Historyczne języki programowania

Nil novi sub sole

twórca Marek Materzok

Email: marek.materzok@tilk.eu

06.06.2006

1 Wstęp

Wraz z pojawieniem się idei komputera konieczne stało się wprowadzenie jakiegoś sposobu zapisu - języka - umożliwiającego zarówno opisanie algorytmu, który chcemy na komputerze wykonać, jak i wprowadzenie go do komputera. Dotychczas znany język matematyki nie był wystarczający z dwóch powodów. Po pierwsze, notacja matematyczna jest deklaratywna - mówi o pewnych obiektach, ale nie zawiera informacji ani o efektywnym sposobie ich skonstruowania w sposób algorytmiczny, ani o sposobie ich zapisu w pamięci komputera. Po drugie, jest to zapis wysokopoziomowy i pełny skrótów, czytelny wprawdzie dla człowieka, lecz zupełnie pozbawiony sensu dla maszyny.

Początkowo opracowano dla komputerów kod binarny, zapisywany na kartach czy taśmach perforowanych, który w zwięzły i łatwy do odczytania przez komputer sposób opisywał algorytm. Jednak taki zapis miał wady. Kod binarny był ściśle związany z konkretnym modelem maszyny, dla którego był przeznaczony; przeniesienie programu na inny model oznaczało konieczność przepisania go od nowa. Binarny zapis jest również całkowicie nieczytelny dla człowieka. Asemblery, będące tekstowym zapisem kodu binarnego, niewiele poprawiły sytuację. Dlatego koniecznym stało się utworzenie czegoś nowego, nieznanego wcześniej - wysokopoziomowego języka programowania.

Obecnie wysokopoziomowe języki programowania są tak powszechne, że nikt nie wyobraża sobie pisania jakiegokolwiek nietrywialnego programu w asemblerze. Co więcej, coraz więcej osób ma kontakt z tymi językami od młodości, nawet przy braku znajomości zaawansowanej matematyki. Mało kto zastanawia się jednak, dlaczego języki te wyglądają tak, jak wyglądają. Okazuje się, że u sedna niewiele jest w nich nowego, są tylko rozwinięciem idei wczesnych języków programowania.

Wybrałem cztery różne języki, sformułowane przez różnych ludzi w różnych celach: Plankalkül, Fortran, Algol i LISP. Aby znaleźć motywację dla ich postaci, przeanalizowałem okoliczności ich powstania. Następnie przyjrzałem się im głębiej, aby znaleźć podobieństwa i różnice zarówno między nimi, jak i z językami używanymi współcześnie. Wybór jest dość subiektywny; znakomitym przeglądem istotnych historycznch języków jest [Sam72].

2 Wybór języków programowania

2.1 Plankalkül - język wyprzedzający epokę

2.1.1 Historia powstania

Konrad Zuse urodził się w Berlinie 22 czerwca 1910. Był bardzo kreatywnym człowiekiem, w młodości lubił budować różnego rodzaju mechanizmy. Postanowił studiować inżynierię budowlaną: „Inżynier budowlany wydaje mi się idelną kombinacją inżyniera i artysty."[1] Zaczął myśleć o automatycznym przeprowadzaniu obliczeń już w roku 1934. W roku 1936 definiuje obliczanie w następujący sposób: „Tworzenie z danych wejściowych nowych danych według pewnego przepisu."[2] Uważa, że komputery powinny być programowalne. W latach 1936-1941 zbudował słynne oparte na przekaźnikach, liczące w systemie binarnym komputery Z1, Z2 i Z3.

1. [Cle03, s.2]
2. [Zus, część 2]

Od roku 1942 - w czasie, gdy pracował nad komputerem Z4 - zaczął zastanawiać się nad uniwersalnym sposobem ścisłego formułowania zadań (planu obliczeń, „Rechnenplan") dla komputera. Maszyny wykonujące tak sformułowane zadania nazywał maszynami logistycznymi („Logistische Maschinen"), aby przeciwstawić je komputerom wykonującym wyłącznie obliczenia numeryczne. Te prace doprowadziły go do sformułowania w roku 1945 języka Plankalkül.

Ponieważ gdy Konrad Zuse formułował język, nie istniały jeszcze komputery z programem w pamięci, nikt również nie myślał o kompilatorach, Plankalkül należy traktować raczej jak język ścisłego opisu algorytmów przeznaczony dla ludzi, a nie komputerów. W ramach prac nad językiem Zuse zapisał w nim wiele ciekawych programów, najbardziej złożonym z nich był program szachowy.

2.1.2 Funkcjonalność

Język Plankalkül nie był pomyślany pod kątem kompilowania go do kodu binarnego maszyn von Neumannowskich, przez co okazało się, że jego implementacja jest bardzo trudna[3]. Jego zapis również nie jest odpowiedni do łatwego wprowadzania do komputera. Jednak pod względem siły wyrazu Plankalkül znacząco przewyższa całą resztę grona wczesnych języków praktycznych.

Plankalkül jest językiem silnie typowanym - każda zmienna ma ściśle określony typ. Jedynym podstawowym typem danych jest bit - *Ja-Nein-Werte* - oznaczany $S0$. Inne typy danych - w szczególności listy i rekordy - są budowane przez składanie typów mniejszych. Dopuszczalny jest polimorfizm typowy - program może operować na strukturach zbudowanych z dowolnych typów. Zmienne typowe oznacza się za pomocą małych liter alfabetu greckiego, jak σ lub τ.[4] Liczby binarne są reprezentowane jako listy bitów, zaś liczby dziesiętne jako listy cyfr dziesiętnych zapisywanych w czterech bitach.

Programy zapisywane są w specyficznych tabelach, zawierających kilka wierszy. Pierwszy wiersz zawiera instrukcje, wiersz V - numery zmiennych (jeśli są potrzebne; dla wygody język pozwala na używanie zmiennych nazwanych), wiersz K - indeksy składowych w typie złożonym, wiersz S - informację o strukturze zmiennej. Niepotrzebne wiersze można pominąć.

Każdy program zaczyna się od *Randauszug*, określającego zmienne wejściowe i wyjściowe programu. Programy mogą być numerowane i odwoływać się do siebie wzajemnie. Przykładowy Randauszug może wyglądać następująco:[5]

	$R1.2($	V	$)$	$\Longrightarrow$	R
V		0			0
S		$m \times 1.n$			0

Zapis ten oznacza, że program 2 z grupy programów 1 potrzebuje jako argument jednej zmiennej $V0$ o strukturze $m \times 1.n$ - co oznacza, że jest zbudowana z m zmiennych typu $1.n$ - list n bitów. Wynikiem programu będzie zmienna $R0$ o strukturze $S0$ - jeden bit. Zmienne wewnętrzne programu nie są definiowane jawnie, nazywane są albo $Z0$, $Z1$, ... - analogicznie do wejść i wyjść - albo za pomocą małych liter, jak i lub j. Zmienne wewnętrzne działają tak, jak zmienne lokalne funkcji w językach współczesnych.

Programy w języku Plankalkül są strukturalne - nie występuje instrukcja *goto*, instrukcje są grupowane w bloki za pomocą nawiasów kwadratowych, separatorem instrukcji jest pionowa kreska. Instrukcja przypisania zapisywana jest za pomocą strzałki $\Longrightarrow$ i oznacza zwartościowanie wyrażenia po lewej stronie i przypisanie go do zmiennej po prawej - kolejność odwrotna, niż współcześnie używana w popularnych językach. Przykład:

	V	$+$	5	$\Longrightarrow$	Z	$+$	$\Longrightarrow$	Z
V	1				0			0
K								0
S	$1.n$				$1.n$			0

3. Pierwszy kompilator języka Plankalkül powstał dopiero w roku 2000. Można go zobaczyć na stronie internetowej: http://www.zib.de/zuse/Inhalt/Programme/Plankalkuel/. Znajdują się tam również dokumenty związane z tym językiem.

4. [FB72, s.679]

5. Wzięte z [Cle03, s.5]

Program ten najpierw odczyta wartość drugiej zmiennej wejściowej programu o strukturze $1.n$, doda do niej 5 i zapisze wynik w zmiennej wewnętrznej $Z0$, po czym zamieni bit 0 zmiennej $Z0$ na 1 ($+$ w Plankalkülu oznacza 1).

Instrukcję warunkową oznacza się za pomocą strzałki $\rightarrow$. Poniższy przykładowy program oznacza: jeśli wartość zmiennej x jest większa niż zero, zmniejsz ją o 1, a następnie ustaw zmiennej $Z1$ wartość 1.

$$V \left| \begin{array}{c} i > 0 \rightarrow \\ \\ \end{array} \right. \left[\begin{array}{c|c} i - 1 \Longrightarrow i & + \Longrightarrow Z \\ & 1 \end{array} \right]$$

Pętla w języku Plankalkül ma ogólną postać $W[c \rightarrow i]$, gdzie c jest warunkiem pętli, a i treścią pętli. Pętla ta jest analogiczna do znanej powszechnie pętli *while*. Oprócz tego występuje „cukier syntaktyczny" w postaci pętli $W0(n)$ - n-krotnego powtórzenia, $W3(n,m)$ - pętli *for* przebiegającej po wybranej zmiennej, a także podobnie zdefiniowanych $W1$, $W2$, $W4$ oraz $W5$.[6] Dla przykładu, poniższy program obliczy sumę liczb od 1 do n.

$$\begin{array}{c|c|c|c} & 0 \Longrightarrow Z & W1(n) & \left[\begin{array}{c} Z + i + 1 \Longrightarrow Z \\ 0 \qquad\qquad\quad 0 \\ 1.m \qquad\qquad 1.m \end{array} \right] \\ V & 0 & & \\ S & 1.m & & \end{array}$$

Na pętlach funkcjonalność języka się nie kończy. Zawiera on w sobie operacje logiczne takie, jak operator przynależności $\in$ czy kwantyfikatory $(x)R(x)$ (dla każdego x prawdą jest $R(x)$) i analogiczny $(Ex)R(x)$ (istnieje x spełniające $R(x)$), gdzie $R(x)$ jest wyrażeniem logicznym. Innymi bardzo wysokopoziomowymi operacjami jest: filtrowanie $\hat{x}\,R(x)$ - utworzenie listy elementów spełniających $R(x)$, wyszukiwanie $\acute{x}\,R(x)$ - znalezienie elementu spełniającego $R(x)$, czy też użyteczne w pętlach $\mu\,x\,R(x)$ - znalezienie następnego elementu spełniającego $R(x)$ (porażka spowoduje przerwanie pętli).[7]

Bardzo bogatym opisem języka, napisanym przez samego Konrada Zuse, jest [Zus45] (w języku niemieckim), gdzie odsyłam głębiej zainteresowanych tematem.

2.1.3 Wnioski

Plankalkül, sformułowany na długo przed powstaniem Algolu 60 czy słynną debatą nad szkodliwością *goto*, jest językiem w całości strukturalnym. Co więcej, kod programu jest zamknięty w procedurach posiadających zmienne lokalne oraz jawnie zdefiniowane parametry (przekazywane przez wartość) oraz wartości zwracane. Plankalkül nie dopuszcza rekurencji.

W Plankalkülu nie istnieje pojęcie bajtu czy słowa, liczby mogą być zapisane w dowolnej ilości bitów. Co więcej, struktury danych nie muszą mieć ustalonej z góry liczby elementów. Czyni to język bardzo kłopotliwym w implementacji, jednak zwiększa jego siłę wyrazu. Zwłaszcza budowane w sposób strukturalny typy są interesujące, również dlatego, że przez długi czas nic analogicznego nie znajdowało się w innych językach. Nie zawierał ich nawet słynny Algol 60.

Wysokopoziomowe operacje na listach, takie jak zliczanie elementów, filtrowanie i przeszukiwanie według zadanego w miejscu predykatu, są bardzo niezwykłymi konstrukcjami, których odpowiedników można szukać dopiero w takich językach jak Lisp czy języki z rodziny ML. Wystąpienie takiej funkcjonalności w pierwszym udokumentowanym języku programowania, jakim jest Plankalkül, jest naprawdę zaskakujące, podobnie jak obecność polimorfizmu typów, który wszedł do powszechnego użytku dopiero z nastaniem C++.

Plankalkül jest niezwykłym językiem, wyprzedzającym swoją epokę. Uważam tak dlatego, że język ten zawiera bardzo wiele wysokopoziomowych konstrukcji, charakterystycznych raczej dla nowszych języków, a jego struktura jest bardzo elegancka. Jest to naprawdę niezwykłe, gdyż powstał on zanim potrzeba sformułowania wysokopoziomowego języka algorytmicznego została zauważona przez innych. Niewątpliwie świadczy to o geniuszu Konrada Zuse, autora języka. Zadziwiająca jest też liczba podobieństw między Plankalkülem a późniejszymi językami, powstałymi niezależnie od niego, co może świadczyć za tym, że pewne idee tylko czekały na odkrycie.

6. [Zus45, s.28]

7. [FB72, s.682]

2.2 Fortran - efektywny język algebraiczny

2.2.1 Historia powstania

Jak podaje John Backus, „Przed rokiem 1954 niemalże wszystkie programy były pisane w języku maszynowym bądź asemblerze."[8] Aby tworzyć efektywne programy, programiści musieli obchodzić wiele ograniczeń narzucanych przez istniejące wtedy komputery. Systemy „programowania automatycznego", które wtedy istniały, dostarczały tylko „sztucznego komputera" o znacznie wygodniejszym zbiorze instrukcji, które były później tłumaczone na kod konkretnej maszyny. Systemy te nie zdobyły większej popularności. Były one bardzo nieefektywne i wprowadzały pięcio- do dziesięciokrotne spowolnienie, które jednak było mało istotne, ponieważ pisane programy spędzały większość czasu na wykonywaniu bardzo powolnych operacji zmiennoprzecinkowych.

Pierwszym systemem algebraicznym - który nie był tylko „sztucznym komputerem", ale dostarczał też pewnych matematycznych abstrakcji - był system Laninga i Zierlera z MIT. Niektóre z idei w nim zawartych mogły wpłynąć na projekt Fortranu, jednak - jak podaje Backus - prace nad Fortranem rozpoczęły się jeszcze zanim poznał system Laninga w czerwcu 1954 roku, co wskazuje na to, że najważniejsze koncepcje stojące za obydwoma systemami zostały opracowane niezależnie. Backus nie wiedział też wtedy o języku Plankalkül.

Celem Fortranu było uproszczenie pisania programów do obliczeń numerycznych przy jak najmniejszej utracie wydajności. Motywacją do jego powstania było kilka istotnych czynników. Jednym był wysoki koszt tworzenia oprogramowania, związany z jego czasochłonnością i kłopotliwym, acz nieuniknionym debuggingiem. Innym było powstanie komputera z koprocesorem matematycznym (był to IBM 704), który przyspieszył operacje zmiennoprzecinkowe tak bardzo, że nieefektywność wcześniejszych systemów „programowania automatycznego" stała się wyraźna.

W związku z wymaganiami dotyczącymi wydajności projekt języka stał się sprawą drugorzędną. Jak napisał Backus, „po prostu wymyślaliśmy język w miarę postępu prac."[9] W wyniku powstał język mało elegancki, ale jego kompilator, ukończony w kwietniu 1957, spełniał postawione mu wymagania. Fortran szybko zdobył popularność, był intensywnie rozwijany, a jego uwspółcześniona wersja (najnowszą definicją języka jest Fortran 2003) jest używana po dziś dzień.

2.2.2 Funkcjonalność

Fortran jest typowym „językiem von Neumannowskim", zaprojektowanym wokół architektury komputera, dla którego jest przeznaczony. Nie posiada struktury; programy są ciągiem numerowanych wierszy, między którymi wykonanie może przeskakiwać w dowolny sposób. Każdy wiersz zawiera dokładnie jedną instrukcję.

Najważniejszą instrukcją, jak w każdym języku imperatywnym, jest instrukcja przypisania, tutaj nazwana „formułą arytmetyczną". Przypisywana wartość jest typu albo całkowitego, albo zmiennoprzecinkowego, możliwe jest jednak mieszanie wartości różnych typów w jednym wyrażeniu. Zmienne mają stały typ. Co interesujące, zmienne całkowite odróżnia się od zmiennoprzecinkowych tym, że ich nazwy zaczynają się od liter `I`, `J`, `K`, `L`, `M` lub `N`. Zmienne mogą również reprezentować jedno-, dwu- lub trójwymiarową tablicę - w takim przypadku konieczne jest wcześniejsze zadeklarowanie, że zmienna jest tablicą, za pomocą instrukcji `DIMENSION`. Indeks w tablicy zapisuje się w nawiasach.

W wyrażeniach, oprócz standardowych operacji arytmetycznych, można używać wywołań funkcji. Dostępny był pewien zbiór funkcji wbudowanych, takich jak wartość bezwzględna, maksimum i minimum. Mimo tego, że nie jest to opisane w [IBM56], w języku Fortran I możliwe było definiowanie własnych funkcji, które składały się z jednego wyrażenia, nie zawierały natomiast instrukcji.[10] Istniała możliwość rozbudowania języka o dodatkowe podprogramy, których można było używać jako funkcje, jednak było to kłopotliwe. Podprogramy i osobna kompilacja wydzielonych części programu zostały wprowadzone dopiero w Fortranie II.[11]

8. [Bac78b, s.165]
9. [Bac78b, s.168]
10. [Bac78b, s.173]
11. [IBM63, s.58]

Instrukcje sterujące Fortranu są bardzo nietypowe. Oprócz instrukcji skoku bezwarunkowego `GO TO` istnieją dwa różne skoki wielokierunkowe. Instrukcja postaci `GO TO n, (w1,...,wm)` oznacza skok do jednego z wierszy `w`, którego numer jest wartością przypisaną do zmiennej całkowitej n za pomocą specjalnej instrukcji `ASSIGN`. Natomiast instrukcja `GO TO (w1,...,wn), n` oznacza skok do wiersza w_i, gdzie i jest wartością zmiennej całkowitej `n` przypisaną zwykłą formułą arytmetyczną.

Instrukcja skoku warunkowego również jest równie nietypowa. Ma ona postać `IF (a) n1, n2, n3`, gdzie a jest dowolnym wyrażeniem, a wartości n są numerami wierszy. Oznacza ona skok pod jeden z trzech wskazanych wierszy jeśli wartość wyrażenia będzie mniejsza, równa bądź większa od zera.

Natomiast instrukcja pętli *for* z Fortranu jest kwintesencją nieczytelności. Zapisuje się ją `DO n i = m1, m2[, m3]`, gdzie n jest numerem wiersza, i identyfikatorem zmiennej całkowitej, zaś m albo stałymi, albo zmiennymi całkowitymi. Znaczeniem tej instrukcji jest wykonanie wszystkich wierszy aż do n włącznie dla wartości i od m_1 do m_2 z krokiem m_3 (domyślnie 1), a następnie przekazanie sterowania do wiersza następującego po wierszu `n`. Nieczytelność tej instrukcji wynika z tego, że koniec bloku iterowanych instrukcji nie jest oznaczony w żaden specjalny sposób i zauważenie, że właśnie w tym miejscu kończy się iteracja, wymaga przeczytania instrukcji `DO` na początku.

Jako przykład działającego programu w języku Fortran zamieszczę program znajdujący największą liczbę w tablicy, który można przeczytać w [IBM56, s.7]:

```
   BIGA = A(1)
   DO 20  I = 2, N
   IF (BIGA - A(I)) 10, 20, 20
10 BIGA = A(I)
20 CONTINUE
```

Występująca tu instrukcja `CONTINUE` oznacza pominięcie pozostałych instrukcji w pętli i wykonanie kolejnej iteracji.

2.2.3 Wnioski

Fortran jest bardzo ubogim językiem, ściśle podporządkowanym zadaniu, dla którego został opracowany, oraz komputerowi, na którym miały pracować kompilowane programy. Brak mu elegancji, którą cechują się późniejsze języki Algol i Lisp. Słabość tego języka objawia się w tym, że brak w nim jakiejkolwiek struktury - brak jest zarówno struktur sterujących znanych z Algolu, jak i strukturalnych typów danych, które można było znaleźć już w języku Plankalkül. Warty zauważenia jest fakt, że aż do Fortranu 66 język ten był ściśle związany z komputerem, dla którego został opracowany. Istniało kilka różnych wersji Fortranu przeznaczonych dla różnych komputerów i przeniesienie programu z jednego komputera do drugiego wiązało się z dodatkową pracą programisty.

Jednak mimo to Fortran jest wzorcowym przykładem „języka von Neumannowskiego" i zawiera wszystkie najważniejsze elementy, które wciąż są obecne we współczesnych językach tego rodzaju. Są nimi: instrukcje wykonywane sekwencyjnie, instrukcja przypisania, wyrażenia arytmetyczne, tablice o predefiniowanym rozmiarze, pętla *for*.

2.3 Algol - pierwszy praktyczny język strukturalny

2.3.1 Historia powstania

W latach 1956-1957 języki programowania przeżywały rozkwit. Nowe języki powstawały w bardzo dużej ilości; dla każdego komputera był tworzony conajmniej jeden odrębny język programowania. Jednak języki te były bardzo podobne do siebie (często wzorowane na Fortranie) i wkrótce zaczęto odczuwać potrzebę posiadania wspólnego języka, niezależnego od komputera i powszechnie używanego. Dnia 10 maja 1957 grupy użytkowników USE, SHARE i DUO wysunęły petycję o rozpoczęcie badań nad takim językiem. Petycję poparł ACM na spotkaniu Rady, które odbyło się w czerwcu 1957. W wyznaczonej na spotkaniu komisji znalazły się takie osoby, jak John Backus i John McCarthy.

Dużo wcześniej, w październiku 1955 roku, w Darmstadt odbyło się międzynarodowe sympozjum z automatycznego obliczania. Wielu z przemawiających kładło nacisk na konieczność opracowania „jednego uniwersalnego, niezależnego od maszyny języka algorytmicznego przeznaczonego dla wszystkich(...)"[12] Po spotkaniu w Darmstadt powołano *podkomisję GAMM ds. Języków Programowania*, która miała zająć się zaprojektowaniem odpowiedniego języka. Października 1957 roku członkowie podkomisji w liście do prezydenta ACM, Johna W. Carra, zasugerowali organizację konferencji przedstawicieli ACM i GAMM celem porozumienia się w kwestii opracowania wspólnego języka algebraicznego.

Obie grupy rozpoczęły bliską współpracę. Na spotkaniu w Filadelfii kwietnia 1958 roku wymieniły się dotychczasowymi wynikami. Ze względu na podobieństwo obu propozycji zdecydowano się zorganizować spotkanie w Zurychu, na którym czterech reprezentantów każdej z grup miało pracować nad unifikacją propozycji. Spotkanie trwało od 27 maja do 1 czerwca 1958. Ustalono na nim założenia, na których miał zostać oparty język IAL (*International Algebraic Language* - międzynarodowy język algebraiczny), który później przemianowano na Algol 58 (*Algorithmic Language* - język algorytmiczny).

Specyfikacja języka Algol 58 została opublikowana w postaci raportu w roku 1959. Specjalnie dla potrzeb Algolu została stworzona notacja opisu gramatyk bezkontekstowych BNF - Backus-Naur Form (czy też, jak niektórzy podają, Backus Normal Form). Notacja ta, rozbudowana do EBNF, jest powszechnie używana do dziś. Zaś sam język posiadał już wtedy wiele swoich charakterystycznych cech, w tym struktury sterujące, posiadał jednak wiele niedoróbek - np. sztuczne rozróżnienie pomiędzy procedurami i funkcjami.[13]

W dniach 11-16 stycznia 1960 roku odbyła się konferencja w Paryżu, na której ustalony został ostateczny kształt języka Algol 60. Jego opis został opublikowany w słynnym raporcie. Algol przyjął się zarówno jako język implementacji, jak i opisu algorytmów, doczekał się następców i naśladowców.

2.3.2 Funkcjonalność

Programy w języku Algol 60 są ustrukturalizowane w bloki. Każdy blok składa się z słowa kluczowego *begin*, ciągu deklaracji, ciągu instrukcji, słowa kluczowego *end*. Widać stąd, że deklaracje i instrukcje nie mogą być mieszane ze sobą.

Wśród deklaracji mogą znajdować się zmienne oraz procedury. Zadeklarowane na początku bloku byty mogą być używane jedynie wewnątrz tego bloku, w tym w innych blokach oraz procedurach zawartych wewnątrz niego. Zmienne mogą być jednego z trzech typów: całkowitego, zmiennoprzecinkowego lub boolowskiego, mogą być też tablicą jednego z tych typów. Rozmiar tablicy nie musi być znany w czasie kompilacji. Zmienne mogą być oznaczone słowem *own* - wtedy ich wartość będzie zachowywana pomiędzy kolejnymi wykonaniami bloku. Przypomina to zmienne statyczne z C++.

Deklaracja procedury składa się z (opcjonalnego) typu wartości zwracanej, nazwy procedury, listy parametrów formalnych, informacji o sposobie przekazywania parametrów oraz ich typach. Typy są zapisywane po liście parametrów formalnych, a nie wewnątrz niej - podobnie, jak we wczesnym C, ale inaczej niż w C++ i Pascalu. Parametry mogły być przekazywane na dwa sposoby: przez wartość i przez nazwę. Pierwszy mechanizm jest obecnie powszechnie używany, jednak drugi z nich już wyszedł z użycia i zasługuje na więcej uwagi. Jego semantyka jest taka, jakby „każdy parametr formalny zamienić w ciele procedury przez parametr faktyczny, zamykając go wcześniej w nawiasach, o ile jest to możliwe. Możliwe konflikty identyfikatorów (...) zostaną uniknięte(...)"[14] Można więc traktować je jak podstawienie na termach z przemianowaniem zmiennych w celu uniknięcia przechwyceń. Mechanizm wywołania przez nazwę pozwala na kilka zadziwiających rzeczy; jedną z nich jest tzw. *wynalazek Jensena*, który pozwala np. na zapisanie uniwersalnej procedury sumującej:

```
procedure sum (N,i,w,s);
```

12. [Nau78, s.16-17]
13. [Per78, s.7]
14. [alg60, s.28]

```
integer N, i;
real w, s;
begin
    s := 0;
    for i := 1 step 1 until N do
        s := s + w;
end
```

Powyższa procedura zwraca w zmiennej przekazanej jako argument s sumę wartości wyrażenia w dla wartości zmiennej i od 1 do N.

Procdury w Algolu mogą być zagnieżdżane w sobie, dozwolone zostały również wywołania rekurencyjne. Dzięki wynalazkowi rekurencji możliwe stało się później zwięzłe i czytelne zapisanie algorytmu Quicksort przez Hoare'a.[15]

Wśród instrukcji Algolu znajdują się: instrukcja przypisania, instrukcja warunkowa z jedną lub dwoma gałęziami, instrukcja pętli (mogła funkcjonować zarówno jak pętla *for* i *while*), instrukcja skoku, instrukcja wywołania procedury. Cały blok instrukcji również mógł być traktowany jako jedna instrukcja - jest to kluczowa cecha programowania strukturalnego.

2.3.3 Wnioski

Algol był kamieniem milowym w rozwoju języków programowania. Wprowadził do nich takie koncepcje, jak typ, parametry formalne i faktyczne, instrukcja złożona, rekurencja. W istocie, większość używanych współcześnie języków (m.in. C, Pascal, Java) jest do Algolu bardzo podobnych koncepcyjnie, a często i składniowo. Liczne dodatki, takie jak obiekty czy wyjątki nie zmieniły ogólnej postaci języka. Bardzo istotnym wkładem wniesionym przez twórców Algolu była notacja opisu gramatyk bezkontekstowych BNF.

Bardzo istotnym jest też fakt, że Algol był powszechnie używany do komunikacji między informatykami. Ukazujące się w publikacjach opisy algorytmów były bardzo często zapisywane właśnie w tym języku.

Ciekawostką jest zastosowanie call-by-name jako metody przekazywania parametrów funkcji. Podejście to bardzo przypomina znane z makrodefinicji podstawienie tekstowe. We współczesnych językach ta metoda już nie jest używana, w zastępstwie stosuje się referencje i typy funkcyjne.

Bardzo interesujące jest porównanie Algolu do języka Plankalkül Konrada Zuse. Algol zawiera znane z niego struktury sterujące (instrukcja warunkowa i pętla), procedury, zmienne lokalne. Hierarchiczne struktury danych zostały wprowadzone dopiero w Algolu 68.[16]

2.4 Lisp - program jest wyrażeniem

2.4.1 Historia powstania

John McCarthy, twórca języka Lisp, odczuł potrzebę istnienia algebraicznego języka przetwarzania list po wysłuchaniu latem 1956 roku wygłoszonego przez Newella, Shawa i Simona opisu języka IPL 2. Na projekt języka wpływ miały problemy z dziedziny sztucznej inteligencji, między innymi problem dowodzenia twierdzeń geometrycznych oraz dedukcji logicznej. W związku z naturą problemów zdecydowano się zrezygnować z popularnej notacji infiksowej i zapisywać wyrażenia w sposób prefiksowy.

Pierwsze kroki w kierunku sformułowania Lispu zostały uczynione przez modyfikowanie Fortranu. W wyniku prac Herberta Gelerntera i Carla Gerbericha z IBM powstał język *Fortran List Processing Language*, w skrócie FLPL. Zawierał on już funkcję *cons*, będącą konstruktorem list. Natomiast John McCarthy wynalazł wyrażenie warunkowe `XIF`, które początkowo zaimplementował jako dodatkową funkcję Fortranu. Jednak taka jego postać była kłopotliwa, gdyż wymuszała zwartościowanie obu gałęzi niezależnie od tego, która z nich była potrzebna. Problem ten doprowadził do odkrycia prawdziwego wyrażenia warunkowego.

15. [Hoa81, s.76]
16. [Ros72, s.593]

Problemy natury technicznej i politycznej związane z dalszym modyfikowaniem Fortranu zachęciły do pracy nad osobnym językiem. John McCarthy rozpoczął prace nad implementacją języka jesienią 1958 roku. Prace zaczęły się od opracowania tak zwanej M-notacji, przypominającej Fortran, a następnie ręcznego przepisywania programów zapisanych w tej notacji do kodu maszynowego w celu zdobycia doświadczenia potrzebnego do implementacji kompilatora. Wkrótce podjęto szereg ważnych decyzji projektowych - w szczególności wybrano *garbage collection* jako metodę zwalniania nieużywanej pamięci i uproszczono system typów.

W roku 1960 McCarthy napisał publikację o tytule „Recursive functions of symbolic expressions and their computation by machine, part I", w której przedstawił Lisp jako język programowania i jako formalizm teorii funkcji rekurencyjnych. Notacja opisana w tej pracy miała dwie istotne zalety. Po pierwsze, pozwalała na proste dowodzenie własności programów. Po drugie, pozwalała na napisanie „uniwersalnej funkcji Lispu" - *eval* - która mogła obliczyć dowolne wyrażenie Lispu. Utworzenie tej funkcji wymagało opracowania notacji reprezentującej funkcje Lispu w postaci danych Lispu. Notacja ta nieoczekiwanie przyjęła się i wyparła M-notację.

Wkrótce S.R. Russell zauważył, że *eval* może pełnić rolę interpretera Lispu, i zakodował go w postaci kodu maszynowego. To niespodziewane pojawienie się interpretera języka można uznać za początek funkcjonowania Lispu jako praktycznego języka programowania. Dziś istnieje wiele dialektów Lispu (w tym Scheme i Common Lisp), a wiele osób nadal uważa go za najlepszy język, jaki kiedykolwiek powstał.

2.4.2 Funkcjonalność

John McCarthy napisał, że Lisp najlepiej scharakteryzować za pomocą następujących idei: „obliczanie za pomocą wyrażeń symbolicznych, a nie liczb, reprezentacja wyrażeń symbolicznych i innych informacji w pamięci komputera za pomocą struktury listy, (...), niewielki zbiór konstruktorów i selektorów wyrażonych za pomocą funkcji, składanie funkcji jako narzędzie tworzenia bardziej złożonych funkcji, użycie wyrażeń warunkowych dla wprowadzenia rozgałęzień do definicji funkcji, rekurencyjne użycie wyrażeń warunkowych jako dostateczne narzędzie do tworzenia funkcji obliczalnych, użycie wyrażeń lambda do nazywania funkcji, reprezentowanie programów w Lispie jako dane Lispu, (...), funkcja Lispu *eval* która funkcjonuje zarówno jako definicja języka, jak i interpreter, garbage collection jako metoda zwalniania pamięci. (...) Niektóre z tych pomysłów zostały wzięte z innych języków, jednak większość była nowa. Pod koniec okresu początkowego stało się jasne, że ta kombinacja pomysłów tworzyła elegancki system matematyczny jak i praktyczny język programowania."[17]

Zacznijmy od podstaw, czyli od omówienia notacji Lispu. Podstawową strukturą danych jest lista zbudowana z par uporządkowanych. Pary zapisuje się notacją `(w1 . w2)`, natomiast dla list stosuje się skrócony zapis `(w1 w2 ... wn)`. Elementami list mogą być inne listy bądź symbole. Dla przykładu, poniższe dwa wyrażenia oznaczają listę liczb od 1 do 5.[18]

```
(1 . (2 . (3 . (4 . (5 . ()))))))
(1 2 3 4 5)
```

Omówię teraz postać wyrażeń w języku Lisp. Wyrażenie może być albo symbolem, który stanowi w tym kontekście odwołanie do wartości nazwanej zmiennej, albo listą, której pierwszym symbolem jest operator, a pozostałymi - argumenty. Mamy więc do czynienia z notacją prefiksową. Żeby dać przykład, wyrażenie $2+3*4-\sin 3$ można zapisać w Lispie następująco:

```
(+ 2 (* 3 4) (- (sin 3)))
```

Można teraz przejść do omówienia najważniejszych funkcji wbudowanych w Lisp. Dwuargumentowa funkcja *cons* jest konstruktorem pary uporządkowanej. Możliwe jest również użycie zamiast niej cytowania: apostrof umieszczony przed wyrażeniem oznacza, że nie jest ono instrukcją do wykonania, a wyrażeniem do zwrócenia. Poniższe wyrażenia konstruują takie same listy:

```
(cons 1 (cons (cons 2 (cons 3 '())) '()))
(list 1 (list 2 3))
'(1 (2 3))
```

17. [McC79, s.1-2]

18. W pierwotnym Lispie nie było liczb, jednak zostały one szybko wprowadzone.

Funkcja *car* wydobywa pierwszy element pary uporządkowanej, natomiast *cdr* - drugi.[19]

Istotnym wynalazkiem Lispu jest wyrażenie warunkowe. Ma ono postać `(cond (p1 e1) ... (pn en))`, gdzie p_i są wyrażeniami warunkowymi, a e_i odpowiadającymi im wyrażeniami. Instrukcja ta sprawdza po kolei warunki p_i i zwraca wartość tego wyrażenia e_i, któremu odpowiada pierwszy prawdziwy warunek. Instrukcja ta nie może być symulowana przez funkcję - przy wywołaniu funkcji wartościowane są wszystkie jej parametry, co tutaj nie może mieć miejsca.

W końcu, Lisp określa metodę na definiowanie funkcji w dowolnym miejscu. Służy temu konstrukcja `(lambda (p1 ... pn) e)`, gdzie p_i są parametrami formalnymi, natomiast e ciałem funkcji. Funkcje zdefiniowane w ten sposób mogą być używane jako operatory. Do przypisania nazwy definiowanej funkcji (lub zmiennej) można użyć konstrukcji `(define n e)`.

Aby zaprezentować użycie powyższej notacji, podaję program przykładowy: poniższa funkcja pobiera jako argumenty dwie listy i zwraca listę dwuelementowych list zawierających odpowiadające sobie elementy z argumentów.[20]

```
(define pair (lambda (x y)
    (cond ((and (null? x) (null? y)) '())
          ((and (pair? x) (pair? y))
           (cons (list (car x) (car y))
                 (pair (cdr x) (cdr y)))))))
```

Działanie powyższej funkcji można opisać następująco. Jeśli obydwa argumenty są pustymi listami, wynikiem jest lista pusta. Jeśli obydwa argumenty są parami, doklejamy dwuelementową listę złożoną z pierwszych elementów argumentów do wyniku działania funkcji na ogonach parametrów.

2.4.3 Wnioski

Lisp jest eleganckim językiem. Zrywa on częściowo z sekwencyjnym, von Neumannowskim modelem obliczeń oraz z rozróżnieniem pomiędzy wyrażeniami a instrukcjami. Kładzie też nacisk na przetwarzanie nie liczb, a wyrażeń, symboli. Siła wyrazu uzyskana dzięki specyficznej konstrukcji języka jest tak duża, że możliwe jest bardzo proste zaimplemetowanie interpretera Lispu w Lispie. Dla Paula Grahama jest to wręcz „definiująca własność Lispu."[21]

Wartym zauważenia jest też fakt, że w Lispie stają się możliwe do zrealizowania wysokopoziomowe konstrukcje języka Plankalkül Konrada Zuse. Hierarchiczne struktury danych, kwantyfikatory logiczne i operacje na listach o dynamicznym rozmiarze łatwo tłumaczą się na konstrukcje Lispu.

W kierunku wskazanym przez Lisp podążyło wiele nowych języków. Do najbardziej interesujących można z pewnością zaliczyć język Haskell oraz rodzinę języków ML z najważniejszymi przedstawicielami: OCaml i SML. Zachowują one ogólną ideę Lispu, ale dodają do niej leksykalne zakresy nazw, silne typowanie oraz typy algebraiczne. Szczególnie Haskell jest wart uwagi ze względu na jego elegancję. Warto też zwrócić uwagę na znany w edukacji język Logo, który jest w dużej części wzorowany właśnie na Lispie. Logo jest znany przede wszystkim z rysującego linie żółwia, ale jest to obraz błędny; Logo jest przede wszystkim bardzo silnym językiem do przetwarzania słów i danych listowych.

Badacze są zgodni, że powinno się zerwać z sekwencyjnym, mającym niewielką siłę wyrazu i nieposiadającym dobrych matematycznych własności von Neumannowskim modelem obliczeń i dążyć w kierunku wyznaczonym przez język Lisp.[22] Niestety, mimo trwających prac w tej dziedzinie[23], wciąż w użytku produkcyjnym przeważają języki będące następcami Fortranu i Algolu. Ciekawym wyjątkiem jest język Erlang.[24] Interesującym jest fakt, że jeden z wynalazków Lispu - garbage collection - dostał się już do popularnych języków programowania i zaczyna być doceniany.

19. Nazwy tych dwóch funkcji są związane z komputerem IBM 709, oznaczają odpowiednio *copy address register* i *copy decrement register*.

20. Wzięte, z drobnymi modyfikacjami, z [Gra02, s.7]

21. [Gra02, s.1]

22. [Gra02, s.1], [Bac78b, s.178], [Bac78a]

23. Dość nowym językiem próbującym połączyć ze sobą oba światy w praktyczną całość jest język Nemerle rozwijany na Uniwersytecie Wrocławskim. http://www.nemerle.org/

24. Erlang jest językiem opracowanym przez firmę Ericsson, który kładzie nacisk na programowanie równoległe i rozproszone, przy czym wiele jego cech wywodzi się z Lispu. http://www.erlang.org/

3 Podsumowanie

Nasuwające się wnioski są niestety dość przykre. Języki będące obecnie w powszechnym użyciu - C++, C#, Java, PHP i podobne - są w istocie, pomimo niewielkich różnic, jedynie kalką tego, co istniało już w latach 60-tych. Można za to powiedzieć o nich, że stały się niepotrzebnie skomplikowane - lata poprawiania i dokładania nowej funkcjonalności sprawiły, że specyfikacje tych języków są koszmarnie złożone, co utrudnia życie zarówno autorom kompilatorów (wystarczy powiedzieć, że na dzień dzisiejszy nie istnieje kompilator C++ w pełni zgodny ze standardem ANSI!), jak i programistom, którzy muszą pamiętać mnóstwo detali dotyczących języka. Wydawałoby się, że przez te lata powinien nastąpić znaczący postęp - jednak to się nie stało. *Nic nowego pod słońcem...*

Pozostaje mieć nadzieję, że programiści zaczną powoli zauważać zalety nowych paradygmatów programowania i odchodzić stopniowo od imperatywnego, von Neumannowskiego sposobu myślenia o programach. Kiedy na sekwencyjne, czysto imperatywne programy będzie się patrzeć tak, jak dzisiaj na upstrzone *goto* programy z pierwszych wersji Fortranu, będzie można uznać, że dokonał się prawdziwy postęp.

Bibliografia

[alg60] *Report on the algorithmic language Algol 60*, 1960.

[Bac78a] John Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, Aug 1978.

[Bac78b] John Backus. The History of Fortran I, II and III. *ACM SIGPLAN Notices*, 13(8):165–180, Aug 1978.

[Cle03] Loek G.W.A. Cleophas. Plankalkül. An overview of the programming language in historical perspective, May 2003.

[FB72] H. Wössner F.L. Bauer. The "Plankalkül" of Konrad Zuse: A Forerunner of Today's Programming Languages. *Communications of the ACM*, 15(7):678–685, Jul 1972.

[Gra02] Paul Graham. The Roots of Lisp, Jan 2002.

[Hoa81] Charles A.R. Hoare. The Emperor's Old Clothes. *Communications of the ACM*, 24(2):75–83, Feb 1981.

[IBM56] IBM Corporation. *The Fortran automatic coding system for the IBM 704 EDPM*, Oct 1956.

[IBM63] IBM Corporation. *Fortran II General Information Manual*, Dec 1963.

[McC79] John McCarthy. History of Lisp, Feb 1979.

[Nau78] Peter Naur. The European side of the last phase of the development of Algol 60. *ACM SIGPLAN Notices*, 13(8):15–44, Aug 1978.

[Per78] Alan J. Perlis. The American side of the development of Algol. *ACM SIGPLAN Notices*, 13(8):3–14, Aug 1978.

[Ros72] Saul Rosen. Programming Systems and Languages 1965-1975. *Communications of the ACM*, 15(7):591–600, Jul 1972.

[Sam72] Jean E. Sammet. Programming Languages: History and Future. *Communications of the ACM*, 15(7):601–610, Jul 1972.

[Zus] Horst Zuse. The Life and Work of Konrad Zuse. http://www.epemag.com/zuse/.

[Zus45] Konrad Zuse. *Der Plankalkül*. 1945. http://www.zib.de/zuse/.