Plankalkül,
czyli pierwszy
wysokopoziomowy język programowania

(Plankalkül,
the first attempt to describe
a high-level programming language)

Jakub Marcinkowski

Praca licencjacka

Promotor: Piotr Polesiuk

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

26 sierpnia 2024

Streszczenie

Niemal 80 lat temu powstał pierwszy opis wysokopoziomowego języka programowania, Plankalküla, napisany przez niemieckiego konstruktora Konrada Zuse. Niniejsza praca ma na celu zrozumieć tę mało znaną ideę oraz jak najpełniej ją opisać. Poza tym wciela ona w życie fragment tej idei poprzez interpreter napisany w Pythonie.

Almost 80 years ago appeared the first description of a high-level programming language, Plankalkül. It was written by the German inventor Konrad Zuse. The goal of this thesis is to understand this not very well known idea and to describe it in detail. Apart from that it brings a part of this idea to life through an interpreter implemented in Python.

Spis treści

Rozdział 1.

Wprowadzenie

Dawno temu, w dzikim kraju rasizmu, mordu i niewolnictwa, żył pionier. Pionier, który wyprzedzał myślą informatyczną resztę świata o dekady. Nie będzie to jednak postać z książek Karola Maya.

Mowa tu o Niemcu, Konradzie Zuse, budowniczym mechanicznych kalkulatorów i komputerów w hitlerowskich i podrugowojennych Niemczech. Jego dziełem były V1 i V3, programowalne kalkulatory, o jednostkach obliczeniowych zadziwiająco przypominających dużo późniejsze procesory, oraz V4 ukończony na początku 1945 roku komputer wybudowany na zamówienie rządu niemieckiego [1]. Aby nazwy nie myliły się z rakietami bombardującymi Londyn, Zuse zmienił nazwy na Z1, Z2, Z3, Z4, itd.*. W międzyczasie Zuse pomagał stworzyć Hs-293, sterowaną przez radio bombę lotniczą[‡], oraz zbudował S1 i S2, maszyny do zdalnego sterowania Hs-293 w nieoptymalnych warunkach pogodowych (zawierały one nawet DAC) [§].

Gdy Zuse budował swoje maszyny (a zwłaszcza, gdy je programował), zrozumiał, że zarówno on, jak i reszta użytkowników maszyn takich jak jego będzie potrzebowała lepszego języka do wyrażania programów komputerowych, niż dziurki na taśmie. Stąd w jego głowie narodził się pomysł języka Plankalkül (plan obliczeń). Języka jednocześnie współczesnego jak i niedzisiejszego, momentami proszącego się o określenie „dziki", a czasem „genialny".

Czytając o tym języku warto pamiętać, że Konrad Zuse nigdy wcześniej nie widział języka programowania. Jego jedynym doświadczeniem z pisaniem programów było wybijanie dziurek na taśmie filmowej, które służyły do programowania jego kalkulatorów V1 i V3. Co więcej, Zuse żył w całkowicie odciętych od świata (zresztą słusznie) Niemczech, co sprawiało, że nie wiedział o rozwoju komputerów takich jak ENIAC czy Manchester Mark 1. Można więc powiedzieć, że Zuse był samoukiem, za-

*O ironio, po ewakuacji z Berlina w lutym 1945, Zuse i jego zespół trafili do jednej z pełnych pracowników przymusowych fabryk Wernhera Von Brauna w regionie gór Harz[†]

[†]Zuse; „The Computer - My Life", s.92 [2]

[‡]Zuse; „The Computer - My Life", s.60 [2]

[§]Zuse; „The Computer - My Life", s.71 [2]

równo w dziedzinie budowy komputerów, jak i w dziedzinie języków programowania.

Myśląc o innych wysokopoziomowych językach programowania myślimy o językach powstałych dzięki swoistej ewolucji, ich ostateczny kształt wynikał z doświadczeń, prób, potrzeb i możliwości. Najpierw z gęstej zupy wyłaniały się jednokomórkowe asemblery, z których powstawały różne morskie potworki lat pięćdziesiątych i sześćdziesiątych. Potem na brzeg wyszedł *C* i stał się przodkiem dla wszystkich innych zwierząt, dla wielbłąda *OCamla*, okapi *Haskella*, pawiana *Javy*, czy szympansa *Pythona*.

Plankalkül jest czymś zupełnie innym. Jeśli stwierdzamy, że inne języki powstały na bazie ewolucji, ten można określić jako dzieło Boskie. Stworzony z wyobrażenia, nie doświadczeń. Z tego wynikają jego niezwykłości i dziwy. I o tym właśnie języku będzie niniejsza praca.

Rozdział 2.

Der Plankalkül

W miarę oczywistym obecnie jest, że pisząc program nie chcemy używać asemblera, lecz porządnego, elegancko wyglądającego, czytelnego języka programowania. Lecz nie od początku była to potrzeba oczywista, w późnych latach czterdziestych istniało w Europie kilka komputerów, każdy o innej architekturze i każdy o niezbyt długim oczekiwanym czasie życia (bo miały je zastąpić nowe, lepsze). Dlatego wygodne języki programowania pojawiały się nieco później; *FORTRAN* w 1954, *Lisp* w 1958, *Cobol* w 1959, czy żyjący z nami do dziś *C* w 1972.

Zuse wyprzedził innych o niemal dekadę, jego projekt wysokopoziomowego języka programowania *Plankalkül* został opublikowany w 1945. Nie miał jednak większych szans na realizację, był jak na swój czas zbyt skomplikowany. W tym rozdziale nieco mu się przyjrzymy*.

2.1. Fasada, czyli składnia i rodzaje zmiennych

Wyobraźmy sobie kościół. Wyobraźmy sobie wielką gotycką budowlę mającą swoim majestatem przytłaczać maluczkich. Idąc zwiedzić taki kościół zaczynamy od przyjrzenia się zewnętrzu, obchodzimy go dookoła i oglądamy fasadę. Tak też zrobimy „zwiedzając" Plankalküla, najpierw obejrzymy jego fasadę. Słowo „fasada" jest tu zupełnie trafne, jako, że Plankalkül używał niezwykłej dwuwymiarowej składni. Na przykład, chcąc odwołać się do zmiennej V_1 o typie 8 (co to za typ dowiemy się za moment), tak jak poniżej.

	V
V	1
K	
S	8

*Na bazie pracy Zusego zredagowanej przez Raula Rojasa [3]

Gdy zaś chcemy dodać do siebie dwie liczby Z_0 i Z_1, i przypisać wartość do zmiennej Z_2 napiszemy tak, jak niżej.

	Z	$+$	Z	$\Rightarrow$	Z
V	0		1		2
K					
S	8		8		8

Po pierwszym rzucie oka naturalnym pytaniem jest „Czym są V i Z?". Te litery nie są dowolnymi nazwami. Konrad Zuse wymyślił, że w Plankalkülu powinny być trzy rodzaje zmiennych.

- Takie, które można wyłącznie odczytywać, które nazywały się V_i (i to indeks zmiennej, Zuse głęboko czerpał z doświadczeń matematycznych), których nazwa bierze się od „Variablen", „zmienne".
- Zmienne tylko do zapisu R_i, od „Resultatwerte", „wartości wynikowe".
- Zmienne zarówno do odczytu i zapisu Z_i, od „Zwischenwerte", „wartości pośrednie".

2.2. Kamień czy glina, czyli typy danych i struktury

Obchodząc kościół możemy się zacząć zastanawiać nad tym, z czego on został zbudowany. Są kościółki drewniane, z muru pruskiego, ceglane, z kamieni. Z czego zaś jest zbudowany Plankalkül? Aby poznać odpowiedź rzućmy okiem na Plankalkülowe typy danych. W tym języku było ich dużo, dlatego teraz obejrzymy dwa z nich, a o pozostałych można przeczytać w rozdziale 2.7.

1. S0 — czyli typu boolowskiego, Zuse nazywał go Ja-Nein-Wert (Wartość tak-nie). Jego wartościami są $+$ (Prawda) i $-$ (Fałsz);
2. A8 — czyli liczby bez dokładnej specyfikacji (Nie jest jasne, czym się różni A od S, po prostu typy danych o numerach od 8 wzwyż mają A, a te niższe S).

Oprócz danych typów danych można było korzystać ze struktur do budowania typów bardziej skomplikowanych.

1. Listy — lista składająca się z n takich samych elementów. Składnia wygląda następująco: $n \times \sigma$ lub $n.\sigma$ (Zuse nie był w tej kwestii zupełnie konsekwentny) gdzie σ to typ elementu listy.
2. Krotki — krotka składająca się z dowolnych typów (niekoniecznie różnych). Składnia: (σ, τ, γ).

Na przykład typ określający listę złożoną z dziesięciu par liczb i wartości boolowskich zapiszemy $10 \times (A8, S0)$. Co więc zrobimy, jeżeli do zmiennej Z_1 chcemy dodać trzeci element listy Z_0 o typie $n \times 8$?
Do określania „komponentu" zmiennej służy linijka oznaczona K (W momencie gdy linijka K nie jest wykorzystywana, można jej nie zapisywać).

	Z	$+$	Z	$\Rightarrow$	Z
V	0		1		2
K	2				
S	8		8		8

Jak widać, w linijce S zapisujemy typ komponentu do którego się odnosimy, nie zaś typ całej zmiennej.

2.3. Fundament i katakumby, czyli słów kilka o strukturze programu

Skoro już obejrzeliśmy fasadę, oraz dowiedzieliśmy się, z czego kościół jest zbudowany, zejdźmy do podziemia i zobaczmy, na jakiej podstawie się wznosi, czyli co jest podstawą struktury programu w Plankalkülu.

W współczesnych językach programowania takim fundamentem są funkcje, które pozwalają budować ustrukturyzowany program. W Plankalkülu istniało coś do złudzenia przypominającego współczesne funkcje. Zuse nazwał tę strukturę Planami.

2.3.1. Zakres indeksów w planie

Indeksy zmiennych zawsze odnoszą się do pojedynczego planu obliczeń, w którym były zadeklarowane, są więc zmiennymi lokalnymi. Na przykład zmienna Z_3 może pojawiać się w wielu planach, i w każdym z nich mieć inną wartość czy typ. Jest to dość nowoczesne podejście.

2.3.2. Składnia planów

Pierwszą instrukcją planu obliczeń jest instrukcja deklarująca typy przyjmowanych i zwracanych zmiennych.

	$R(V,$	$V)$	$\Rightarrow$	$(R,$	$R)$
V	0	1		0	1
S	0	0		0	0

Plan zawsze przyjmuje zmienne V, które w planie można tylko odczytywać, i zwraca zmienne R, do których można tylko pisać. W powyższym przykładzie plan przyjmuje dwie zmienne V_0 i V_1 zaś zwraca zmienne R_0 i R_1.

2.3.3. Wywołanie podplanu

W planie obliczeniowym możemy odwołać się do planu, pisząc jak w poniższym przykładzie.

	$R9.10(Z)$	$\Rightarrow$	Z
V	0 0		1
S	0		0

Instrukcja wyżej pierwszą wartość wynikową planu o numerze identyfikacyjnym 9.10 wywołanego z argumentem Z_0 przypisuje do zmiennej Z_1.

Możemy też wyciągnąć obie wartości z takiego wywołania.

	$R9.10(Z)$	$\Rightarrow$	$(Z,$	$Z)$
V	0		1	2
S	0		0	0

No dobrze, ale w deklaracji podplanu nie występuje nigdzie identyfikator, do którego odwołujemy się podczas wywołania, skąd więc wiemy, do czego sie odwołać? Trzeba pamiętać, że Zuse pisał swoje plany wyłącznie na papierze, i nad nimi było zapisane np. P9.10, co je identyfikowało. W dodatku zdarzały się plany, które miały normalne nazwy, na przykład plan „Maj", który zwracał maksimum z dwóch liczb.

2.3.4. Plany prawie sztywne

Zmienny operator

Wyobraźmy sobie kilka planów, które różnią się jedynie niektórymi operatorami. Pierwszy będzie zwracał koniunkcję argumentów,

	$R(V,$	$V)$	$\Rightarrow$	R
V	0	1		0
S	0	0		0
	V	$\wedge$	V	$\Rightarrow$ R
V	0		1	0

Drugi alternatywę argumentów.

	$R(V,$	$V)$	$\Rightarrow$	R
V	0	1		0
S	0	0		0
	V	$\vee$	V	$\Rightarrow$ R
V	0		1	0

A trzeci implikację między argumentami.

	$R(V,$	$V)$	$\Rightarrow$	R
V	0	1		0
S	0	0		0
	V	$\rightarrow$	V	$\Rightarrow$ R
V	0		1	0

Wszystkie te plany moglibyśmy zapisać jednym planem.

	$R(V,$	$V)$	$\Rightarrow$	R
V	0	1		0
S	0	0		0
	V	Φ	V	$\Rightarrow$ R
V	0		1	0

Jednak, musielibyśmy jeszcze jakoś dostarczać Φ. Temu służą plany prawie sztywne, czyli takie, w których taki operator (albo wiele takich operatorów, tylko wówczas musimy je indeksować, oczywiście od 0 wzwyż) Φ dostarczamy jako argument:

	$R(\Phi,$	$V,$	$V)$	$\Rightarrow$	R
V		0	1		0
S		0	0		0
	V	Φ	V	$\Rightarrow$	R
V	0		1		0

Nie jest do końca jasne, jak Zuse wyobrażał sobie wywoływanie takich planów w praktyce. W swojej pracy nadawał on takim planom osobne identyfikatory dla różnych operatorów na poziomie metajęzyka, i to takie podplany wołał (co może przypominać makra z współczesnych języków). To oczywiście trochę ogranicza siłę wyrazu którą można było osiągnąć operatorem Φ i być może, gdyby implementował Plankalküla w praktyce, stworzyłby bardziej ogólną składnię.

2.4. Kolumny i łuki, czyli co jeszcze podpiera nasz program

Ale sam fundament nie wystarczy, aby zbudować wielki gotycki kościół. Potrzeba więcej, by móc zbudować taką wielką budowlę. Dlatego kierujemy się ku wyjściu

z piwnic, idziemy obejrzeć wspaniałe gotyckie kolumny, podpory i łuki, trzymające kościół w górze.

2.4.1. Kolumny, czyli wyrażenia warunkowe

Jednym z ważnych elementów podtrzymujących kościół są wielkie kolumny. Tak samo do budowy języka programowania przydają się wyrażenia warunkowe. Co ciekawe, Zuse w latach czterdziestych o tym wiedział, dlatego je do języka dodał. W Plankalkülu konstrukcja wyrażeń warunkowych była reprezentowana znakiem $\dot{\rightarrow}$. Na lewo od $\dot{\rightarrow}$ znajdował się warunek, na prawo zaś kod do wykonania, jeżeli warunek był spełniony. Na przykład, napisana poniżej przez Zusego funkcja „Maj", o której niedawno wspomnieliśmy.

	Maj(V,	V)	$\Rightarrow$	R
V		0	1	0
S		8	8	8

	$V \geq V$	$\dot{\rightarrow}$	$(V \Rightarrow$	$R)$
V	0	1	0	0
K				
S	8	8	8	8

	$\overline{V \geq V}$	$\dot{\rightarrow}$	$(V \Rightarrow$	$R)$
V	0	1	1	0
K				
S	8	8	8	8

W pierwszej instrukcji mamy deklaracje funkcji, która przyjmuje dwa argumenty i zwraca jedną liczbę. W drugiej instrukcji sprawdzamy, czy V_0 jest większe lub równe V_1, i jeżeli tak, przypisujemy V_0 do zmiennej wynikowej R_0. W instrukcji trzeciej sprawdzamy, czy warunek z wiersza drugiego ewaluuje się do Fałszu, i jeśli tak, do zmiennej wynikowej R_0 przypisujemy V_1. Warto zauważyć, że w Plankalkülu nie było wyrażenia „w przeciwnym przypadku", chcąc go wyrazić, trzeba było po prostu ten przypadek uwzględnić.

2.4.2. Łuki, czyli pętle

Nad kolumnami wyrażeń warunkowych wznoszą się w Plankalkülu łuki pętli. Pętla w Plankalkülu może budzić pewne skojarzenia z współczesnymi realizacjami pętli *while*. Współczesna pętla while może mieć następującą składnię.

```
while warunek:
  kod do wykonania
```

Jednak w Plankalkülu wygląda to nieco inaczej.

$$W \left[\begin{array}{ccc} F_0 & \dot{\rightarrow} & P_0 \\ F_1 & \dot{\rightarrow} & P_1 \\ \vdots & & \\ F_n & \dot{\rightarrow} & P_n \end{array} \right]$$

W pochodzi od *Wiederholungspläne*, „plan powtarzalny". Taka pętla W wykonuje się dopóki przynajmniej jeden z warunków F_1, F_2, ..., F_n ewaluuje się do Prawdy. W Plankalkülu powinien się tam jeszcze pojawić warunek (choć Zuse stwierdził, że „w sumie nie ma potrzeby go dawać")

$$\overline{F_1} \wedge \overline{F_2} \wedge \overline{F_3} \wedge ..\overline{F_n} \Rightarrow Fin^2$$

gdzie Fin jest operatorem wychodzenia z bloku kodu (jak *break*), zaś „^{2}" mówi jak wiele bloków kodu chcemy opuścić (można więc używać Fin, Fin^2, Fin^3, ...). Czemu w warunku wyjścia używamy więc Fin^2 a nie Fin? Ponieważ przy pomocy Fin wyszlibyśmy tylko ponad warunek wyjścia, nie zaś z samej pętli. Warto też zwrócić uwagę, że w warunku wyjścia pojawia się znak $\Rightarrow$, a nie $\dot{\rightarrow}$. Jest to niezwykłe*, ponieważ jest to wyrażenie warunkowe, korzystające z operatora przypisania, który w tym przypadku zachowuje się tak jakby był operatorem $\dot{\rightarrow}$.

Wróćmy jednak do zwiedzania. Gdy podnosimy głowę, by dokładniej przyjrzeć się łukom w tym gotyckim kościele, ze zdumieniem zauważamy, że coś jest bardzo nie tak! Nie dość, że każdy łuk jest inny, w dodatku wszystkie są pomalowane złotą farbą! Uświadamiamy sobie straszną prawdę, nie jesteśmy w kościele gotyckim, to kościół przerobiony na barok. Przyjrzyjmy się więc pierwszym oznakom baroku, jakie zauważyliśmy. Oprócz pętli W Zuse opisał jeszcze pętle iterujące się określoną liczbę razy, wszystkie one są jednak tak naprawdę lukrem syntaktycznym nad pętlą W opisaną wyżej.

Pętla $W0$.

$$W0(n)[P()]$$

Pierwszą z takich pętli jest pętla $W0$, która powtarza ciało $P()$ dokładnie n razy. Po odlukrzeniu równoważny tej pętli kod w Plankalkülu wygląda jak poniżej.

$$0 \Rightarrow \varepsilon \mid W \ [\, \varepsilon < n \dot{\rightarrow} [P \mid \varepsilon + 1 \Rightarrow \varepsilon \,]]$$

W powyższym kodzie ε jest niejawny, nie można go użyć w kawałku programu oznaczonym przez P, zaś symbol | jest separatorem.

*Choć w swojej pracy miał również przykłady, gdy Fin występował po $\dot{\rightarrow}$ jak również jako osobna operacja

Pętla $W1$

$$W1(n)[P(i)]$$

Ta pętla tak samo powtarza ciało $P(i)$ dokładnie n razy, przy czym w środku mamy możliwość odwołania się do iteratora i rosnącego od 0 do $n-1$. Można to równoważnie zapisać w Plankalkülu jak niżej.

$$0 \Rightarrow i \mid W \ [\, i < n \mathrel{\dot{\rightarrow}} [P(i) \mid i+1 \Rightarrow i \,]]$$

Pętla $W2$

$W2$ od pętli $W1$ różni się wyłącznie tym, że iterator maleje a nie rośnie.

$$W2(n)[P(i)]$$

Również odcukrzony równoważny kod w Plankalkülu jest podobny.

$$0 \Rightarrow i \mid W \ [\, i > 0 \mathrel{\dot{\rightarrow}} [P(i) \mid i-1 \Rightarrow i \,]]$$

Pętla $W3$

Kolejna pętla $W3$ przyjmuje dwa argumenty n i m.

$$W3(n,m)[P(i)]$$

Warunkiem jej użycia jest to, żeby n było mniejsze lub równe m.

$$n \leq m$$

W ciele pętli możemy korzystać z iteratora i który w tym przypadku rośnie od n do $m-1$.

$$n \Rightarrow i \mid W \ [\, i < m \mathrel{\dot{\rightarrow}} [P(i) \mid i+1 \Rightarrow i \,]]$$

Pętla $W4$

Pętla $W4$ jest podobna do pętli $W3$.

$$W4(n,m)[P(i)]$$

Warunkiem użycia tej pętli jest jednak to, aby n było większe lub równe m.

$$n \geq m$$

W pętli $W4$ iterator maleje od n do $m+1$.

$$n \Rightarrow i \mid W \ [\, i > m \mathrel{\dot{\rightarrow}} [P(i) \mid i-1 \Rightarrow i \,]]$$

Pętla $W5$

Pętla $W5$ również przyjmuje dwa argumenty n i m.

$$W5(n,m)[P(i)]$$

Ta pętla łączy w sobie możliwości pętli $W3$ i pętli $W4$. Iterator dąży od n do $m \pm 1$, w zależności, czy m jest mniejsze czy większe od n.

$$n \Rightarrow i \mid W \left[i \neq m \mathrel{\dot{\to}} \begin{bmatrix} P(i) \\ m > n \mathrel{\dot{\to}} (i+1) \Rightarrow i \\ m < n \mathrel{\dot{\to}} (i-1) \Rightarrow i \end{bmatrix} \right]$$

Pętla $W6$

Istnieje też pętla $W6$. Bierze ona pierwszy element z listy, dopóki nie jest ona pusta, i w jakiś sposób pozwala go wykorzystać w kodzie w środku pętli, zaś do następnej iteracji przekazując tylko ogon listy, przetworzony w niewystarczająco opisany dla zrozumimenia przeze mnie sposób.

2.4.3. Zmienne jako indeksy

Naturalnym po rozdziale o pętlach jest pytanie jak iterować się po liście? Możliwym jest indeksowanie w zmiennych (w linijce K) za pomocą innej zmiennej. Jak to zrealizować?

Pierwszą możliwością jest wpisanie w linijce K indeksu i.

	V
V	1
K	i
S	8

Problem pojawia się, gdy jesteśmy w głębszym niż 1 zagnieżdżeniu pętli, wówczas do i trzeba dodać jego indeks. Jak wiemy, musimy to robić w reprezentacji wierszowej, czyli nie możemy wpisać i w linijkę K. Ale jest rozwiązanie.

	V	Z
V	0	1
K		
A	$1.n$	9

Widzimy tu zmienną V_0, która jest ciągiem bitów, oraz zmienną Z_1, będącą liczbą całkowitą dodatnią, która stanowi indeks w zmiennej V_0. Warto przypomnieć, że w linijce S^* powinniśmy wówczas wpisać typ indeksowanego komponentu, nie całej zmiennej.

*W powyższym przykładzie linijka S nazywa się A. Zuse w przypadkach niektórych typów używał do jej nazwania litery A zamiast S

2.4.4. Wielokrotnie zagnieżdżone pętle

W przypadku, gdy zagnieżdżamy pętle wielokrotnie, nadajemy im indeksy. Na przykład w tym kodzie sprawdzającym, czy istnieją dwa elementy listy mające tę samą wartość

$$\begin{array}{c|lll} & R(Z) & \Rightarrow & R \\ V & 0 & & 0 \\ S & n \times \sigma & & 0 \end{array}$$

$$\begin{array}{c|lll} & + & \Rightarrow & Z \\ V & & & 1 \\ S & & & 0 \end{array}$$

$$\begin{array}{c|llll} & W1(n) & \left[W3(i+1,n)\right. & \left[Z\right. & \left[i \neq Z\right. & \left[i \wedge Z \Rightarrow Z\right.\right.\right.\right] \\ V & 0 & 1 \quad 0 & 0 \quad 0 & 0 \quad 0 & 1 \quad 1 \quad 1 \\ K & & & & & \\ S & & & \sigma \quad\quad \sigma & & 0 \quad 0 \end{array}$$

pętle są indeksowane od 0, zaś odpowiadające im i mają te same co one indeksy. Jak widać, w tym kodzie wykorzystujemy dwa rodzaje pętli, ta bardziej zewnętrzna przechodzi po wszystkich elementach listy, ta wewnętrzna po elementach o wyższym indeksie niż obecnie wskazywany przez zewnętrzną. Kod wewnątrz pętli wsprawdza czy dwa elementy listy mają różne wartości, i jeśli tak, zapisuje do Z_0 koniunkcję wcześniejszego Z_0 i prawdy. Jeśli elementy listy mają te same wartości, zapisuje do Z_0 koniunkcję wcześniejszego Z_0 i fałszu czyli fałsz, który już tam zostanie do końca wykonania pętli.

2.5. Więcej złota

Przyglądając się wnętrzu kościoła możemy dojść do wniosku, że mamy do czynienia z barokiem niemieckim, nie polskim. Czym różni się barok niemiecki od polskiego? Tym, że polski jest barokiem wysublimowanym, w którym kolory z sobą współgrają, ozdoby nie przytłaczają i widać w nim plan i artyzm*. Barok niemiecki z kolei polega na pomalowaniu wszystkiego złotą farbą i dorzuceniu setki marmurowych cherubinów, żeby wewnątrz człowiek czuł się jak po zjedzeniu dwudziestu pączków; przesłodzony i zemdlony.

I teraz właśnie zetkniemy się z tym złotem, którym wszystko jest pokryte.

*W tym miejscu chciałbym bardzo polecić zwiedzanie Bazyliki Jasnogórskiej, najpiękniejszy barok w Polsce.

Zmienne znaki w planie

Rozważmy następujące ciała planów.

	$(V$	$\wedge$	$V)$	$\vee$	$(\overline{V}$	$\wedge$	$\overline{V})$	$\Rightarrow$	R
V	0		1		0		1		0
S	0		0		0		0		0

	$(V$	$\wedge$	$\overline{V})$	$\vee$	$(\overline{V}$	$\wedge$	$V)$	$\Rightarrow$	R
V	0		1		0		1		0
S	0		0		0		0		0

Te dwa podobne plany przypisują do R_0 wartość odpowiednio równoważności i XORa zmiennych V_0 i V_1.

Można je zapisać jednym kodem w którym $\sim$ to negacja XORa..

	$(V$	$\wedge$	$(u$	$\sim$	$V))$	$\vee$	$(\overline{V}$	$\wedge$	$(\overline{u}$	$\sim$	$\overline{V}))$	$\Rightarrow$	R
V	0				1		0				1		0
S	0		0		0		0		0		0		0

Teraz jeżeli za u podstawimy $+$ (czyli Prawdę), dostaniemy równoważność, zaś jeśli $-$ (czyli Fałsz) XOR.

Deklaracja takiego planu wyglądałaby następująco.

	$(R(u))$	$(V,$	$V)$	$\Rightarrow$	R
V		0	1		0
S	0	0	1		0

Jak widać, i co jest nieco zaskakujące, zmienna u jest związana z Planem w inny, silniejszy sposób niż inne argumenty.

Zmienne typy w planie

Wyobraźmy sobie, że chcemy policzyć wyznacznik poniższej macierzy.

$$\Delta = \begin{vmatrix} V_0 & V_1 \\ V_2 & V_3 \end{vmatrix}$$

Gdyby typem zmiennych $V_0, \ldots, V_3$ były liczby całkowite, funkcja do liczenia wyznacznika miałaby postać jak niżej.

	$R(V,$	$V,$	$V,$	$V)$	$\Rightarrow$	R
V	0	1	2	3		0
A	8	8	8	8		8

	V	$\times$	V	$-$	V	$\times$	V	$\Rightarrow$	R
V	0		3		1		2		0
A	8		8		8		8		8

Co jeśli chcielibyśmy jednak zdefiniować wyznacznik dla macierzy zawierającej wartości binarne i napisać plan go obliczający? Zuse to zapisał to jak poniżej.

	$R(V,$	$V,$	$V,$	$V)$	$\Rightarrow$	R
V	0	1	2	3		0
S	0	0	0	0		0

	V	$\vee$	V	$\wedge$	V	$\vee$	V	$\Rightarrow$	R
V	0		3		1		2		0
S	0		0		0		0		0

Zauważmy, że te plany są dość podobne. Jak już się nauczyliśmy wcześniej, moglibyśmy zapisać jeden plan, który przyjmuje operatory jako argument. Pozostaje wówczas jeszcze problem typów, które mogą być różne, niekoniecznie obsługiwane przez dane operatory. Okazuje się, że typy też możemy podać jako argument planu!

	$(R(\alpha,$	$\Phi,$	$\Phi))$	$(V,$	$V,$	$V,$	$V)$	$\Rightarrow$	R
V		0	1	0	1	2	3		0
S				α	α	α	α		0

	$(V$	Φ	$V)$	Φ	$(V$	Φ	$V)$	$\Rightarrow$	R
V	0	0	3	1	1	0	2		0
S	α		α		α		α		α

Czy struktura zmiennych operatorów, znaków i typów nie powinna nam przypominać czegoś z współczesnych języków programowania? Okazuje się, że tak! Te narzędzia w sile wyrazu niezwykle przypominają nowoczesne wzorce z języka `C++`, makra czy polimorfizm parametryczny.

2.6. Rzeźbione drewna, czyli operacje logiczne

Gdy już oślepiło nas całe to złoto, opuszczamy wzrok i dostrzegamy kolejne ozdobniki, rzeźbione ławy dla wiernych czy błyszczącą ambonę. W Plankalkülu tymi ozdobami są operacje logiczne, do których Zuse zaliczył dość dużo funkcji.

2.6.1. Operatory $\forall$, $\exists$

Konrad Zuse stwierdził, że w Plankalkülu powinny istnieć odpowiedniki predykatów $\forall$ i $\exists$. Predykat $\forall$ zapisujemy jak niżej.

	$(x)R(x)$
V	
S	$0 \quad 0$

Powyższy zapis oznacza „Dla każdego x zachodzi $R(x)$". Rodzi to naturalne pytanie, co znaczy „dla każdego x"? Oznacza to każdą wartość którą można zareprezentować danym typem danych. Na przykład dla typu $S0$ oznacza to wartości $+$ i $-$. Dla typu $1.n$ oznaczałoby to wszystkie liczby n-bitowe. A dla innych typów liczbowych? No cóż, nie wiadomo, Zuse mógł nie przemyśleć, że liczb może być nieskończenie wiele, bądź typy miały mieć ograniczenie na długość liczb (choć takich liczb przecież też jest wystarczająco dużo, aby sprawdzenie wszystkich wartości było problematyczne). Równoważnym dla kodu wyżej kodem w Plankalkülu jest kod niżej.

	$L(A(x)) \Rightarrow Z$	$+\Rightarrow Z$	$W1(N(Z))$	$R\square(Z) \wedge Z \Rightarrow Z$	$Z\Rightarrow R$
V	0	1	0	$0 \quad 1 \quad 1$	$1 \quad 0$
K				i	
S	$\sigma \quad \square\times\sigma$	0		$0 \quad \sigma \quad 0 \quad 0$	$0 \quad 0$

Przeanalizujmy ten kod kawałek po kawałku*:

	$L(A(x)) \Rightarrow Z$	$+ \Rightarrow Z$
V	0	1
K		
S	$\sigma \quad \square\times\sigma$	0

Funkcja $A(x)$ zwraca typ (czyli zawartość linijki S) zmiennej x, zaś funkcja $L(x)$ przyjmuje typ i zwraca listę wszystkich wartości, które może dany typ reprezentować. Na przykład:

- $L(S0)$ oznacza listę $[+,-]$.
- $L(1.n)$ oznacza listę wszystkich ciągów n-bitowych.

Na początek tworzymy listę wszystkich możliwych wartości reprezentowanych typem σ i nazywamy ją Z_0.

*W tym przykładzie $\square$ nie jest częścią języka, to blank obrazujący, że funkcja ma działać dla dowolnie długiej listy. Symbol ten pojawia się jeszcze kilka razy

Następnie* do zmiennej boolowskiej Z_1 zapisujemy + (prawdę).

$$\begin{array}{r} W1(N(Z)) \\ 0 \end{array} \left[\begin{array}{llll} R\Box(Z) & \wedge & Z & \Rightarrow & Z \\ 0 & & 1 & & 1 \\ i & & & & \\ 0 & \sigma & 0 & & 0 \end{array}\right]$$

Potem mamy pętlę, która wykonuje się $N(Z_0)$ razy, czyli tyle, ile elementów ma lista Z_0. W ciele pętli sprawdzamy, czy plan $R\Box$ z i-tym elementem listy Z_0 jako argumentem zwraca prawdę. Jeżeli nie, do Z_1 zapisujemy fałsz, jeżeli tak, do Z_1 zapisujemy poprzednią wartość Z_1.

$$\left|\begin{array}{lll} Z & \Rightarrow & R \\ 1 & & 0 \\ & & \\ & & \\ 0 & & 0 \end{array}\right.$$

Na koniec wartość Z_1 zapisujemy do zmiennej wynikowej R_0.

Wykorzystanie predykatu $\exists$ wygląda nieco inaczej

$$\begin{array}{r} \\ V \\ K \\ S \end{array} \left|\begin{array}{l} (Ex) \end{array}\right. \left[\begin{array}{llll} x & \in Z & \wedge & \overline{x} \\ & 0 & & \\ & & & 1 \\ (\sigma, 0) & m \times (\sigma, 0) & & 0 \end{array}\right]$$

Powyżej mamy przykład użycia predykatu $\exists$, w którym sprawdzamy, czy istnieje x typu $(\sigma, S0)$ (para sigmy i wartości boolowskiej), należący do Z_1 (listy m takich elementów), taki, że drugi element pary (typu $S0$) jest fałszem (negacja jest prawdą).

Implementacja predykatu $\exists$ w Plankalkülu wygląda niemal identycznie, jak implementacja predykatu $\forall$, różni się tylko tym, że do Z_1 zapisujemy Fałsz, a w pętli robimy alternatywę, a nie koniunkcję, poprzedniej wartości Z_1, z wynikiem planu $R\Box$.

2.6.2. Operator $\hat{x}$

Operator $\hat{x}$ służy do wybierania z listy elementów spełniających dane własności, ale bez powtórzeń.

$$\hat{x}\left(x \in {}^{V}_{0} \wedge R\Box(x)\right) \Rightarrow {}^{R}_{0}$$

Notacja ${}^{V}_{0}$ oznacza zmienną V_0. Powyższe oznacza, że chcemy wybrać elementy x należące do V_0 dla których $R\Box(x)$ zwraca prawdę i zapisać je jako listę R_0. Na przykład dla poniższej listy V_0

*Przypomnienie: pionowa kreska jest separatorem

$$V_0 = (0, 3, 5, 4, 3, 3, 6, 12, 6, 4)$$

prawdziwe będą poniższe równości.

$$\hat{x}\left(x \in \binom{V}{0}\right) = (0, 3, 5, 4, 6, 12)$$

$$\hat{x}\left(x \in \binom{V}{0} \wedge Ger(x)\right) = (0, 4, 6, 12)$$

$Ger(x)$ jest funkcją przyjmującą licbę całkowitą, i zwracającą + lub − (prawdę lub fałsz), w zależności, czy x jest parzysty.

Tu i w kilku następnych podrozdziałach na koniec może pojawić się pytanie „Jak to współgra z ściśle typowanym językiem programowania, w którym podaję zawsze długość listy?" Czyli, krócej „Jaki jest typ listy wynikowej?". Jest to pytanie bardzo dobre, na które nie znam odpowiedzi. Niestety wydaje się, że Zuse też nie przemyślał, że może tu powstać problem.

2.6.3. Operator $\hat{\hat{x}}$

Ten operator jest bardzo podobny do $\hat{x}$, różnią się wyłącznie tym, że $\hat{x}$ wybiera elementy bez powtórzeń, zaś $\hat{\hat{x}}$ z powtórzeniami. Ten operator bardzo przypomina filter z współczesnych języków. Przykładowo, dla takiej samej listy jak w poprzednim podrozdziale.

$$V_0 = (0, 3, 5, 4, 3, 3, 6, 12, 6, 4)$$

Prawdziwe będą poniższe równości.

$$\hat{\hat{x}}\left(x \in \binom{V}{0}\right) = (0, 3, 5, 4, 3, 3, 6, 12, 6, 4)$$

$$\hat{\hat{x}}\left(x \in \binom{V}{0} \wedge Ger(x)\right) = (0, 4, 6, 12, 6, 4)$$

2.6.4. Operator $\acute{x}$

Operator $\acute{x}$ służy do znajdowania w liście elementu który spełnia podane warunki. Warunkiem użycia tego operatora jest to, że w liście występuje dokładnie jeden spełniający warunki element.

2.6.5. Operatory μx i λx

Operator μx to operator który ma znajdować w liście kolejne elementy spełniające warunek, i służy do wykorzystania w pętli. Używa się go jak w przykłdzie poniżej.

$$\begin{array}{c|c|c|c|c|} & W & \lceil \mu x(x \in V) \ \wedge \ R\square(x) \Rightarrow Z & & P(Z) \rceil \\ V & & 0 & & 0 \\ K & & & & \\ S & & \lfloor m\sigma & & \sigma \rfloor \end{array}$$

Taka pętla znajduje w liście V_0 elementy spełniające własność $R\square$ i zapisuje je w zmiennej tymczasowej Z_0, którą możemy wykorzystać w dalszej części ciała pętli (tutaj opisane jest to przez $P(Z)$).

Operator λx ma działanie analogiczne do μx, tylko robi to od końca listy.

2.6.6. Operator μx na prawo od znaku przypisania

Operator μ na prawo od znaku przypisania służy temu, by przypisywać wartości kolejnym elementom listy. Na przykład kod

$$\begin{array}{c|c|c|} & W & \lceil F \Rightarrow \mu R \rceil \\ V & & 0 \\ S & & \lfloor \sigma \rfloor \end{array}$$

oznacza „przypisz F do kolejneych pól listy R", odpowiadający temu kod w Plankalkülu wygląda jak poniżej.

$$\begin{array}{c|c|c|c|c|} & 0 \Rightarrow \varepsilon & W & \lceil F \Rightarrow R & \varepsilon + 1 \Rightarrow \varepsilon \rceil \\ V & & & 0 & \\ K & & & \varepsilon & \\ S & & & \lfloor \sigma & \rfloor \end{array}$$

2.6.7. Operatory $\wedge R, \vee R, \Sigma R, \Pi R$

Rozważmy wyrażenie poniżej.

$$\begin{array}{c|lllllllllll} & V & \wedge & V & \wedge & \ldots & \wedge & V & \wedge & \ldots & V & \Rightarrow & R \\ V & 0 & & 0 & & & & 0 & & & 0 & & 0 \\ K & 0 & & 1 & & & & i & & & n-1 & & \\ S & 0 & & 0 & & & & 0 & & & 0 & & \end{array}$$

Jest to koniunkcja n elementów listy zapisanej w V_0 Gdybyśmy chcieli napisać kod liczący taką koniunkcję, wolelibyśmy jednak napisać pętlę, która przechodzi po wszystkich elementach listy, zamiast ręcznie je wypisywać.

$$\begin{array}{c|c|c|c|c|}
 & + \Rightarrow \quad Z & W1(n) & \left[\begin{array}{ccc} V \wedge Z \Rightarrow & & Z \end{array}\right. & Z \Rightarrow \quad R \\
V & 0 & & \begin{array}{ccc} 0 & 0 & 0 \end{array} & \begin{array}{cc} 0 & 0 \end{array} \\
K & & & \left.\begin{array}{ccc} i & & \end{array}\right] & \\
S & 0 & & \begin{array}{ccc} 0 & 0 & 0 \end{array} & \begin{array}{cc} 0 & 0 \end{array}
\end{array}$$

Żeby było jednak jeszcze krócej, Zuse wprowadził również poniższą notację.

$$\begin{array}{c} W1(n)(V \Rightarrow \wedge R) \\ 0.i \quad 0 \end{array}$$

Działa to tak samo dla $\vee$, Σ i Π.

2.6.8. Zmienna jako wykładnik potęgi

Składnia podnoszenia do potęgi zapisanej w zmiennej jest podobna, jak ta służąca do indeksowania zmienną.

$$\begin{array}{c|cc}
 & Z \diagdown Z \\
V & 0 \quad 1 \\
K & \\
S & 8 \quad 8
\end{array}$$

Powyższy zapis oznacza podniesienie wartości zmiennej Z_0 do potęgi Z_1.

2.6.9. Lista pusta

Zuse zauważył, że jeżeli lista może mieć zmienną liczbę elementów, to w szczególności może mieć zero elementów.

Takie wyrażenie

$$\begin{array}{cc} \circledcirc \Rightarrow & Z \\ & 0 \\ & \square \times \sigma \end{array}$$

oznacza, że lista Z_0 stanowi wartość tymczasową i początkowo jest pusta. Przykładowe użycie może być następujące: jeśli wynikiem jest lista, a elementy tej listy są tworzone sekwencyjnie, na początku budowana lista wyniku jest pusta. Można sobie zadać pytanie, czy $\square$ w powyższym przykładzie jest częścią języka i oznacza zmienną liczbę elementów typu σ, czy jest częścią metajęzyka coś obrazującą. Podejrzewam, że w tym przypadku chodziło o pierwszą opcję, choć pewności nie mam.

Może się też zdarzyć tak, że lista Z_0 ma zawierać jeden element początkowy (Zuse go nazwał elementem tworzącym) V_0, wówczas możemy napisać jak niżej.

	V	$\Rightarrow$	Z
V	0		0
S	σ		γ

W linijce S dając V_0 inny typ niż ma Z_0 (takie coś będzie chyba tworzyć listę Z_0 z jednym elementem V_0). Dawanie różnych typów w przypisaniu jest dozwolone wyłącznie w takim jednym przypadku.

2.7. Z czego robimy cegły, czyli wszystkie typy danych

Na początku tego długiego rozdziału oglądaliśmy front kościoła, zastanawiając się, z czego jest on zbudowany. Okazuje się, że możemy zajrzeć nawet głębiej, i zobaczyć z czego są zrobione cegły, które wtedy oglądaliśmy. Ujrzymy wówczas wszystkie pozostałe typy danych. Również i przy nich Zuse zdecydował się na bardzo barokowe rozwiązania. Z tego powodu typów danych jest wiele, i mogą być różnie reprezentowane. Stąd bierze się ta długa lista poniżej.

1. Podstawowym typem danych była wartość boolowska, przez Zusego nazywana Ja-Nein-Wert (wartość tak-nie). Oznaczeniem zmiennej boolowskiej było $S0$, a jej wartości to $+$, $-$.

2. $S1.n$ — typ ogólny oznaczający ciąg n wartości $S0$.

3. $S2$ — typ ogólny oznaczający parę wartości dowolnego typu (Zauważmy, że w lini S w kodzie zapiszemy konkretny typ, nie $S2$).

4. $S3$ — typ ogólny oznaczający listę.

5. $S4$ — typ ogólny oznaczający listę par.

6. $A8$ — liczba bez dokładnej specyfikacji, być może ciąg bitów wspomniany w punkcie 2.

7. $A9.2$ — liczba całkowita dodatnia o reprezentacji binarnej ($n \times S0$ n-bitowa liczba całkowita, Zuse jednak nie podzielił się z nami, czym jest n, ani tu, ani w innych przykładach poniżej).

8. $A9.10$ — liczba całkowita dodatnia o reprezentacji dziesiętnej ($n \times S1.4$ n-cyfrowa liczba całkowita reprezentowana w BCD).

9. $A10.2.0$ — liczba całkowita o dowolnym znaku reprezentowana binarnie w systemie uzupełnień do 2 ($S1.n$).

10. $A10.2.1$ — liczba całkowita o dowolnym znaku reprezentowana binarnie z pierwszym bitem mówiącym o znaku $(S0, S1.(n-1))$.

11. $A10.10.0$ — podobnie jak wyżej w systemie uzupełnień do 2, ale z BCD $(n \times S1.4)$.

12. $A10.10.1$ — analogicznie $(S0, A9.10)$.

13. $A11.2$ — liczba ułamkowa dodatnia reprezentowana przez parę liczb całkowitych. Pierwsza mówi o części przed przecinkiem, druga o części po przecinku; $(A8.2, A8.2)$.

14. $A11.10$ — tak samo jak $A11.2$, tylko dziesiętnie: $(A8.10, A8.10)$.

15. $A12.2.\{0,1\}$ — liczba ułamkowa o dowolnym znaku reprezentowana binarnie.

16. $A12.10.\{0,1\}$ — liczba ułamkowa o dowolnym znaku reprezentowana dziesiętnie.

17. $A13$ — liczba zespolona. Reprezentowana przez parę liczb $A12$, gdzie pierwsza oznacza część rzeczywistą, a druga część urojoną.

2.8. I jeszcze trochę marmurowych cherubinów, czyli pozostałe operatory i funkcje

Zwiedzając barokowy kościół zawsze można znaleźć więcej ozdób, cherubinów, rzeźb i złoceń. A więc zobaczmy, co jeszcze Zuse dodał do swojego języka.

2.8.1. Funkcja I

Funkcja $I(x)$ zwraca pozycję argumentu x w liście. W jakiej liście? Tego nie wiadomo. O tym że warto by było ją podać jako argument, Zuse najpewniej zapomniał.

2.8.2. Funkcja Qz

Funkcja Qz (od *„Querzusammensetzung”*) działa niczym zamek błyskawiczny. Przyjmuje dwie listy Z_1 i Z_2 długości n i zwraca listę Z_3 n par takich, że i-ty element listy wynikowej jest parą sklejoną z i-tych elementów list wejściowych:

	Qz	$(Z,$	$Z)$	$\Rightarrow$	Z
V		1	2		3
K					
S		$n.\sigma$	$n.\gamma$		$n.(\sigma, \gamma)$

Co więcej, funkcji Qz można użyć dla dowolnej liczby list o tej samej liczbie elementów. A także na przykład dla listy Z_1 i pojedyńczego elementu Z_2, w którym to przypadku dostajemy następującą listę par Z_3, taką, że i-ta para składa się z i-tego elementu listy Z_1 i z Z_2.

	Qz	$(Z,$	$Z)$ $\Rightarrow$	Z
V		1	2	3
K				
S		$n.\sigma$	γ	$n.(\sigma,\gamma)$

2.8.3. Funkcja Lz

Funkcja Lz (od *Längszusammensetzun*) służąca do konkatenacji co najmniej dwóch list (ich długości mogą być różne, ale elementy muszą być tego samego typu).

2.8.4. Funkcja Nr

Funkcja Nr przyjmuje listę Z_1 i tworzy listę par Z_2, taką, że pierwszy element pary jest indeksem w tablicy:

	$Nr(Z)$	$\Rightarrow$	Z
V	1		3
K			
S	γ		$(1.n,\gamma)$

2.8.5. Funkcja $\ominus$

Wyrażenie:

	V	$\ominus$	V	$\Rightarrow$	Z
V	1		2		3
K					
S	$n.\sigma$		$n.\sigma$		0

mówi czy dwie listy zawierają te same elementy, niekoniecznie w tej samej kolejności.

Oryginalnie ta funkcja miała być przeznaczona dla list elementów typu $S4$, mającemu reprezentować relacje, później Zuse ją uogólnił dla dowolnych list. Tak więc pierwotną rolą tej funkcji było sprawdzać identyczność dwóch relacji.

2.8.6. Funkcje Ord

Funkcje Ord słuzyły do sortowania list. Istniało kilka funkcji Ord:

- $Ord0$ — Funkcja biorąca dwie liczby i zwracająca je w kolejności rosnącej.

- $Ord1$ — Funkcja sortująca listę rosnąco, realizowana przez sortowanie przez wstawianie.

- $Ord2$ — Tak jak $Ord1$, tylko sortowała listę malejąco.

- $Ord3$ — Sortowała listę par rosnąco ze względu na pierwszy element, nie uwzględniając drugiego.

- $Ord4$ — Sortowała listę par leksykograficznie rosnąco.

- $Ord5$ — Sortowała listę par leksykograficznie rosnąco, ale drugie elementy miały pierwszeństwo przed pierwszymi.

- $Ord6$ — Sortowała elementy par w liście par.

2.8.7. Operatory $\cap$, $\cup$

Wyrażenie

	Z	$\cap$	Z	$\Rightarrow$	Z
V	1		2		3
K					
S	$n \times \sigma$		$m \times \sigma$		$\square \times \sigma$

do Z_3 przypisuje część wspólną list Z_1 i Z_2. Operator $\cup$ działa analogicznie, przypisując do Z_3 sumę list Z_1 i Z_3.

2.8.8. Funkcje Sp

Istniały dwie funkcje Sp (od *Spalte einer Liste*):

- $Sp0$ — Funkcja przyjmowała listę i zwracała liczbę elementów występujących w niej bez powtórzeń.

- $Sp1$ — Funkcja przyjmowała listę i zwracała listę par składających się z elementów listy, i liczby ich wystąpień w liście będącej argumentem.

2.9. Przykładowe programy

P0

$$
\begin{array}{c|ccc}
 & R(V) & \Rightarrow & R \\
V & 0 & & 0 \\
K & & & \\
S & 10 & & 10
\end{array}
$$

$$
\begin{array}{c|ccc|ccc}
 & 1 & \Rightarrow & Z & 1 & \Rightarrow & Z \\
V & & & 0 & & & 1 \\
K & & & & & & \\
S & & & 10 & & & 10
\end{array}
$$

$$
\begin{array}{c|c|l}
 & W1(V) & \left[\begin{array}{lcccc} Ger(i) = + & \dot{\rightarrow} & \left[\begin{array}{ccccc} Z & + & Z & \Rightarrow & Z \\ 1 & & 2 & & 1 \\ & & & & \\ 10 & & 10 & & 10 \end{array}\right] \\ Ger(i) = - & \dot{\rightarrow} & \left[\begin{array}{ccccc} Z & + & Z & \Rightarrow & Z \\ 1 & & 2 & & 2 \\ & & & & \\ 10 & & 10 & & 10 \end{array}\right] \end{array}\right] \\
V & 0 & \\
K & & \\
S & 10 &
\end{array}
$$

$$
\begin{array}{c|ccc}
 & Z & \Rightarrow & R \\
V & 1 & & 0 \\
K & & & \\
S & 10 & & 10
\end{array}
$$

Powyżej widzimy plan P0, który ma za zadanie obliczać liczbę Fibbonacciego o indeksie V_0 i zwraca jej wartość poprzez zmienną R_0. W pierwszej instrukcji zapisujemy do zmiennych tymczasowych Z_0 i Z_1 wartość 1. W kolejnej instrukcji mamy pętlę, której ciało powtórzy się V_0 razy. W ciele pętli, w zależności, czy indeks i jest parzysty czy nie, sumę $Z_0 + Z_1$ zapisujemy do Z_0 lub Z_1. W ostatniej instrukcji obliczoną wartość zapisujemy do zmiennej wynikowej R_0.

P1

$$\begin{array}{c|l} & R(V) \;\Rightarrow\; R \\ V & \;0 \qquad\quad 0 \\ K & \\ S & \;10 \qquad\quad 10 \end{array}$$

$$\begin{array}{c|l} & V = 0 \vee V = 1 \mathbin{\dot{\to}} \left[\begin{array}{l} 1 \;\Rightarrow\; R \\ \qquad\quad 0 \\ \\ \qquad\quad 10 \end{array}\right] \\ V & 0 \qquad\quad\;\; 0 \\ K & \\ S & 10 \qquad\quad\;\; 10 \end{array}$$

$$\begin{array}{c|l} & V \geq 2 \mathbin{\dot{\to}} \left[\begin{array}{l} R2(V - 1) + R2(V - 2) \Rightarrow R \\ \qquad\quad 0 \qquad\qquad\quad 0 \qquad\quad 0 \\ \\ \qquad\quad 10 \qquad\qquad\quad 10 \qquad\quad 10 \end{array}\right] \\ V & 0 \\ K & \\ S & 10 \end{array}$$

Plan P1, jak w zasadzie widać, robi to samo co program P0, tylko, że w programie P1 korzystamy z rekursji.

2.10. Podsumowanie

Nie ma wątpliwości, że Konrad Zuse starał się stworzyć język kompletny, który spełniałby wszelkie potrzeby programistów na wiele lat do przodu. To właśnie mogło być jednym z powodów porażki jego języka, był zdecydowanie zbyt duży i skomplikowany jak na tamte czasy. Nie pomogło również to, że nie miał w zasadzie nic do czynienia z zachodnimi ośrodkami akademickimi, jego praca była chałupnicza, w dodatku tworzona w słusznie odciętych od świata Niemczech. Trzeba jednak oddać cesarzowi co cesarskie, Zuse opisał język znacznie wyprzedzający swoje czasy. Rozumiał też, że język musi być uruchamiany na rzeczywistej maszynie, i w ostatnich kawałkach swojej pracy opisuje problemy takie jak przepełnienia, konwersje typów czy pojawienie się reszty z dzielenia całkowitego. Pokazuje to, że mimo wrażenia które można odnieść po przeczytaniu tego rozdziału, Plankalkül nie był całkiem oderwany od rzeczywistości, choć z całą pewnością nie był do niej przystosowany. Był po prostu hipsterem wśród języków programowania.

Jednakowoż zaprojektowania języka bez wcześniejszego doświadczenia z wysokopoziomowym językiem programowania, a w dodatku bez implementowania go, jest bardzo trudne. Dużo z jego problemów widać dopiero przy próbie implementacji lub przynajmniej bardziej praktycznego przemyślenia. Nie zmienia to jednak faktu, że Zuse myślał przyszłościowo. Co więcej, nie wszystkie rzeczy o których wiedział,

że kiedyś się pojawią na świecie włożył do swojego języka. Rozumiał on, że kiedyś komputery będą używane do trzymania danych o ludziach, i będzie można taki komputer prosić o wydrukowanie danych na przykład mężczyzn o wzroście powyżej 165cm. Wiedział też, że komputery będą mogły być używane w bardziej przyziemnych celach, na przykład do liczenia wypłat pracowników, obciążeń budowlanych, gry w szachy, liczenia symbolicznego pochodnej[4], czy wielu innych praktycznych zastosowaniach[5]. Jednak nie próbował wkładać możliwości implementowania takich rzeczy do Plankalküla.

Rozdział 3.

Implementacja Plankalküla

Wyjdźmy z Zusowskiego kościoła i zastanówmy się nad własną implementacją języka Plankalkül. Pewnym problemem z Plankalkülem jest to, że wiele jego rozwiązań nie pasuje do współczesnych komputerów, niektóre są wewnętrznie sprzeczne a inne wręcz niemożliwe do zrealizowania. Dlatego nasza implementacja będzie inspirowana oryginalnym Plankalkülem, lecz także sporo w nim zmodyfikuje, a niektóre pomysły nawet porzuci. Ponieważ jest to język podobny do Plankalküla, jednak nie identyczny nazwiemy go Plankalkül24*.

3.1. Dwuwymiarowość

Pierwszą niewygodą w programowaniu Plankalküla jest dwuwymiarowa składnia (a już zwłaszcza indeksowanie zmiennymi), którą niełatwo zrealizować we współczesnych komputerach, w których pliki składają się z ciągów znaków, nie z wielowierszowych linijek. Dlatego w mojej implementacji korzystam z składni spłaszczonej, w której wszystko mieści się w jednej linijce. I tak weźmy przykład zmiennej z Plankalküla.

	Z
V	1
K	
S	10

W Plankalkülu24 napiszemy ją jak niżej.

```
Z[1;;10]
```

Rozważmy poniższe wyrażenie z Plankalküla.

*Kod interpretera języka Plankalkül24 znajduje się w repozytorium na stronie github.com/kubamarc/plankalkul

	Z	$+$	Z	$\Rightarrow$	Z
V	0		1		2
K	2				
S	10		10		10

Je zapiszemy w naszym języku jak poniżej.

```
Z[0;2;10] + Z[1;;10] => Z[2;;10]
```

3.2. Deklaracje

Tak jak Zuse chcemy typem odwoływać się do odpowiedniego komponentu zmiennej, nie zaś do pełnego typu. To zaś rodzi pewien problem. Co jeśli w pierwszej instrukcji chcemy przypisać wartość do drugiego elementu listy bądź krotki? Zapiszemy instrukcję jak niżej.

```
5 => Z[2;2;8]
```

I co ma w tej sytuacji zrobić interpreter, który nie wie, jakiego typu jest Z_2? Aby mu pomóc możemy wprowadzić specjalną instrukcję do deklarowania zmiennej.

```
Deklarieren Z[2;;10.8]
```

3.3. Plany

Jak widzieliśmy, w Plankalkülu nazwy poszczególnych planów przy ich deklaracjach występowały raczej na poziomie meta, i nie było jasne, gdzie zaczyna się wykonywać program. Dlatego trzeba te kwestie usystematyzować. Po pierwsze, nazewnictwo przy deklaracji, w Plankalkülu Zusego pisaliśmy jak poniżej.

	$R(V,$	$V)$	$\Rightarrow$	(R)
V	0	1		0
S	0	0		0

W implementacji napiszemy jak niżej.

```
P 9.10 ()()(V[0;;0]; V[0;;1]) => (R[0;;0]) {}
```

Jak widać, plan jest nazwany w miejscu deklaracji, zaś jego ciało musi znajdować się w {}, które służą do ograniczania kodów bloku (tak jak u Zusego służyły do tego duże nawiasy kwadratowe). Widać jednak też dość dużą liczbę nawiasów przed znakiem przypisania. W pierwszym mogą być wyłącznie zmienne typowe (o nazwach `mi`[*indeks*]), w drugim zmienne operatory (o nazwach `Phi`[*indeks*]) a w trzecim zmienne liczbowe. Przykładowo jak niżej.

```
P 2 (mi[1]) (Phi[1]) (V[0;;mi[1]]; V[1;;mi[1]]) => (R[0;;mi[1]]) {
    V[0;;mi[1]] Phi[1] V[1;;mi[1]] => R[0;;mi[1]]
}
```

W Plankalkülu24 plany mogą zwracać tylko jedną wartość.

3.4. Podstawowe operatory i wartości

Ponieważ na typowej klawiaturze nie występują symbole $\wedge$ czy $\vee$, zastępujemy je symbolami $*$ i $+$. Ciężko jest również zapisać negację w sposób w jaki zapisywał ją Zuse (kreska nad wyrażeniem), dlatego wprowadzamy symbol negacji $\sim$. I tak, chcąc przypisać negację zmiennej Z_2 do zmiennej R_5 napiszemy jak niżej.

```
~Z[2;;0] => R[5;;0]
```

Zaś chcąc napisać $R_3 = Z_1 \vee \neg Z_2$ następująco.

```
Z[1;;0] + ~Z[2;;0] => R[3;;0]
```

Zamiast znaków $+$ i $-$ do wyrażania wartości boolowskich, aby nie przeciążać tych symboli użyjemy, żeby poczuć w sobie tego niemieckiego ducha, słów „Ja" i „Nein" (Tak jak Zuse nazywał wartości boolowskie „Ja-Nein-Wert"). Czyli, chcąc przypisać Prawdę do zmiennej Z_7 napiszemy tak jak pod spodem.

```
Ja => Z[7;;0]
```

3.5. Pętle, wyrażenia warunkowe, operatory przerwania wykonania

Żaden przyzwoity język nie może się obyć bez wyrażeń warunkowych, tak więc i w tej implementacji można z nich korzystać. Przykłady użycia wyrażeń warunkowych wyglądają następująco.

```
V[1;;10] = 0 -> Nein => R[0;;0]
```

To co przed `->` jest warunkiem wykonania, zaś to co na prawo od `->` jest tym, co się wykonuje przy spełnionym warunku. W tym przypadku jeśli V_1 jest równe 0, zapisujemy Fałsz do R_0. Po prawej od strzałki możemy mieć też kilka instrukcji.

```
Z[3,,10] < 10 -> {
    Z[3,,10] + 1 => Z[3,,10]
    Z[5,,0] * Ja => Z[5,,0]
}
```

W powyższym kodzie jeżeli Z_3 jest mniejsze niż 10, zwiększamy je o 1, zaś do Z_5 zapisujemy koniunkcję Z_5 z Prawdą.* W implementacji Plankalküla możemy korzystać zarówno z pętli W, jak i pozostałych pętli W_0, W_1, W_2,...,W_5, które zachowują się tak jak w oryginalnym Plankalkülu.

```
W {
  Z[3;;10] < 10 -> {
    Z[3;;10] + 1 => Z[3;;10]
  }
  Z[3;;10] < 20 -> {
    Z[3;;10] + 1 => Z[3;;10]
  }
}
```

W tym przypadku pętla będzie się wykonywać dopóki Z_3 jest mniejsze od 20. Kiedy będzie mniejsze od 10 będą wykonywać się dodawania z obu warunkowych instrukcji, gdy będzie wynosić przynajmniej 10 tylko z drugiego.

Pozostałe pętle mają składnię analogiczną do poniższej, która wykona się pięć razy dodając po 1 do Z_3, chyba, że Z_3 osiągnie wartość 3, wówczas wykona się instrukcja Fin2 warunkowego wyjścia z pętli.

```
W 5 (1, 5) {
    Z[3;;10] + 1 => Z[3;;10]
    Z[3;;10] = 3 -> Fin2
}
```

3.6. Indeksowanie zmienną i iteratorem

Skoro już omówiliśmy pętle, na pewno chcielibyśmy wiedzieć jak napisać pętlę w której iterujemy się po tablicy. Jak wiemy z jednego z przykładów powyżej, chcąc zapisać odwołanie do trzeciego elementu listy o dwudziestu elementach napiszemy jak niżej.

```
Z[1;3;10]
```

Możemy się jednak odwoływać również zmienną.

```
Z[1;Z[2;;10];10]
```

Czy też iteratorem w pętli.

```
W 3 (2, 12) {
    i - 1 => Z[2;; 10]
    i - 2 => Z[3;; 10]
    Z[1; Z[2;; 10]; 10] + Z[1; Z[3;; 10]; 10] => Z[1; i; 10]
}
```

*Ważne: w wyrażeniach arytmetycznych, czy porównania stałe nie mogą występować po lewej od operatora.

Jeżeli pętle są zagnieżdżone, iterator trzeba indeksować.

```
W 1 (5) {
    W 1 (5) {
        i[0] - 1 => Z[2;; 10]
        i[1] - 2 => Z[3;; 10]
        Z[1; Z[2;; 10]; 10] + Z[2; Z[3;; 10]; 10] => Z[3; i[0]; 10]
    }
}
```

indeksujemy od 0, tak, że $i[0]$ odnosi się do najbardziej zewnętrznej pętli.

3.7. Typy i struktury danych

Jak widzieliśmy w poprzednich przykładach, możemy bez problemu używać list, których składnia deklaracji jest w zasadzie identyczna z jedną ze składni Zusego. Ponadto możemy używać krotek.

```
Deklarieren Z[1;;(8, 8, 10)]
```

Mamy też dostęp do pewnych typów danych:

1. Wartości binarnych:

   ```
   Ja => Z[2;;0]
   ```

2. Liczb całkowitych:

   ```
   -3 => Z[3;;10]
   7 => Z[4;;10]
   ```

3. Liczb rzeczywistych:

   ```
   1.3 => Z[5;;12]
   -7.5 => Z[6;;12]
   ```

4. Oraz liczb zespolonych:

   ```
   (1.3, -2.5) => Z[5;;13]
   (-7.5, 4) => Z[6;;13]
   ```

3.8. Operacje I/O

Istotnym brakiem w Plankalkülu, który uważny czytelnik z pewnością zauważył jest brak klarownego sposobu komunikowania się z programem. Należy pamiętać, że komputery Zusego (i w ogóle starożytne komputery) działały nieco inaczej, program

nie wczytywał danych, a tylko pobierał je z pamięci, stąd nie było operacji wczytywania danych, ani ich wypisywania, jako, że można było sobie po prostu wydrukować jakieś miejsce z pamięci. Jednak my jesteśmy już rozleniwieni dostępem do funkcji I/O, dlatego trochę sobie ułatwimy, dodając operację *Drucken*, która służy wypisywaniu wartości. Operacja potrafi wypisać wyrażenia o typach podstawowych (tj. wartości boolowskie, i liczby).

3.9. To teraz coś zaprogramujmy!

Uzbroiliśmy się w całą wiedzę potrzebną do napisania pierwszego programu! A więc zróbmy to. Oczywistym pierwszym wyborem programu do napisania jest taki, który wypisuje „Hello World". Jednakowoż, nie mamy takiej możliwości, bo nie mamy w naszym języku znaków. Z tego powodu tak jak w rozdziale 2.9. napiszemy obliczanie dwunastej liczby Fibbonacciego dwiema metodami. Pierwsza metoda jest iteracyjna:

```
P 1 ()()() => () {
    Deklarieren Z[1;;12.10]
    1 => Z[1; 0; 10]
    1 => Z[1; 1; 10]
    Z[1; 1; 10] => Z[1; 1; 10]
    W 3 (2; 12) {
      i - 1 => Z[2;;10]
      i - 2 => Z[3;;10]
      Z[1; Z[2;;10]; 10] + Z[1; Z[3;;10]; 10] => Z[1; i; 10]
    }
    Drucken Z[1; 11; 10]
}
```

Tworzymy tu tablicę Z_1, do której pierwszych dwóch elementów zapisujemy 1, a następnie iterujemy się za pomocą pętli $W3$ od 2 do 11 zapisując do i-tego elementu sumę dwóch poprzednich. Na koniec wypisujemy ostatnią wartość.

Drugą metodą napisania takiego programu jest metoda rekurencyjna:

```
P 1 ()()() => () {
    Drucken P 2 () () (12)
}

P 2 ()()(V[1;;10]) => (R[0;;10]) {
    V[1;;10] + 1 = 1 -> 1 => R[0;;10]
    V[1;;10] = 1 -> {1 => R[0;;10]}
    V[1;;10] > 1 -> {
      P 2 ()()(V[1;;10] - 1) + P 2 ()()(V[1;;10] - 2) => R[0;;10]
    }
}
```

W funkcji $P1$ wywołujemy tylko funkcję $P2$ z argumentem liczbowym. W funkcji $P2$ sprawdzamy najpierw, czy argument V_1 jest równy 0 lub 1, jeśli tak, do R_0 przypisujemy 1 (czyli zwracamy 1), jeśli nie, do R_0 przypisujemy wartość sumy wywołań P_2 dla argumentów $V_1 - 1$ i $V_1 - 2$ (czyli typowo dla ciągu Fibbonacciego)*.

3.10. Inne zaprogramowane funkcje

- Funkcja N — Funkcja przyjmująca jako argument listę i zwracająca jej długość
- Funkcja Ger — Funkcja przyjmująca jako argument liczbę typu 10 i zwracająca jej parzystość.
- Funkcja Ord — Funkcja sortująca listę, korzysta z sortowania Pythonowego, więc sortuje tak jak ono.

*Inne przykładowe programy dostępne są w katalogu `examples` w repozytorium na stronie github.com/kubamarc/plankalkul

Rozdział 4.

Podsumowanie

Pisząc tę pracę, zwłaszcza rozdział opisujący pomysł Zusego, w pewnej mierze miałem na celu wywołanie w czytelniku odczuć podobnych do tych, których sam doznałem. Szoku kulturowego pomieszanego ze zdumieniem i podziwem. Szoku kulturowego, ponieważ Plankalkül jest językiem diametralnie innym od tych, które znamy współcześnie, zaś podziwu ponieważ Plankalkül miał być językiem bardziej zaawansowanym i dojrzałym niż którykolwiek powstały przez następnych trzydzieści lat. Jednocześnie próbowałem przekazać swoje uczucie przytłoczenia spowodowanego wielością rozwiązań wymyślonych dla Plankalküla. Czytanie oryginalnej pracy przypominało chodzenie po górach ukształtowanych przez lodowce, gdy dochodząc do przełęczy człowiek odkrywa, że to jednak nie przełęcz a kolejne wybrzuszenie morenowe. Ale jednocześnie widać, że już tam-tuż jest ta przełęcz*. W takiej sytuacji samo się wyrywa „A niech to! Jeszcze jedna morena?" Mam nadzieję, że moje streszczenie budziło nieco podobne uczucia i myśli „A niech to! Jeszcze jedna funkcja?", zwłaszcza, że ja czytając oryginał takich doznawałem.

Lecz po przeczytaniu opisu Plankalküla nadszedł dla mnie czas, żeby go zaimplementować. Jeszcze przed pisaniem miałem świadomość, że trzeba będzie dokonać pewnych uproszczeń,† jednak gdy próbowałem implemenować niektóre z własności i funkcji ze zdumieniem odkrywałem, że biorąc pod uwagę inne aspekty języka, są one niemożliwe do zaimplementowania w niezmienionej formie.

Niektórzy mogliby się oburzyć i z rozbawieniem stwierdzić, że to świadczy o niższości Plankalküla i jego twórcy w stosunku do późniejszych języków. Jednak ja bym się nie zgodził z taką konkluzją. Raczej bym podkreślił swoją tezę ze wstępu, że jest to to co odróżnia języki stworzone od wyewoluowanych. Że twórca języka bardziej zaawansowanego powinien najpierw zobaczyć i zrozumieć jakiś język mniej zaawansowany. Zuse nie miał takiego luksusu. Jednocześnie Zuse nie miał możliwości implementowania swojego języka, jedynymi komputerami do których mógł mieć dostęp były te, które sam zbudował, co dodatkowo utrudniało tworzenie zaawanso-

*Oczywiście to nie ona, to kolejne wybrzuszenie morenowe.

†Zwłaszcza w kwestii dwuwymiarowej składni.

wanego języka. Właśnie to czyni jego dzieło bardziej imponującym, mimo pewnych drobnych* problemów języka.

*Lub mniej drobnych.

Bibliografia

[1] Zamówienie na budowę maszyny do wykonywania planów (*Kriegsauftrag für ein Planfertigungsgerät*) http://zuse.zib.de/file/1rUAfKDkirW8o3gT/31/2a/3c/ba-2e00-45eb-84cf-872470763f9a/0/original/b73a7928e2cd01d47359112295f43533.pdf

[2] Konrad Zuse; *The Computer - My Life*

[3] Konrad Zuse; *Der Plankalkül* http://zuse.zib.de/album/ImvGLEqrWp9c9LA/item/DB2j_t_w1fbxvaiq

[4] Konrad Zuse; *Über den Allgemeinen Plankalkül als Mittel zur Formulierung schematisch-kombinativer Aufgaben* http://zuse.zib.de/album/ImvGLEqrWp9c9LA/item/USg6f9md3p0B9SCf

[5] Konrad Zuse; *Über Theorie und Anwendungen logistischer Rechengeräte* http://zuse.zib.de/album/ImvGLEqrWp9c9LA/item/eFKWFP2kyF9neCW8