (polecająca). MOV to czasownik, po którym następują dwa dopełnienia. Oznacza tyle, co „skopiuj z X do Y". %EBP to rzeczownik, oznacza pomocniczą komórkę pamięci umieszczoną bezpośrednio w procesorze. Podobnie %AL. 1(X) to wyrażenie oznaczające „jedna komórka dalej niż". 1 odpowiada tu przydawce w języku polskim. Wyrażenie 1(%EBP) oznacza „jedna komórka pamięci za komórką o numerze przechowywanym w %EBP".

Język mnemoniczny łatwo przetłumaczyć na kod maszynowy, w zasadzie wystarczy słownik kodów mnemonicznych (np. XOR → 49) i parę prostych zasad dotyczących interpretacji wyrażeń takich jak X(Y), np. 1(%EBP). Tę pracę programiści wykonywali kiedyś ręcznie. Program napisany na papierze w języku mnemonicznym był ręcznie tłumaczony na kod maszynowy i wprowadzany do komputera. Zadanie to przerzucono na komputery w latach 50. XX wieku. Był to duży postęp, ale otchłań pomiędzy językiem mnemonicznym a płynną angielszczyzną komputera HAL 9000 wciąż była olbrzymia. W szczególności programiści musieli bardzo pilnować, co program przechowuje w których komórkach pamięci. Przypadkowe zapisanie danej do komórki przechowującej program mogło spowodować wytworzenie błędnego kodu, skutkującym nieprzewidzianym wykonaniem reszty programu.

Programiści piszący programy w języku maszynowym musieli nieźle się napocić, żeby zaprogramować nawet proste dodawanie. Nic dziwnego, że tęsknili za dobrze nam znanym językiem wyrażeń algebraicznych. Uczymy się go w szkole, składają się na niego liczby, litery, znaki działań (+, –, *, kreska ułamkowa), nawiasy. Zapis wyrażeń w takim języku ma wadę, gdyż na przykład dzielenia nie można zapisać w jednym wierszu. Wybrnięto z tego wprowadzając specjalny znak dzielenia /, a także umożliwiając zapisywanie potęgowania w jednym wierszu za pomocą znaku ^, zatem wyrażenie 6^10 oznacza tyle, co 6^{10}. W takiej konwencji na język wyrażeń składają się liczby, litery, znaki działań (+, –, *, /, ^), nawiasy, a zasady łączenia symboli są następujące:

(1) dwa wyrażenia można zestawić umieszczając obok siebie i łącząc je znakiem działania,
(2) wyrażenie można otoczyć parą nawiasów i otrzymać nowe wyrażenie.

Wystarczy spojrzeć na rysunek 4, żeby uświadomić sobie, o ile wygodniejszy jest język wyrażeń algebraicznych od języka mnemonicznego. Pierwszą osobą, którą zrobiła z tym problemem coś konkretnego był niemiecki inżynier Konrad Zuse. Zaprojektował on i zbudował podczas II wojny jeden z pierwszych komputerów na świecie. Pod koniec wojny salwował się ucieczką z bombardowanego Berlina i oderwany od pracy przy budowie komputera zaczął zastanawiać się nad tym, co trapiło go od pewnego czasu: kod maszynowy nie jest najwygodniejszym językiem programowania. Przemyślenia zaowocowały projektem języka Plankalkül, który Zuse opublikował parę lat po wojnie. Plankalkül wyprzedził swoją epokę i mimo wielu przełomowych rozwiązań, w szczególności możliwości zapisywania programów za pomocą składni przypominającej wyrażenia algebraiczne, nie doczekał się popularyzacji aż do lat 70. XX wieku.

Konrad Zuse, niemiecki inżynier budownictwa, przed wojną trafił do firmy Henschel, producenta lokomotyw i samolotów. Monotonia wykonywanych codziennie ręcznych obliczeń matematycznych skłoniła go do skonstruowania programowalnego mechanicznego kalkulatora V1, który służył do wykonywania obliczeń na liczbach wymiernych; instrukcje były wczytywane z dziurkowanej taśmy filmowej. Gdy we wrześniu 1939 roku samoloty Henschel Hs 123 bombardują Warszawę, Zuse ma 29 lat, a na koncie dwa patenty dotyczące konstrukcji maszyn liczących. Powołany do wojska przekonuje przełożonych, że konstruowane przez niego maszyny przysłużą się machinie wojennej. Tak powstaje kalkulator programowalny V2, ulepszona wersja V1, w której Zuse zastąpił elementy mechaniczne elektrycznymi przekaźnikami telefonicznymi. Gdy w czerwcu 1941 roku Niemcy atakują Związek Radziecki, a Hitler instaluje sztab w Wilczym Szańcu pod Kętrzynem, Zuse kończy prace nad maszyną V3, pierwszym

działającym komputerem elektromechanicznym. Komputer V3 pracuje dla Niemieckiego Instytutu Lotnictwa, gdzie jest używany do obliczeń związanych z aerodynamiką bomb szybujących. Zuse stara się o finansowanie maszyny V4, ulepszonej wersji V3, ale niemiecka administracja mu odmawia. Wiara w rychłe zwycięstwo Niemiec jest tak wielka, że urzędnicy nie widzą potrzeby budowania kolejnej maszyny. Pod koniec wojny Zuse ucieka z Berlina, a bomby aliantów niszczą jego komputery. Na przymusowych wakacjach w południowych Niemczech wymyśla Plankalkül, system zapisu rozkazów dla maszyn liczących, uważany za pierwszy zaawansowany język programowania. Na skutek izolacji spowodowanej przez wojnę do swoich odkryć dochodzi praktycznie sam, nie będąc świadomym prac Amerykanów i Brytyjczyków. Zuse po wojnie powrócił do konstruowania maszyn liczących. By uniknąć skojarzeń z niemieckimi pociskami rakietowymi V1 i V2, przemianował swoje konstrukcje na Z1, Z2, Z3. Komputer Z4 został zamówiony przez politechnikę w Zurychu i dostarczony przez Zusego w czerwcu 1950 roku. Język Plankalkül wyprzedził swoją epokę. Jego opis opublikowany pod koniec lat 40. XX wieku przeszedł bez echa. Plankalkül został odkryty ponownie w latach 70. i okazał się niezmiernie ciekawym projektem, zawierającym wiele użytecznych mechanizmów obecnych we współczesnych językach programowania.

Powstawały jednak inne projekty. Użytkownicy wczesnych komputerów szybko dostrzegli to, co Zuse przewidział już w latach 40. XX wieku: programowanie komputerów za pomocą ciągów liczb reprezentujących rozkazy jest bardzo uciążliwe. Konstrukcja komputera zaproponowana przez von Neumanna polegała na tym, że komputer pobierał z pamięci kolejne liczby i interpretował je jako rozkazy. Rozkazy mogły zmieniać zawartość pamięci, wprowadzać dane wejściowe, wyprowadzać dane wyjściowe lub wpływać na to, jaki rozkaz będzie wykonany w następnej kolejności. Nie od razu było to jasne, ale żeby dało się zaprogramować pewne obliczenia, potrzebny był **rozkaz rozgałęzienia**. Taki rozkaz polega na tym, że maszyna wykonuje różne akcje w zależności od prawdziwości pewnego warunku. Przykładem problemu, który jest trudno zrealizować bez użycia rozkazu rozgałęzienia, jest znajdowanie elementu maksymalnego w ciągu liczb dodatnich: wczytaj ciąg liczb i wyprowadź tę z nich, która ma największą wartość. Jeśli ograniczymy się do operacji arytmetycznych (+, –, *, /), to tego zadania w ogóle nie da się rozwiązać. Można to zrobić, jeżeli dysponujemy dodatkową operacją obliczania modułu liczby, czyli wartości bezwzględnej. Łatwo sprawdzić prawdziwość następującej równości:

$$\max(A, B) = (A + B + |A - B|) / 2$$

Kod maszynowy, realizujący obliczenie według powyższego wzoru ma następującą postać:

1	IN	1	do komórki nr 1 wprowadź liczbę A z wejścia (np. z karty perforowanej)
2	IN	2	do komórki nr 2 wprowadź liczbę B z wejścia (np. z karty perforowanej)
3	MOV	1,3	zapisz wartość z komórki nr 1 (A) do komórki nr 3
4	SUB	2,3	odejmij wartość z komórki nr 2 (B) od wartości w komórce nr 3 (A) i zapisz w komórce nr 3 (A – B)
5	ABS	3,3	w komórce nr 3 weź wartość bezwzględną jej zawartości
6	ADD	2,1	do wartości w komórce nr 1 (A) dodaj wartość z komórki nr 2 (B)
7	ADD	3,1	do wartości w komórce nr 1 (A+B) dodaj wartość z komórki nr 3 (|A – B|)
8	DIV	#2, 1	podziel wartość w komórce nr 1 (A+B+|A – B|) przez 2
9	OUT	1	wyprowadź wartość z komórki nr 1 ((A+B+|A – B|) / 2) na urządzenie wyjściowe

A jak obliczyć maksimum z trzech liczb bez użycia rozkazu rozgałęzienia? Zamiast od początku pisać nowy program, skorzystajmy z obserwacji, że:

$\max(A, B, C) = \max(\max(A, B), C)$,

a wtedy obliczenia mogą mieć następujący przebieg:

$\max(A, B) = (A + B + |A - B|) / 2$
$\max(X, C) = (X + C + |X - C|) / 2$
podstawiamy max(A, B) za X:
$\max(\max(A, B), C) = ((A + B + |A - B|) / 2 + C + |(A + B + |A - B|) / 2 - C|) / 2$

Program, który służy do obliczania wartości tego wyrażenia ma podobną postać jak wyżej, przy czym początkowy fragment kodu (obliczanie max (A, B)) jest identyczny, a zmiany zaczynają się od rozkazu 9 (w prawej kolumnie komentujemy tylko, jakie wartości są obliczane w poszczególnych komórkach przez kolejne rozkazy):

1	IN	1	1: A
2	IN	2	2: B
3	MOV	1,3	3: A
4	SUB	2,3	3: A – B
5	ABS	3,3	3: |A – B|
6	ADD	2,1	1: A + B
7	ADD	3,1	1: A + B + |A – B|
8	DIV	#2, 1	1: (A + B + |A – B|) / 2
9	IN	2	2: C
10	MOV	1,3	3: (A + B + |A – B|) / 2
11	SUB	2,3	3: (A + B + |A – B|) / 2 – C
12	ABS	3,3	3: |(A + B + |A – B|) / 2 – C|

13	ADD	2,1	1: (A + B + |A – B|) / 2 + C
14	ADD	3,1	1: (A + B + |A – B|) / 2 + C + |(A + B + |A – B|) / 2 – C|
15	DIV	#2,1	1: ((A + B + |A – B|) / 2 + C + |(A + B + |A – B|) / 2 – C|) / 2
16	OUT	1	wyprowadź wartość komórki nr 1

Rozkazy 2-8 i 9-15 są identyczne, ponieważ realizują to samo działanie – obliczanie maksimum dwóch liczb: rozkazy 2-8 obliczają maksimum z liczb A i B, natomiast rozkazy 9-15 obliczają maksimum z liczb max(A, B) i C. Ten schemat można w prosty sposób rozszerzyć na cztery liczby A, B, C, D. Wystarczy w tym celu jeszcze raz skopiować rozkazy 2-8 i wstawić przed rozkaz 16. Uzasadnienie i wykonanie tego rozszerzenia programu pozostawiamy Czytelnikowi.

Jak widać, bez użycia rozkazu rozgałęzienia można podać program służący do obliczania maksimum z ciągu liczb o ustalonej długości, nie można jednak podać jednego programu, który będzie działał dla ciągu o dowolnej długości, np. dla ciągu, którego koniec reprezentuje liczba – 1 (tzw. wartownik końca ciągu). Z rozkazem rozgałęzienia staje się to natomiast możliwe:

1	IN	1	1: pierwsza liczba z wejścia
2	IN	2	2: K z wejścia
3	CMP	# –1,2	pomiń następny rozkaz, jeśli wartość w komórce nr 2 jest równa –1
4	JMP	+2	przejdź do wykonania rozkazu 6 (o dwa rozkazy dalej)
5	JMP	+8	przejdź do wykonania rozkazu 13 (o osiem rozkazów dalej))
6	MOV	1,3	3: zapisz dotychczasowe maksimum z komórki nr 1 do komórki nr 3
7	SUB	2,3	3: odejmij liczbę K od wartości w komórce nr 3
8	ABS	3,3	3: weź wartość bezwzględną z wartości w tej komórce
9	ADD	2,1	1: dodaj wartość z komórki nr 2 do wartości w komórce 1
10	ADD	3,1	1: dodaj wartość z komórki nr 3 do wartości w komórce nr 1
11	DIV	#2, 1	1: podziel wartość w komórce nr 1 przez 2
12	JMP	–10	przejdź do wykonania rozkazu 2 (o 10 rozkazów wcześniej)
13	OUT	1	wyprowadź wartość komórki nr 1

Rozkaz rozgałęzienia (np. CMP w powyższym programie) daje komputerom ich prawdziwą moc. Niektóre z pierwszych maszyn liczących nie miały takich możliwości, np. kalkulator programowalny Z1 Konrada Zuse nie potrafił wykonywać rozgałęzień.

Wczesne kalkulatory programowalne nie zawierały rozkazu rozgałęzienia. Każdy program dla takiego kalkulatora można zapisać w postaci wzoru matematycznego. Na przykład, program, który sortuje dwie liczby M[1] i M[2], generuje posortowany ciąg O[1], O[2] według wzorów:

$$O[1] = M[1] / 2 + M[2] / 2 - |M[1] / 2 - M[2] / 2|$$
$$O[2] = M[1] / 2 + M[2] / 2 + |M[1] / 2 - M[2] / 2|$$

Proponujemy w podobny sposób opisać program, służący do sortowania trzech liczb M[1], M[2], M[3], a jeśli to jest zbyt proste, to proponujemy zapisanie sortowania czterech liczb M[1], M[2], M[3], M[4].

5. Język pomaga unikać błędów

Programy komputerowe rosły, a wraz z nimi rosły problemy. Wyobraźmy sobie, że pomiędzy rozkazy 11 i 12 wstawimy nowy rozkaz, powodujący wyprowadzenie dodatkowej informacji na urządzenie wyjściowe:

1	IN	1	1: pierwsza liczba z wejścia
2	IN	2	2: K z wejścia
3	CMP	# –1,2	pomiń następny rozkaz, jeśli wartość w komórce nr 2 jest równa –1
4	JMP	+2	przejdź do wykonania rozkazu 6 (o dwa rozkazy dalej)
5	JMP	+8	przejdź do wykonania rozkazu 12 (o siedem rozkazów dalej))
6	MOV	1,3	3: zapisz dotychczasowe maksimum z komórki nr 1
7	SUB	2,3	3: odejmij liczbę K od wartości w komórce nr 3
8	ABS	3,3	3: weź wartość bezwzględną z wartości w tej komórce
9	ADD	2,1	1: dodaj wartość z komórki nr 2 do wartości w komórce 1
10	ADD	3,1	1: dodaj wartość z komórki nr 3 do wartości w komórce nr 1
11	DIV	#2, 1	1: podziel wartość w komórce nr 1 przez 2
12	OUT	1	wyprowadź dotychczasowe minimum na urządzenie wyjściowe *<-nowy*
13	**JMP**	**–10**	przejdź do wykonania rozkazu 2 (o 10 rozkazów wcześniej)
14	OUT	1	wyprowadź wartość komórki nr 1

Nowy rozkaz ma teraz numer 12, a rozkazy, które były 12. i 13. stały się 13. i 14. Taki program ma jednak poważny błąd: na skutek przesunięcia się dawnych rozkazów 12 i 13, obecny rozkaz 13 (czyli JMP –10) spowoduje kontynuowanie wykonania programu od rozkazu 3 bez wczytania nowej danej, co powinno nastąpić w rozkazie 2. Rozkazy 3-13 będą wykonywane w nieskończonej pętli i program nigdy nie zakończy działania. Błąd polega na tym, że przy wstawianiu rozkazu trzeba uważać na odległości podane w rozkazach **skoków** JMP

(ang. *jumps*) i odpowiednio je poprawiać. To jest bardzo uciążliwe, programiści wymyślali więc mechanizmy umożliwiające unikanie tego kłopotu. Można na przykład nazwać niektóre rozkazy w programie:

```
                IN      1
WCZYTANIE:      IN      2
                CMP     # –1,2
                JMP     +2
                JMP     ZAKOŃCZENIE
                MOV     1,3
                SUB     2,3
                ABS     3,3
                ADD     2,1
                ADD     3,1
                DIV     #2, 1
                JMP     WCZYTANIE
ZAKOŃCZENIE:    OUT     1
```

Człowiek ma potrzebę nazywania: krajów, zjawisk, przedmiotów, innych ludzi. Coś, co nazwane, łatwiej zrozumieć i łatwiej się tym posługiwać. Nazwanie wybranych rozkazów programu umożliwia szybsze zrozumienie struktury programu, a także zapobiega błędom. Jeśli powtórzymy przykład z wstawieniem dodatkowego rozkazu OUT po rozkazie DIV, to dzięki temu, że posługujemy się nazwami, a nie numerami rozkazów, rozkaz JMP nie będą źródłem błędów.

Dalsze uproszczenie struktury programu uzyskuje się przez nazwanie komórek pamięci. Ostatecznie otrzymujemy następujący program:

```
A               DATA                    obliczone dotychczas maksimum
K               DATA                    nowa wczytana liczba
POM             DATA                    wartość pomocnicza do obliczania
                                        |A – K| / 2
                IN      A               A ← wejście
WCZYTANIE:      IN      K               K ← wejście
                CMP     #–1,K           jeżeli K = –1, pomiń następny
                                        rozkaz
                JMP     +2              pomiń następny rozkaz
                JMP     ZAKOŃCZENIE     kontynuuj od rozkazu
                                        ZAKOŃCZENIE
                MOV     A,POM           POM ← A
                SUB     K,POM           POM ← POM – K, czyli w POM
                                        mamy A – K
                ABS     POM,POM         POM ← |POM|, czyli w POM
                                        mamy |A – K|
                ADD     K,A             A ← A + K
                ADD     POM,A           A ← A+POM
                DIV     #2, A           A ← A / 2
                JMP     WCZYTANIE
ZAKOŃCZENIE:    OUT     A
```

Tak ewoluowały wczesne języki programowania. Posługiwanie się nazwami rozkazów, komórek pamięci, bloków rozkazów, często używanych fragmenty programów itd. umożliwiło swobodniejszą komunikacje programistów z komputerami, ale wciąż było daleko do wydawania głosem rozkazów dla komputera HAL 9000. W latach 50. XX wieku pojawił się język **Fortran**, do dzisiaj używany w obliczeniach numerycznych. Program do obliczania maksimum z ciągu liczb może mieć w Fortranie następującą postać (1957):

```
1    READ INPUT TAPE A          wprowadź do A pierwszą liczbę z wejścia
2    READ INPUT TAPE K          wprowadź do K kolejną liczbę z wejścia
3    IF (K) 6,4,4               jeśli K < 0, przejdź do instrukcji 6
                                jeśli K = 0, przejdź do instrukcji 4
                                jeśli K > 0, przejdź do instrukcji 4
4    A = (A+K+ABS(A–K))/2       przypisz do A wartość (A + K + |A – K|) / 2
5    GOTO 2                     przejdź do instrukcji 2
6    WRITE OUTPUT TAPE A        wyprowadź wartość z A
7    STOP                       zakończ wykonanie
```

To był postęp, szczególnie jeśli chodzi o wyrażenia algebraiczne. Instrukcja[4] 4 to olbrzymie uproszczenie w porównaniu z ciągiem rozkazów MOV/SUB/ABS/ADD/ADD/DIV z programu w języku maszynowym. Komunikacja z maszyną przeszła na wyższy poziom. Programiści przestali się martwić o całą masę szczegółów (jak odległość w rozkazach JMP, używanie dodatkowych komórek pamięci do mozolnego obliczania wartości wyrażeń arytmetycznych), dzięki czemu mogli bardziej skupić się na szukaniu rozwiązań optymalnych, a nie łatwych do zapisania. Kiedy proste jest proste, trudne staje się możliwe.

Nowe języki umożliwiały wyrażanie złożonych programów, ale nie ma róży bez kolców: wzrósł również stopień złożoności samych języków. Język maszynowy jest prosty. Składa się z kilkudziesięciu (czasem kilkuset) rozkazów. Każdy rozkaz realizuje prostą i łatwą do opisania operację. Rozkazy wykonywane są po kolei, wszystkie odstępstwa od tej zasady (np. rozkazy JMP lub CMP) są jasno widoczne w programie. Nie było łatwo używać tych języków, ale łatwo było je zrozumieć. Programy były duże i złożone, ale język prosty. Nowe języki sprawiły, że programy stały się krótsze i bardziej treściwe. Okazało się jednak, że opisanie, jak funkcjonuje język, nie jest już takie proste. Pułapki kryją się nawet w tak prostych konstrukcjach języka, jak doskonale

[4] Przyjęliśmy tutaj powszechnie stosowaną konwencję, że **rozkaz** oznacza polecenie dla komputera zapisane w kodzie maszynowym lub w języku mnemonicznym, a **instrukcja** jest poleceniem zapisywanym w językach wyższego poziomu, takich jak Fortran, Algol, Ada, Pascal, C, C++.

nam znane z matematyki wyrażenia algebraiczne. Wyrażenie (A + B – ABS (A – B)) / 2 jest[5] równe mniejszej spośród liczb A i B, zatem dla dowolnych liczb A i B zachodzi równość:

min(A, B) = (A + B – ABS(A – B)) / 2

W komputerach zazwyczaj nie można wykonywać obliczeń na dowolnych liczbach całkowitych. Jedna komórka współczesnej pamięci komputerowej może przechowywać tylko skończoną liczbę różnych wartości – w jednym bajcie można zapisać tylko 256 różnych wartości. Ponieważ w obliczeniach zazwyczaj nie występują dowolnie wielkie liczby, przyjmuje się, że wszystkie pojawiające się wartości liczbowe należą do pewnego ograniczonego zakresu, np. [–128..127] lub [–2147483648..2147483647]. Upraszcza to realizację obliczeń, ale wymaga dyscypliny, należy bowiem pilnować, aby wartości pojawiające się podczas obliczeń nie przekroczyły przyjętego zakresu.

Wyrażenie X + Y – Z to w zasadzie to samo, co X + Y + (– Z). Dzięki łączności dodawania, nie ma znaczenia, które dodawanie wykonamy najpierw, wynik powinien być taki sam:

X + Y – Z = (X + Y) – Z = X + (Y – Z)

Jeżeli jednak działania wykonujemy na liczbach z ograniczonego zakresu, to może nas spotkać przykra niespodzianka, np. dla X = 1, Y = 127, Z = 126 otrzymamy:

(X + Y) – Z = 128 – 126 → źle, nastąpiło przekroczenie zakresu [–128..127]
X + (Y – Z) = 1 + (127 – 126) = 1 + 1 = 2 ⟶ poprawnie

W zależności od tego, jak zinterpretujemy wyrażenie (A + B – ABS(A – B)) / 2,

((A + B) – ABS(A – B)) / 2 czy (A + (B – ABS(A – B))) / 2

dla A = 1 i B = 127 otrzymamy w pierwszym przypadku błąd, a w drugim poprawny wynik.

Tego rodzaju problemy stymulowały poszukiwania nowych metod precyzyjnego opisu języków programowania. Zaczęto zwracać uwagę na języki służące do opisu języków. Jednym z takich wynalazków jest metoda wykorzystana do opisu gramatyki języka **Algol 58**, zaproponowana przez Johna Backusa, znana

[5] W wielu językach programowania, wartość bezwzględna (moduł) |x| jest zapisywana ABS(x).

w informatyce pod nazwą **BNF** (Backhus-Naur Form, czyli Postać Backusa-Naura)[6]. W tej notacji wyrażenie można zdefiniować w następujący sposób:

wyrażenie::= wyrażenie znak wyrażenie | liczba | litera | ‘(’wyrażenie‘)’
znak::= ‘+’ | ‘–’ | ‘*’ | ‘/’ | ‘^’
liczba::= cyfra | liczba cyfra
cyfra::= ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’
litera::= ‘A’ | ‘B’ | ‘C’ | ‘D’ | ‘E’ | ‘F’ | ‘G’ | ‘H’ | ‘I’
| ‘J’ | ‘K’ | ‘L’ | ‘M’ | ‘N’ | ‘O’ | ‘P’ | ‘Q’ | ‘R’
| ‘S’ | ‘T’ | ‘U’ | ‘V’ | ‘W’ | ‘X’ | ‘Y’ | ‘Z’

Powyższy opis warto porównać z naiwnym, tekstowym opisem składni wyrażeń algebraicznych zamieszczonym kilka akapitów wcześniej. Słowa wzięte w cudzysłowy to **symbole** opisywanego języka. Słowa bez cudzysłowów to tzw. **metasymbole**, elementy używane do definicji języka. W notacji BNF metasymbole odpowiadają kategoriom elementów opisywanego języka. W tym przypadku kategoriami są: wyrażenia, działania, liczby, cyfry, litery itp. W tej notacji definicje metasymboli nazywa się **klauzulami**.

Klauzulę: wyrażenie::= wyrażenie znak wyrażenie | liczba | litera |
‘(’wyrażenie‘)’

należy czytać (interpretować) następująco:

Wyrażenie może być:
parą wyrażeń połączonych znakiem lub
liczbą lub
literą lub
wyrażeniem wziętym w parę nawiasów.

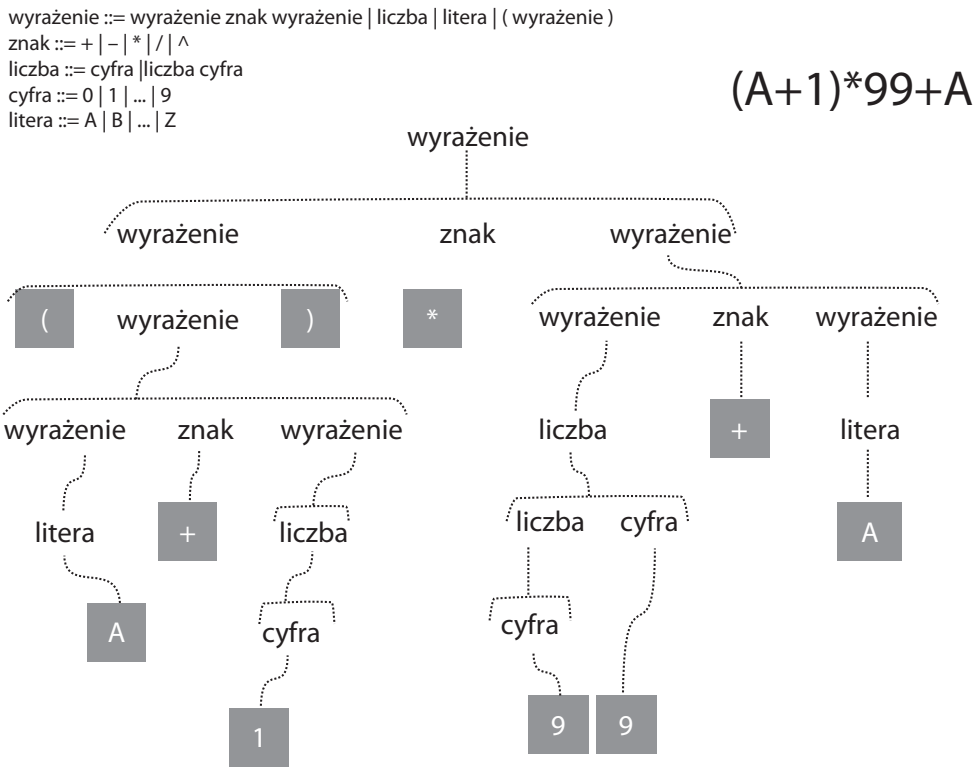
Z kolei klauzulę: liczba::= cyfra | liczba cyfra
należy czytać:

Liczba może być:
pojedynczą cyfrą
liczbą, po której następuje pojedyncza cyfra.

[6] Nie od razu zauważono, że Postać Backhusa-Naura jest bardzo podobna do metody zastosowanej przez Pāniniego do opisu gramatyki Sanskrytu w VI w p.n.e.

Notacja BNF rozwiewa wątpliwości związane z regułami łączenia symboli w wyrażenia języka. Proces tworzenia wyrażenia algebraicznego na podstawie powyższej gramatyki może przebiegać następująco:

(1) weź metasymbol 'wyrażenie'
(2) zastąp dowolny metasymbol dowolną z klauzul stojących po jego prawej stronie
(3) jeżeli są jeszcze jakieś metasymbole, wróć od punktu (2).

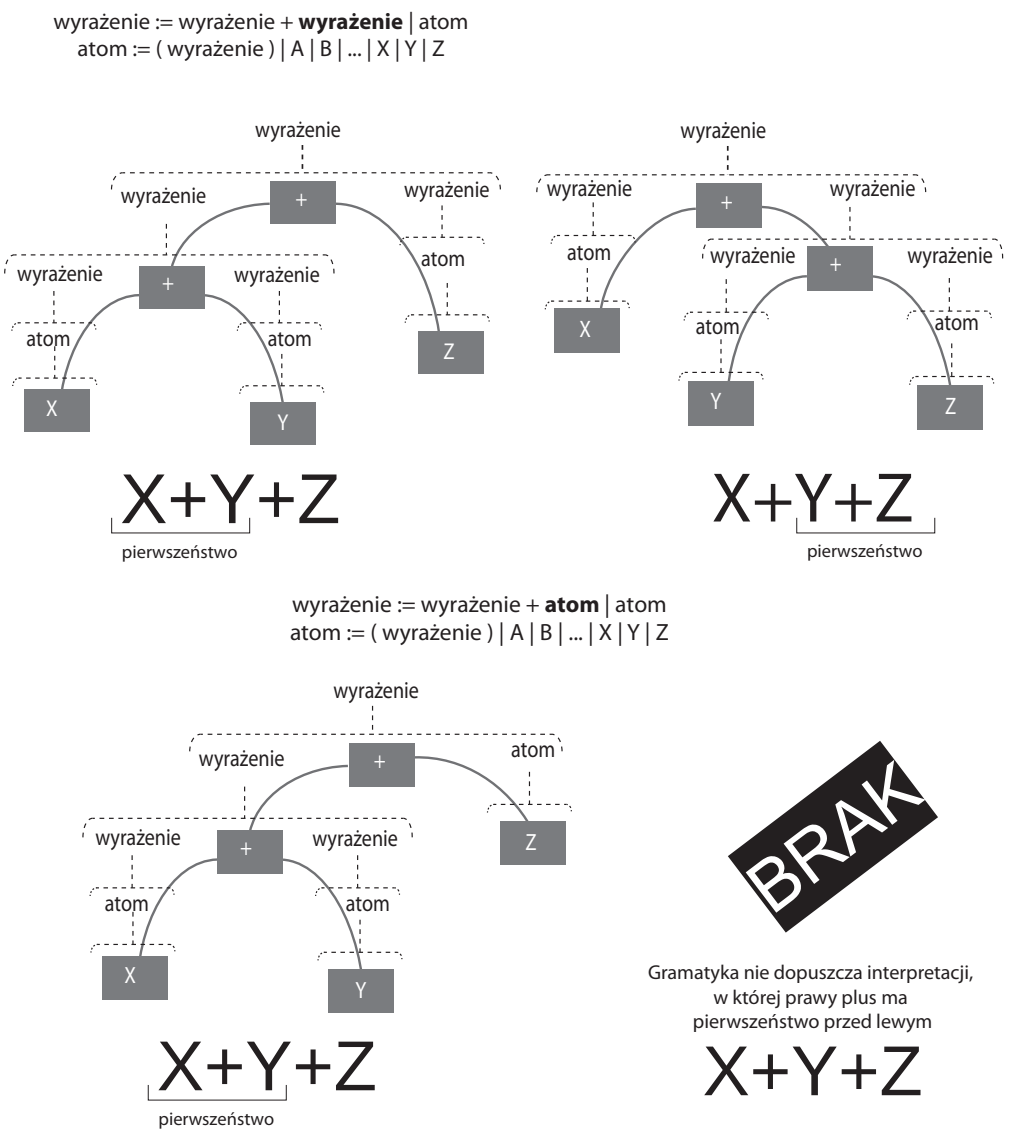


Rysunek 6. Tworzenie wyrażenia (A+1)*99+A na podstawie gramatyki. Ta gramatyka nie opisuje poprawne wyrażenia, ale nie sugeruje właściwej kolejności wykonania działań (np. pierwszeństwo mnożenia przed dodawaniem)

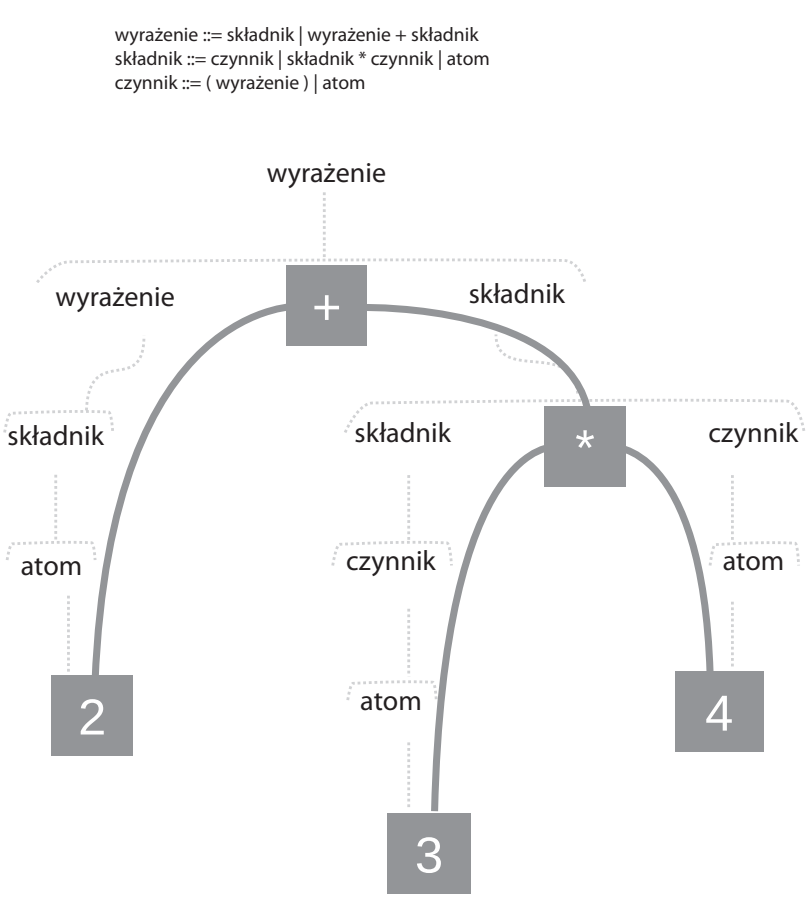
Na zapis w języku BNF można patrzeć z dwóch stron. Po pierwsze jest to opis zasad łączenia symboli pewnego (innego) języka. W powyższym przykładzie opisujemy zasady budowania prostych wyrażeń algebraicznych. Po drugie gramatyka w języku BNF może służyć do interpretacji wyrażeń języka. Na rysunku 7 pokazano dwie gramatyki BNF, opisujące proste wyrażenia.

Gramatyki różnią się jednym szczegółem (w pierwszej dopuszczamy dowolne wyrażenia po obu stronach znaku plus, a w drugiej prawe wyrażenie musi być

albo zmienną, albo wyrażeniem w nawiasach). Pierwsza gramatyka nie precyzuje więc kolejności wykonywania działań i wyrażenie X + Y + Z można w niej zinterpretować równie dobrze jako „najpierw X + Y, potem wynik + Z", jak i „najpierw Y + Z, potem X + wynik". Z kolei druga gramatyka dopuszcza wyłącznie tę pierwszą interpretację. Obie gramatyki opisują ten sam zbiór dopuszczalnych wyrażeń, ale druga gramatyka wymusza interpretację kolejności wykonywania działań. Taki mechanizm umożliwia bardzo precyzyjnie opisywać zachowanie programów, np. kolejność wykonywania obliczeń.



Rysunek 7. Wymuszanie kolejności obliczeń za pomocą gramatyki (w gramatyce na dole lewy plus wiąże mocniej)



Rysunek 8. Wymuszanie kolejności obliczeń za pomocą gramatyki (pierwszeństwo mnożenia przed dodawaniem)

Język Algol odniósł sukces, a notacja BNF stała się popularna. Możliwość zapisu programów za pomocą wyrażeń algebraicznych stała się standardem. Nowe, coraz lepsze języki mnożyły się jak grzyby po deszczu, a wraz z nimi specjalne programy (kompilatory), służące do tłumaczenia tych języków na kod maszynowy. W roku 1960 pojawił się **COBOL**, język programowania, który rozpalał nadzieje, że mit komputera rozumiejącego ludzki język się ziści. Postać programów w języku COBOL bardzo przypominała zdania w naturalnym języku angielskim. Twórcy języka wierzyli, że analitycy, managerowie i inni nie-programiści będą w stanie czytać i rozumieć programy. Niestety, okazało się, że rozdźwięk pomiędzy strukturą języka angielskiego a kodem maszynowym, do którego w końcu trzeba było tłumaczyć wyrażenia w języku COBOL, jest zbyt wielki. Pozorne podobieństwo języka COBOL do języka angielskiego sprowadzało nie-programistów na manowce, a programistom wcale nie ułatwiało pracy. W latach 70. XX wieku okazało się, że struktura programów w tym języku nie ułatwia ich analizy. Edsger Dijkstra, propagator metod programowania opartych

na hierarchicznej strukturze programów (której brakuje programom w języku COBOL), w roku 1975 zgromił COBOL, posuwając się do stwierdzenia, że „używanie języka COBOL kaleczy mózg, nauczanie tego języka powinno więc być uznane za przestępstwo". Pierwsza nadzieja na komunikację z maszyną w języku naturalnym nieco zgasła.

6. Ewolucja języków programowania

Stawało się powoli jasne, że różne osoby będą komunikować się z komputerem w różny sposób. Oprócz specjalistów rozumiejących mechanizmy działania maszyn i szkolonych w metodach ich programowania, pojawili się zwykli użytkownicy – osoby nieprzeszkolone w programowaniu, które dzięki coraz większej dostępności komputerów uzyskały do nich bezpośredni dostęp, by realizować zadania naukowe lub komercyjne. Ewolucja języków komunikacji z komputerem poszła w dwóch kierunkach.

6.1. Język powłoki – konwersacja z maszyną

W roku 1969 pojawił się system operacyjny **Unix**. Był dziełem niewielkiego zespołu zdolnych programistów pracujących dla firmy Bell Labs. **System operacyjny** to główny program sterujący pracą komputera, zarządzający jego zasobami (pamięcią wewnętrzną, pamięcią dyskową, czasem procesora) oraz umożliwiający uruchamianie innych programów, tzw. aplikacji. Twórcy systemu Unix zaprojektowali go tak, by główna część systemu była możliwie mała, a jak najwięcej zadań było realizowanych przez niewielkie, odizolowane programy narzędziowe. Funkcje systemu uruchamiało się z tzw. **programu powłoki** (ang. *shell*). Program powłoki współpracował z użytkownikiem na zasadzie konwersacji: użytkownik wpisywał polecenie, a program powłoki odpowiadał, wypisując informację za pomocą drukarki lub terminala ekranowego. Program powłoki udostępniał prosty język poleceń. Jeśli jednak użytkownik wpisał polecenie nierozpoznawane przez program powłoki, system próbował znaleźć na dysku program o tej nazwie, co nieistniejące polecenie i uruchomić go. Dzięki takiemu rozwiązaniu rozszerzenie programu powłoki o nowe polecenia było bardzo proste – wymagało tylko zainstalowania na dysku programów o odpowiednich nazwach. Ponieważ środowisko systemu Unix było przyjazne dla programistów, język powłoki systemu Unix (wraz z nazwami najpopularniejszych programów narzędziowych) stał się niezwykle popularny wśród zawodowych programistów. Język powłoki ewoluuje do dziś. Występuje w dwóch głównych odmianach (Bourne-shell i C-shell). Daleko mu do swobody porozumiewania się z komputerem HAL 9000, ale opiera się na podobnej zasadzie – operator prowadzi konwersację z systemem.

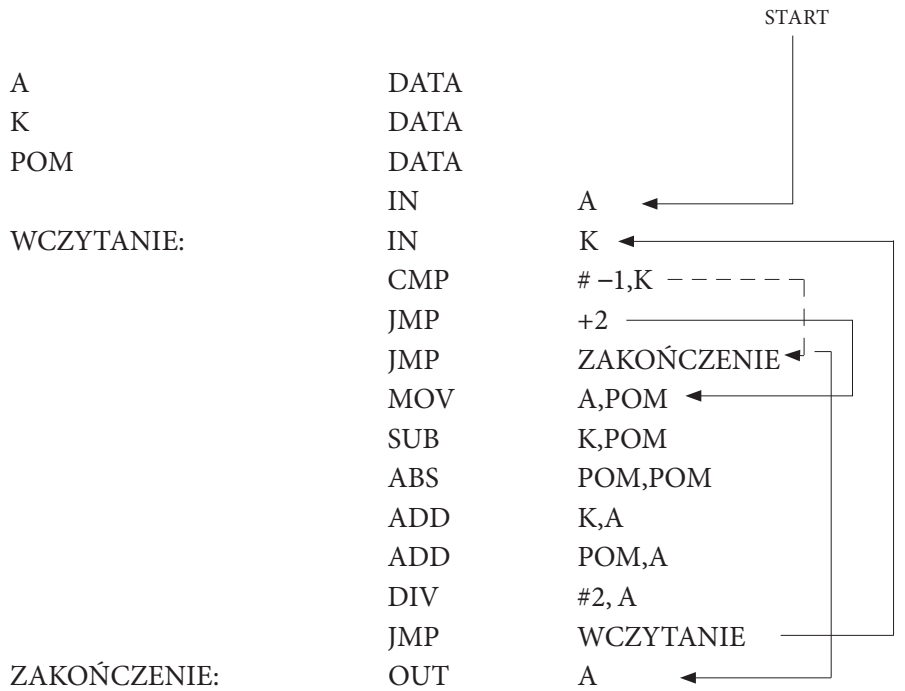
Sam program powłoki byłby jednak bezużyteczny, gdyby nie masa programów i programików, w które obrósł system Unix. Dziesiątki, setki, a z czasem tysiące małych i dużych programów wypełniających funkcje od najprostszych (zliczanie rozmiarów plików, wyszukiwanie frazy tekstowej, sortowanie) do bardzo złożonych (tłumaczenie programów na inne języki, skład tekstów, rozwiązywanie dużych układów równań) powstały i obsiadły system Unix jak rój pszczół.

Współczesne popularne systemy GNU/Linux mają budowę odziedziczoną po systemie Unix. Na system składa się duża liczba luźno powiązanych programów realizujących przeróżne funkcje, zarówno bardzo proste, jak i bardzo złożone. Na przykład system Ubuntu Linux zawiera ponad 38 tysięcy programów służących do wykonywania wszelkich wyobrażalnych zadań, które jest dziś w stanie wykonać komputer. Wiele z tych programów jest zbudowanych w ten sposób, że można nimi sterować z poziomu programu powłoki, często twórcy programów zachowują przy tym podobny styl komunikacji, różne programy obsługuje się zatem podobnie. Ta menażeria składa się na język komunikacji z maszyną. By rozpocząć konwersację w tym języku, wystarczy włączyć dowolny system GNU/Linux i uruchomić program Terminal.

6.2. Wojna z instrukcją Goto

Coraz większe komputery i coraz wygodniejsze języki programowania dawały coraz większe możliwości, a apetyty użytkowników rosły w miarę jedzenia. Pod koniec lat 60. XX wieku pojawiła się pierwsza mysz komputerowa, w latach 70. rozwinęły się interfejsy graficzne, stworzono gry zręcznościowe oraz arkusz kalkulacyjny. Stare nawyki umierają jednak powoli. Nowe języki programowania (C, Pascal) umożliwiały zwięzłe i eleganckie zapisywanie programów, jednak wielu programistów nie potrafiło się przestawić i skorzystać z tych udogodnień. Jedną z trudności w programowaniu za pomocą języka maszynowego są rozkazy powodujące zmianę miejsca wykonania kodu, tzw. **rozkazy skoku**. Na rysunku 9 pokazano program maszynowy opisywany wcześniej, z zaznaczonymi rozkazami skoków. Skoki są w swej naturze asymetryczne. Łatwo wypatrzeć początek skoku, ale trzeba się uważnie rozejrzeć by zauważyć, że np. rozkaz „MOV A,POM" jest zakończeniem jakiegoś skoku. Po uważnej analizie okolicznych rozkazów stwierdzimy, że faktycznie dwa wiersze wyżej znajduje się rozkaz „JMP +2", który powoduje skok do rozkazu „MOV A,POM". Stwierdzenie, czy w programie istnieją jakiekolwiek inne rozkazy skoku prowadzące do „MOV A,POM" wymaga przejrzenia... całego programu! To nie jest pro-

blemem, gdy program jest krótki, ale im dłuższy, tym trudniej mieć pewność (a rozmiar współczesnych programów sięga milionów rozkazów maszynowych). Nie koniec jednak na tym.



Rysunek 9. Program maszynowy z oznaczonymi instrukcjami skoków

Rozkazy skoków na rysunku 9 oznaczono strzałkami. Strzałka prowadząca od JMP WCZYTANIE do WCZYTANIE biegnie pod prąd normalnego kierunku wykonywania rozkazów, przez co pewna grupa rozkazów jest wykonywana wielokrotnie. Taką grupę rozkazów nazywamy **pętlą**. Dzięki pętli program może działać dla ciągów danych o różnej długości, powtarzając tę samą sekwencję rozkazów dla kolejnych elementów. Pętla daje komputerom moc. Bez pętli programowanie jest proste, ale słabe. Wielu programów nie da się napisać bez pętli. Pętle są możliwe dzięki rozkazom rozgałęzienia (CMP), inaczej wykonanie rozkazów pętli trwałoby w nieskończoność – rozgałęzienie jest potrzebne, aby zakończyć wykonanie pętli. Stwierdzenie, czy w programie wykonanie pętli kończy się poprawnie, jest podstawowym i czasami bardzo trudnym problemem przy pisaniu programów. Pod koniec lat 60. minionego stulecia teoretycy programowania, Robert Floyd oraz Antony Hoare, opracowali precyzyjne metody wnio-

skowania, pomagające stwierdzić, czy pętle w programie kończą się poprawnie. I tu wracamy do strzałek: strzałka od JMP ZAKOŃCZENIE i strzałka od JMP WCZYTANIE przecinają się. Jeżeli strzałki skoków mają przecięcia, to metody wnioskowania o pętlach bardzo się komplikują. Ale sprawa nie jest beznadziejna. Już w połowie lat 60. inni teoretycy (Corrado Böhm i Giuseppe Jacopini) wykazali, że każdy program można tak napisać, żeby strzałki się nie przecinały! Wystarczy chcieć i mieć taki nawyk. Najwyraźniej nie był to nawyk powszechny w roku 1968, kiedy holenderski informatyk i luminarz Edsger Dijkstra wylał swoją frustrację w artykule *Go To Statement Considered Harmful* (pl. Instrukcja Go To jest niebezpieczna). Wiele ówczesnych języków programowania (np. Fortran, Algol) zawierało instrukcję **Go To**, która była odpowiednikiem rozkazu skoku w języku maszynowym. Dijkstra argumentował, że jej nieodpowiedzialne użycie powoduje powstawanie programów złożonych i trudnych do analizy.

Artykuł Dijkstry był elementem szerszego ruchu znanego pod nazwą **programowanie strukturalne**, nawołującego do ograniczenia środków wyrazu przy pisaniu programów w celu polepszenia ich czytelności i jakości. Takie zmiany wymagają odejścia od przyzwyczajeń i nie dokonują się szybko. Realny wpływ nawoływań zwolenników programowania strukturalnego na kształt nowych języków programowania dało się zauważyć dopiero w latach 90.

To może wydawać się dziwne, ale wszystkie dominujące języki programowania począwszy od lat 60. po dzień dzisiejszy, są do siebie bardzo podobne. Wynika to z faktu, że do końca XX wieku w zasadzie wszystkie wytwarzane przemysłowo komputery miały konstrukcję zasadniczo nieodbiegającą od modelu von Neumanna.

Tak jak w większości języków naturalnych można wyróżnić rzeczowniki, czasowniki, przymiotniki i przysłówki, tak w większości współczesnych języków programowania można wyróżnić cztery podstawowe kategorie gramatyczne:

- **zmienne** (zazwyczaj oznaczane literami np. A, B... lub słowami), które reprezentują fragmenty pamięci komputera i służą do przechowywania danych;
- **wyrażenia** (zapisywane, jak wyrażenia algebraiczne w matematyce), które opisują niewielkie fragmenty obliczeń, np. (A + B + ABS(A – B)) / 2;
- **instrukcje przypisania** (zapisywane za pomocą znaków =,:= lub ←), które powodują przypisanie obliczonych wartości wyrażeń zmiennym, którym odpowiadają komórki pamięci;
- **instrukcje sterujące** (**if**, **goto**, **while**, **for**, **case**, **switch**, **call**...), które mają funkcje podobne do skoków i rozgałęzień w języku maszynowym, tj. przesądzają o tym, które fragmenty kodu będą wykonywane.

Zwolennicy programowania strukturalnego postulowali porzucenie nawyku tworzenia pętli za pomocą instrukcji **goto**. Współczesne języki programowania

na ogół zawierają pętlę **while,** którą można wykorzystać zamiast **goto** – gramatyka tej pętli ma postać:

instrukcja::= **while** warunek (wyrażenie logiczne) **do** blok |...
blok::= instrukcja | { ciąg-instrukcji }
ciąg-instrukcji::= instrukcja | ciąg-instrukcji **;** instrukcja

Wykonanie tej pętli polega na sprawdzeniu prawdziwości warunku i, jeśli jest spełniony, wykonaniu instrukcji oraz powrotu do wykonania pętli od nowa. Poniżej jest pokazana relacja między instrukcją **while** a **goto**.

```
while (WARUNEK) {
        INSTRUKCJE
}
```

```
pętla:
if (not WARUNEK) goto koniec;
… INSTRUKCJE …
        goto pętla
koniec:
```

zagnieżdżone pętle	zazębione pętle
… A: … … B: … … JMP B … JMP A …	… A: … … B: … … JMP A … JMP B …

while (X < N) {
…
while (Y < M) {
…
}
}
	gramatyka instrukcji **while** uniemożliwia zapisanie zazębionych pętli

Rysunek 10. Zazębiające się pętle

Na rysunku 10 zilustrowano niekorzystne zjawisko zachodzenia na siebie dwóch pętli w programie maszynowym. Takie zazębianie się pętli utrudnia analizę (strzałki skoków przecinają się). Pętla **while** uniemożliwia zazębienie się dwóch pętli.

Instrukcję **goto** wyłączono z języka Java, nie ma jej także w języku Python. Wiele innych współczesnych języków wciąż zawiera instrukcję **goto**, np. ogłoszony w roku 2000 język C#.

6.3. Podprogramy – rozszerzanie języków

```
void main() {
   int A, K;
   scanf(„%d”, &A);
wczytanie:
   scanf(„%d”, &K);
   if (K == – 1) goto koniec;
   A = (A + K + abs(A – K)) / 2;
   goto wczytanie;
zakończenie:
   printf(„%d\n”, A);
}
```

```
int wprowadź_liczbę() {
   int liczba;
   scanf(„%d”, &liczba);
   return liczba;
}

int wyprowadź_liczbę(int liczba) {
   printf(„%d\n”, liczba);
}

int minimum(int x, int y) {
   return (x + y + abs(x – y)) / 2;
}

void main() {
   int A, K;
   A = wprowadź_liczbę();
   while ((K = wprowadź_liczbę()) != –1)
{
      A = minimum(A, K);
   }
   wyprowadź_liczbę(A);
}
```

Rysunek 11. Niestrukturalna i strukturalna wersja programu, służącego do obliczania maksimum w języku C; użycie podprogramów zwiększa rozmiar kodu źródłowego, ale bardzo podnosi czytelność

Drugim postulatem programowania strukturalnego było posługiwanie się **podprogramami**. Wiele języków programowania ma możliwość nazwania i wyodrębnienia fragmentu kodu, który później może zostać użyty przez wymienienie nazwy. Jest to ta sama filozofia, która leży u podstaw systemu Unix – system (w tym przypadku język) składa się z małej liczby podstawowych elementów, do których użytkownik może dodać nowe elementy, opisane za pomocą tych już istniejących. Na podprogramy można patrzeć jak na mechanizm umożliwiający rozszerzenie języka o nowe wyrażenia.

Na rysunku 11 ilustrujemy postulaty programowania strukturalnego w praktyce. Program po lewej stronie jest napisany bez poszanowania zasad programowania strukturalnego, a ten z prawej korzysta ze strukturalnej pętli **while** oraz z podprogramów. Wszystkie, nawet bardzo proste operacje, zostały zamknięte w podprogramy, którym nadano czytelne nazwy. Nawet osoba nierozumiejąca do końca użytych w programie wyrażeń (np. scanf albo int) jest w stanie zrozumieć program po prawej, właśnie dzięki obecności zrozumiałych nazw podprogramów (wprowadź_liczbe, wyprowadź_liczbę, minimum).

Mechanizm podprogramów zapewnia nie tylko czytelność, ale także umożliwia często używane kawałki kodu napisać raz i odłożyć do ponownego użytku. W latach 80. XX wieku popularne stały się tzw. **biblioteki podprogramów** – zbiory podprogramów, przeznaczonych do konkretnych zadań (np. obliczeń na macierzach, obliczeń statystycznych, generowania grafiki trójwymiarowej itp.). Wielkim przebojem okazała się biblioteka STL dla języka C++, a niektóre języki zdobyły olbrzymią popularność dzięki łatwo dostępnym rozległym bibliotekom (np. Perl, Java).

Szacowana liczba języków programowania to dziś kilka tysięcy. W powszechnym użyciu jest jednak tylko kilkanaście z nich, reszta to języki wymarłe niszowe albo nowatorskie.

7. I co dalej?

Firma Apple 28 kwietnia 2010 roku kupiła firmę Siri, producenta oprogramowania o tej samej nazwie. Oprogramowanie Siri to automatyczny asystent, system adaptacyjnie rozpoznający ludzką mowę, zdolny do wykonywania prostych zadań związanych z wyszukiwaniem informacji i rezerwacją terminów w kalendarzu. Jesteśmy chyba wciąż bardzo daleko od komunikowania się z komputerami z taką swobodą, jak astronauci w *Odysei Kosmicznej 2001* Stanleya Kubrika. Pomiędzy językami naturalnymi, w którym komunikujemy się ze współplemieńcami, a językami programowania, w których instruujemy komputery, zieje wciąż olbrzymia otchłań. Programowanie cały czas wymaga dużej ostrożności i solidnego treningu. Z drugiej strony – postęp nie ustaje. Oto kilka pomysłów, które pojawiły się na przestrzeni ostatnich pięćdziesięciu lat:

Języki obiektowe

Wymyślone na początku lat 70. ubiegłego wieku i rozpropagowane w latach 80. były udaną próbą rozwiązania problemu nadmiernych zależności pomiędzy komponentami dużych systemów. Do tej rodziny należą języki Simula, Smalltalk, C++, Java, C# i działający w każdej przeglądarce internetowej JavaScript.

Języki funkcyjne

Powstałe w połowie lat 70. zrywają z tradycyjnym modelem *zmienne/wyrażenia/instrukcje*. W niektórych językach funkcyjnych w ogóle nie ma instrukcji. To pomaga uzasadniać poprawność napisanych w nich programów. Te języki są ważne również z tego powodu, że coraz więcej komputerów produkowanych w XXI wieku nie jest już zgodnych z tradycyjnym modelem von Neumanna, w którym instrukcje są wykonywane po kolei. Współczesne komputery osobiste wykonują wiele wątków obliczeń równolegle, choć wciąż te możliwości są słabo wykorzystane, m.in. ze względu na brak odpowiednich języków programowania. Wydaje się, że języki funkcyjne mogą lepiej nadawać się do programowania takich komputerów. Do tej rodziny należą: Lisp, SML, Haskell, Ocaml.

Języki skryptowe

Przedkładają wygodę pisania i uruchamia kodu oraz łatwość użycia nad wydajność. Zyskały popularność w latach 90., kiedy wydajność komputerów osobistych wzrosła na tyle, by wolniejsze wykonanie programów w językach skryptowych nie było uciążliwe dla użytkownika. Jeśli program napisany np. w języku C działa sekundę, to jego wersja napisana w języku Python może wymagać nawet 10 sekund. Szybkie komputery sprawiły, że języki skryptowe przestały się w dostrzegalny sposób ślamazarzyć. Do tej rodziny należą Python, Perl, Lua, Ruby.

Języki powłoki

To języki zintegrowane ze środowiskiem systemu operacyjnego. Można w nich pisać programy, ale np. oprogramowywanie obliczeń nie jest specjalnie wygodne. Spisują się natomiast bardzo dobrze jako języki dialogu operatora z systemem operacyjnym oraz spoiwo do sklejania ze sobą mniejszych programów, przekazywaniem danych między nimi i automatyzacją nudnych zadań. Wśród nich: Bash, TCSH, ZSH.

Języki przetwarzania danych

Służą do operacji na zbiorach danych[7]. To szeroka i różnorodna rodzina, jej członkiem jest zarówno leciwy, acz bardzo popularny język SQL do tworzenia

[7] Więcej na temat baz danych i ich języków można przeczytać w rozdziale *Homo informaticus colligens, czyli człowiek zbierający dane.*

i wyszukiwania informacji w bazach danych, ale także język Xpath do wyszukiwania informacji w ustrukturyzowanych plikach tekstowych. Do tej rodziny można również zaliczyć język akceptowany przez wyszukiwarkę Google, pozwalający modyfikować lub precyzować wyniki wyszukiwania.

Języki pomocnicze

Powstało wiele małych wyspecjalizowanych języków programowania, służących do jednego konkretnego celu. Na przykład język Sed służy do prostych operacji tekstowych. Pewne jego operacje rozumie nawet komunikator Skype! Jeżeli pomylisz się w czacie Skype (np. napiszesz „gura" zamiast „góra"), to już po wysłaniu komunikatu możesz wpisać w czat instrukcję języka Sed s/gura/góra/, a błąd w poprzednim komunikacie zostanie poprawiony! Inne języki z tej grupy to Awk i wyrażenia regularne (*regular expressions*).

Istnieje też pokaźna grupa sztucznych języków związanych z informatyką, które nie służą do pisania programów. Wśród nich jest rodzina języków do opisu wyglądu dokumentów i stron WWW (HTML, CSS, XLST), rodzina języków do opisu dokumentów przeznaczonych do druku (TeX, PostScript, Troff), języki do opisu trójwymiarowych scen dla programów generujących grafikę komputerową (np. PovRay), języki do opisu zachowania i topologii układów scalonych (Verilog, VHDL) itp.

Istniejących języków programowania jest kilka tysięcy. Ich ewolucja trwa od pół wieku. Z upływem czasu pojawia się coraz więcej wątpliwości, czy któryś z nich kiedykolwiek zbliży się do języka ludzkiego, umożliwiając człowiekowi rozmowę z maszyną tak, jak w *Odysei Kosmicznej 2001*. Być może tak się kiedyś stanie. Rok 2001 był jednak terminem zbyt ambitnym.

Epilog – Których języków programowania się uczyć?

Współczesny programista nie musi znać wielu języków programowania. Zazwyczaj dobrze zna kilka, a kilkanaście zna pobieżnie. Nie chodzi jednak o ilość. Wyobraź sobie, że uczysz się języków naturalnych: polskiego, rosyjskiego, słowackiego. Czy znając te języki warto uczyć się ukraińskiego? Zapewne nie, bo jest on podobny do trzech języków, które już znasz. Jeśli znasz C++, Javę i Pascala, zainwestuj w naukę języka, który jest istotnie odmienny. Oto subiektywna propozycja portfolio dla informatyka:

- przynajmniej jeden wysokowydajny popularny obiektowy język kompilowany (np. Java, C++, C#, Ada),
- język maszynowy jakiegoś procesora, bo warto wiedzieć, jak programy działają „na samym spodzie",
- język C, bo jest to wciąż najpopularniejszy język do pisania systemów operacyjnych i sterowników,

- język funkcyjny (Haskel, Standard ML, Ocaml),
- popularny wygodny język skryptowy (np. Python, Ruby, Lua) do pisania programów, w których poprawność jest dużo ważniejsza niż wydajność obliczeń (np. serwery WWW),
- język używanej powłoki (np. język powłoki Windows, Bash, ZSH, TCSH),
- języki opisu dokumentów WWW (obecnie HTML i CSS, ew. XML),
- dominujący język do implementacji interfejsów użytkownika w aplikacjach WWW (obecnie JavaScript),
- języki do operacji na tekstach, przydatne przy automatyzacji małych codziennych zadań (Sed, Awk, Perl),
- język o silnym systemie modułów (np. Modula-2, Standard ML, Ada),
- język skryptowy o dużej bibliotece (np. Python, Perl), do szybkiej implementacji zadań prostych, ale żmudnych (np. ściągnij stronę WWW przez HTTPS, zanalizuj jej treść w języku HTML i podaj rozmiar w znakach pisarskich),
- dominujący język zapytań baz danych (obecnie SQL),
- dominujący język programowania na wybranej dużej platformie mobilnej (np. iOS → Objective-C, Android → Java).

Oprócz tego, tak jak humaniście przystoi znać języki klasyczne łacinę i grekę, tak programiście wypada znać Lisp i Smalltalk.

Literatura

1. Dijkstra E.W., *Go To Statement Considered Harmful*, „Comm. ACM" 11(3) 1968
2. Dijkstra E.W., *Umiejętność programowania*, WNT, Warszawa 1978
3. Harel D., Feldman Y., *Algorytmika. Rzecz o istocie informatyki*, WNT, Warszawa 2008
4. Von Neumann, J., *First Draft of a Report on the EDVAC*, 1945, http://systemcomputing.org/turing%20award/Maurice_1967/TheFirstDraft.pdf
5. Wirth N., *Algorytmy + Struktury Danych = Programy*, WNT, Warszawa 1999

Grzegorz Jakacki ukończył studia na Wydziale Matematyki i Informatyki Uniwersytetu Warszawskiego z tytułem magistra informatyki, przedstawiając pracę magisterską dotyczącą przyrostowego sposobu opisu semantyki języków programowania. Podczas studiów uczestniczył w pracach jury Olimpiady Informatycznej i Zawodów w Programowaniu Zespołowym. Od roku 2000 pracował jako programista, projektant oprogramowania i tutor w USA, Chinach i Polsce. W latach 2000-2001 oraz 2008-2009 był członkiem zespołu opracowującego projekt i implementację fragmentu języka Verilog, służącego do opisu własności układów scalonych. W latach 2001-2006 prowadził społeczny projekt rozwoju kompilatora czołowego OpenC++. Pracował nad systemem analizy kodu źródłowego w języku SystemC (2001-2003) oraz nad interpreterem języka Verilog-AMS (2003-2005). Przełożył na język polski książki: *Nowoczesne programowanie w C++* oraz *Sztuka Programowania* (Tom I). Jest założycielem i prezesem firmy Codility Ltd., zajmującej się automatycznym testowaniem umiejętności poprawnego programowania. Sekretarz społecznego inkubatora technologicznego Warszawski Hackerspace. Prowadzi zajęcia z praktyki programowania na Uniwersytecie Warszawskim.

jakacki@codility.com