Mała rzecz o schematach blokowych

D. Daniecka

październik 2002

„Każdy ludzki problem ma rozwiązanie
— eleganckie, wiarygodne i błędne“

H. L. Mencken

1 Schematy blokowe — czy warto się z nimi męczyć?

Jeśli ktoś napisał dziesiątki tysiący linii kodu, z których i twórca i użytkownik tego kodu byli zadowoleni; jeśli owa osoba dotychczas nie słyszała o schematach blokowych, nigdy ich nie widziała lub widziała, ale czuła prawie fizyczny ból ;) na samą myśl o ich tworzeniu, to odpowiedź na powyższe pytanie jest jednoznaczna i brzmi: NIE, nie warto zadręczać się schematami blokowymi. Zdrowie i święty spokój są w życiu najważniejsze.

Jeśli jednak jesteśmy osobami, które zaczynają pisać programy i chcemy nauczyć się to robić w elegancki sposób, to warto się przełamać do schematów, choćby na jakiś czas. Rozpoczynając od schematów bardzo szczegółowych, gdzie prawie każdemu elementowi schematu odpowiada pojedyncza instrukcja kodu programu; kończąc na schematach, które opisują ogólną strukturę programu, czy obrazują kolejność podejmowanych działań (zwykle każdemu elementowi odpowiadać będzie wiele instrukcji kodu programu). Nabycie umiejętności patrzenia z bardzo bliska i z bardzo daleka na tworzony program jest już tylko o krok od umiejętności niezbędnych przy projektowaniu małych i całkiem sporych systemów. Termin *pisać programy* błędnie podkreśla czasownik *pisać*. Tymczasem umieć pisać programy, to przede wszystkim umieć je projektować. Bez względu na wielkość programu, jest on dobrze zaprojektowany nie tylko wtedy, gdy wykonuje określone zadanie poprawnie, lecz gdy ma czytelną i logiczną strukturę oraz gdy algorytm rozwiązania postawionego problemu jest przemyślany i zoptymalizowany pod względem wybranych przez projektanta kryteriów. Schematy odzwierciedlają charakter proponowanego rozwiązania, często odkrywają błędną kolejność podejmowanych w algorytmie decyzji. Równie ważne jest to, że schematy blokowe są niezależne od języka, w którym zaimplementujemy algorytm oraz nie wymagają dodatkowych założeń ze względu na to, czy piszemy w języku niskiego czy wysokiego poziomu.

Zatem, czy warto? Opinie są bardzo podzielone. Wielu ludzi wspomina okres studiów, wypełnionych schematami za czas stracony. Wielu innych uważa, że przez schematy trzeba przejść, bo „*to zmusza do myślenia*“. No to się zmuśmy... lekko, by nie zrazić się do schematów.

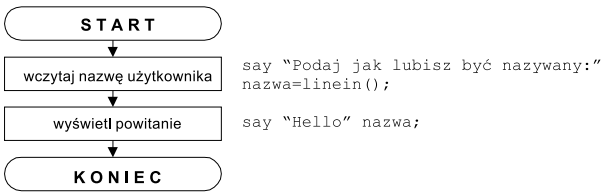
2 Schemat blokowy jako forma zapisu algorytmu

2.1 Algorytmy sekwencyjne

Książkowe kursy programowania w dowolnym języku w ostatnich latach rozpoczynają się od wyświetlenia sakramentalnego już „`Hello World`“. Program ten działa zawsze w ten sam sposób wyświetlając do znudzenia ten sam tekst, nie zważając, czy program uruchomił prezes wielkiej firmy, czy uczeń pierwszej klasy. Jeden z nich może czuć się niedowartościowany, drugi poczuć pępkiem świata. Poprawmy więc program tak, aby każda osoba uruchamiająca program była zadowolona.

Zdefiniujemy zadanie; przedstawimy pomysł na jego rozwiązanie; rozwiązanie przedstawimy w postaci algorytmu (ściśle określonych kroków, które należy podjąć aby zrealizować wybrane rozwiązanie); następnie narysujemy schemat blokowy, a na końcu metodą „*małpy*“— ostatecznie wysiłek podjęty dotychczas jest wystarczający :) — napiszemy kod programu.

- **zadanie:**
  Każdy z użytkowników programu ma być przywitany i nie czuć się obrażonym po jego uruchomieniu;
- **rozwiązanie:**
  Zapytajmy użytkownika jak chciałby abyśmy do niego mówili, a następnie wyświetlmy zmodyfikowaną odpowiednio wersję „`Hello World`“;
- **algorytm:**
  1. wczytaj od użytkownika, jak lubi gdy go nazywają;
  2. wyświetl powitanie: `Hello` + informacja, którą wczytałeś od użytkownika;
- **schemat blokowy:**



spostrzeżenia:
Nie ma co udawać, program jest prymitywny, choć troszkę mniej niż jego standardowa wersja. Prymitywność ta pozwala się jednak skupić na ważniejszym elemencie: **sposobie postępowania**. Najpierw zdecydowalismy się jaki problem rozwiążemy, potem jak go rozwiążemy, a następnie podaliśmy algorytm rozwiązania problemu w postaci kolejnych kroków i schematu blokowego. Pisanie kodu było już właściwie tylko i wyłącznie formalnością.

Zapisanie algorytmu w punktach określa w sposób jednoznaczy, kiedy każdą z operacji należy wykonać. Nie ma bowiem sensu wyświetlać powitania, dopóki nie dowiemy się jak mamy nazywać użytkownika, który uruchomił nasz program. Dokładnie tą samą własność ma wyrażenie algorytmu w postaci schematu blokowego (strzałki pełnią rolę przejścia do kolejnego punktu).

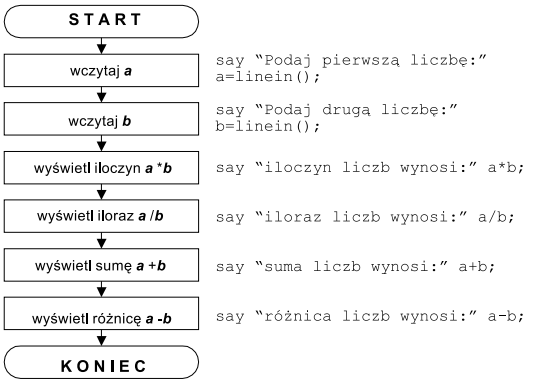
Przedstawiony algorytm jest przykładem algorytmu sekwencyjnego. Każda ze zdefiniowanych czynności wykonywana jest **zawsze i dokładnie jeden raz**.

Gdyby wszystkie programy były tego typu, programowanie byłoby lekko nudnawe. Nie istniałaby także dyskusja, czy warto rysować schematy blokowe. Nie byłoby warto, ponieważ kolejność czynności jest czytelnie określona, gdy zapiszemy algorytm w postaci punktów.

2.2 Czas najwyższy dokonać jakiegoś wyboru...

Wszystkie instrukcje kodu programu realizujące algorytm sekwencyjny, jak wspomnieliśmy, wykonują się **zawsze**. Napiszmy więc program, który będzie działał jak prymitywny czterodziałaniowy kalkulator.

- **zadanie:**
  Program ma pobrać od użytkownika dwie liczby, a następnie wyświetlić ich iloczyn, iloraz, sumę i różnicę.
- **rozwiązanie:**
  Wczytujemy dwie liczby, a następnie wyświetlamy wyniki działań: $*$, /, $+$, $-$;
- **algorytm:**
  1. wczytaj od użytkownika pierwszą liczbę;
  2. wczytaj od użytkownika drugą liczbę;
  3. wyświetl iloczyn wczytanych liczb;
  4. wyświetl iloraz wczytanych liczb;
  5. wyświetl sumę wczytanych liczb;
  6. wyświetl różnicę wczytanych liczb;
- **schemat blokowy:**

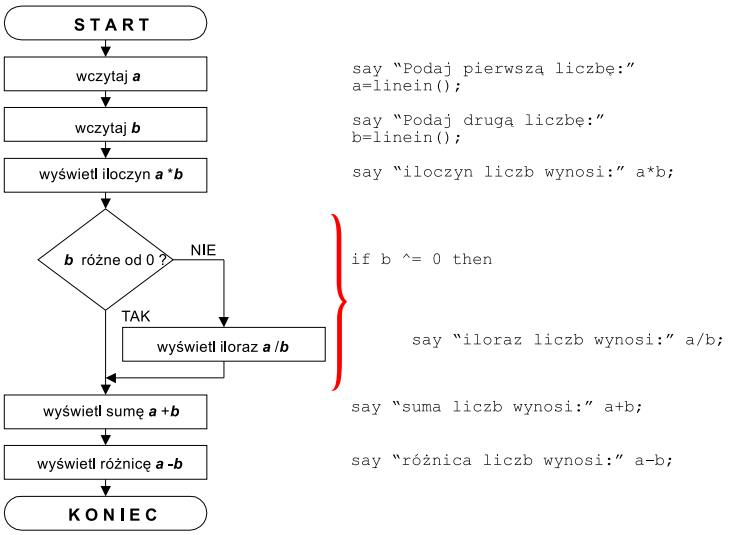


spostrzeżenia:
Wszystko działa jak w szwajcarskim zegarku dopóki druga z wczytywanych liczb nie jest zerem. Możemy zabronić użytkownikowi podawać wartość zero jako drugą liczbę, choć przecież pozostałe działania możemy wykonać. Dobrze byłoby mieć możliwość sprawdzenia w trakcie działania programu, czy są wystarczające warunki, aby wykonać pewne instrukcje. Dokonajmy niezbędnych zmian w algorytmie.

- **algorytm po korektach (I):**

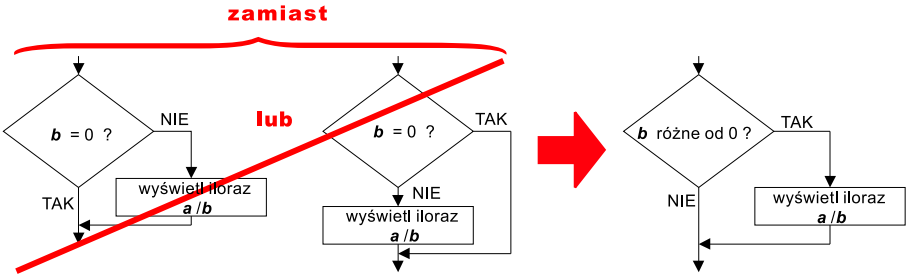
  1. wczytaj od użytkownika pierwszą liczbę;
  2. wczytaj od użytkownika drugą liczbę;
  3. wyświetl iloczyn wczytanych liczb;
  4. sprawdź wartość drugiej liczby; jeżeli jest różna od zera wykonaj punkt 5, jeśli jest równa zero przejdź do punktu 6;
  5. wyświetl iloraz wczytanych liczb;
  6. wyświetl sumę wczytanych liczb;
  7. wyświetl różnicę wczytanych liczb;

- **schemat blokowy po korektach (I):**



spostrzeżenia:
Zapis słowny algorytmu wymaga od nas pewnego skupienia, aby dowiedzieć się jaki krok ma być wykonany jako kolejny po kroku 4. W przypadku schematu blokowego, w ogóle nie musimy czytać, co w poszczególnych elementach jest zapisane. Jak wyglądałaby czytelność algorytmu zapisanego słownie, gdybyśmy dokonywali wyboru (czy instrukcje mają być wykonane, czy nie) w kilku miejscach?

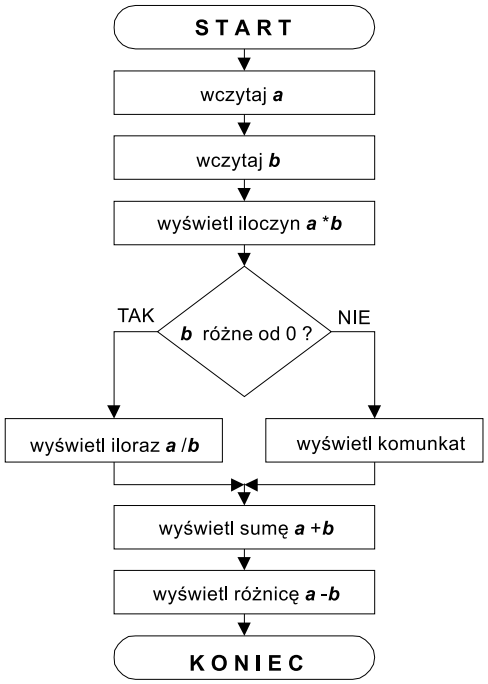
Najczęściej spotykanym błędem w tworzonym schemacie, który uniemożliwia bezpośrednie napisanie fragmentu kodu jest umieszczenie na schemacie dosłownego „*ominięcia*“:



Należy więc pamiętać, że pytanie (warunek), od którego zależy wykonanie lub niewykonanie pewnych instrukcji, należy skonstruować tak, aby instrukcje były wykonywane, gdy odpowiedź na pytanie brzmi: TAK.

Program działa już bez zastrzeżeń, choć użytkownik nie wiedzący, że nie można dzielić przez zero może mieć wrażenie , że program raz wyświetla iloraz, a raz nie — cóż za niederministyczny twór udało nam się napisać! Użytkownik budzi nas więc o trzeciej nad ranem, tłumacząc nam, że program, który mu napisaliśmy jest bardzo dobry, ale czasem nie działa... Trzecia rano, to nie pora na żarty, więc poprawmy program tak, aby nie było już żadnych wątpliwości. Wyświetlimy stosowny komunikat w przypadku, gdy nie można policzyć ilorazu.

- **algorytm po korektach (II):**
  1. wczytaj od użytkownika pierwszą liczbę;
  2. wczytaj od użytkownika drugą liczbę;
  3. wyświetl iloczyn wczytanych liczb;
  4. sprawdź wartość drugiej liczby; jeżeli jest różna od zero idź do punktu 5, w przeciwnym wypadku idź do punktu 6;
  5. wyświetl iloraz wczytanych liczb; idź do punktu 7;
  6. wyświetl informację, że iloraz nie istnieje; idź do punktu 7;
  7. wyświetl sumę wczytanych liczb;
  8. wyświetl różnicę wczytanych liczb;
- **schemat blokowy po korektach (II):**




```
say "Podaj pierwszą liczbę:"
a=linein();

say "Podaj drugą liczbę:"
b=linein();

say "iloczyn liczb wynosi:" a*b;


if b <> 0 then
      say "iloraz liczb wynosi:" a/b;
else
      say "iloraz liczb nie istnieje";

say "suma liczb wynosi:" a+b;

say "różnica liczb wynosi:" a-b;
```

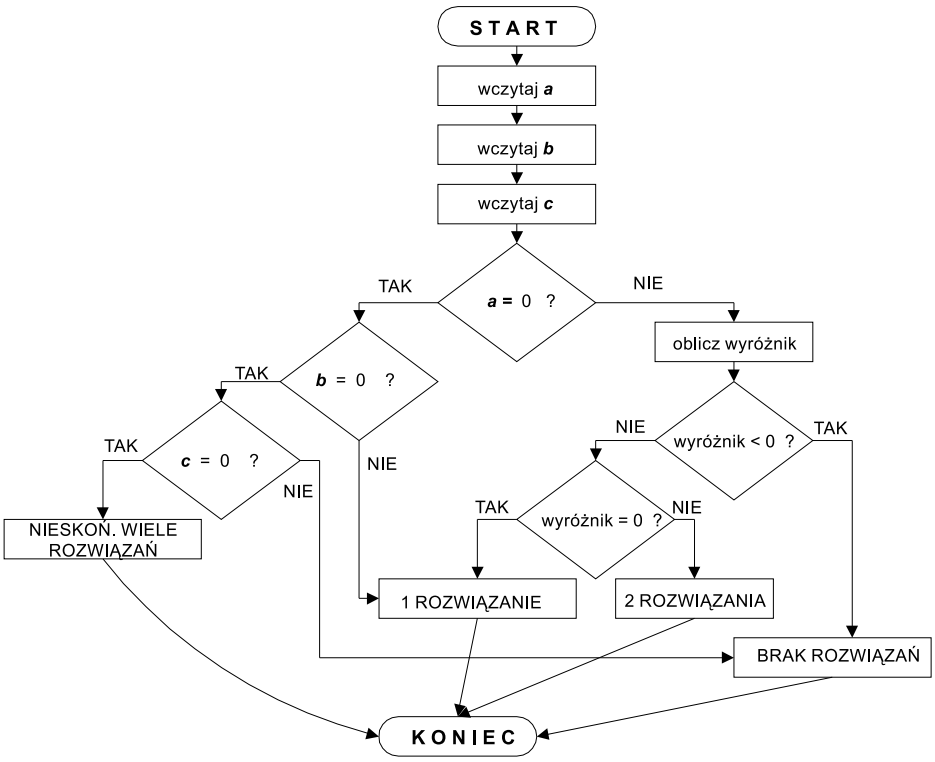
2.3 Jeśli mamy wybór, to może odrobina elegancji...

Sam fakt, że możemy napisać programy:

- w których pewne instrukcje mogą się wykonać, ale nie muszą;
- w których zamiast pewnych instrukcji wykonają się inne;

może być już źródłem małych nieszczęść. Dopóki wolno nam było rysować pojedyncze strzałki (od elementu do elementu) trudno było cokolwiek wyfantazjować. W przypadku, gdy dokonujemy pewnego wyboru (niekoniecznie jednokrotnie) można już wykazać się inwencją np. w następujący sposób:

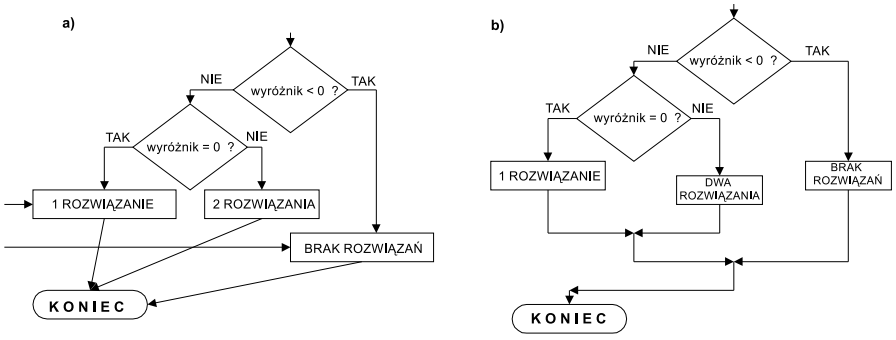
- **zadanie:**
  Podaj liczbę rozwiązań równania: $ax^2 + bx + c = 0$.
- **schemat blokowy:**



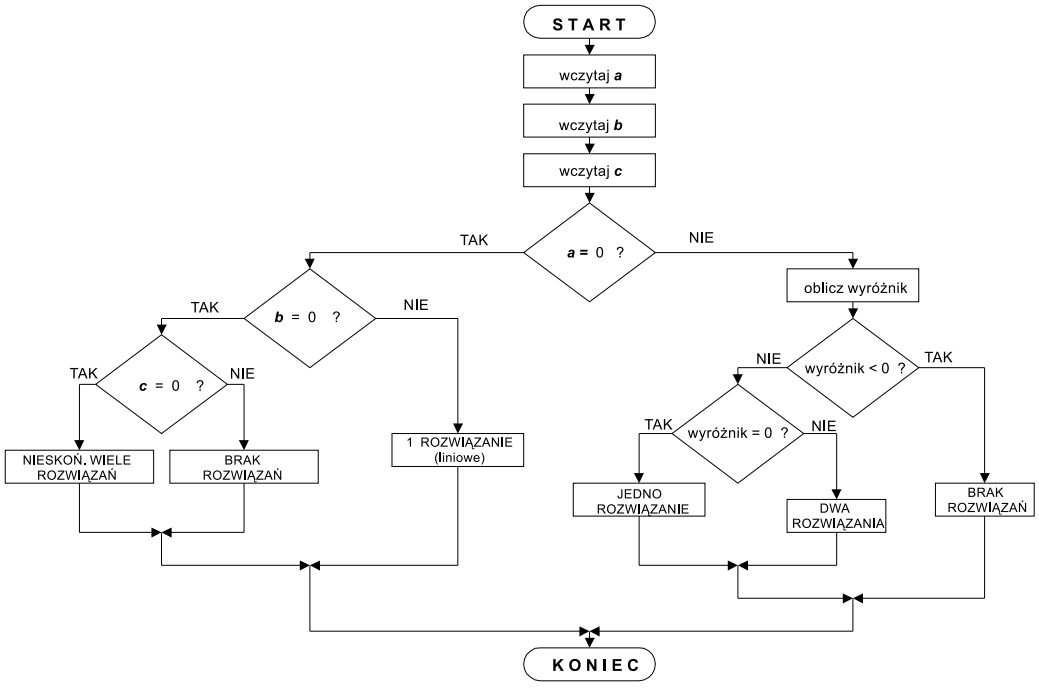
Początkujący programista zadowolony jest z tego, że ma schemat, który przedstawia poprawny algorytm realizujący podane wcześniej zadanie — dla poszczególnych przypadków uruchamiając program, wodząc palcem po schemacie, otrzymamy pozornie prawidłowe rozwiązania. Dlaczego pozorne?

Jedno rozwiązanie w przypadku gdy równość redukuje się do równania liniowego jest czymś innym, niż jedno rozwiązanie w przypadku trójmianu kwadratowego (autor schematu dokonał nieprawidłowego uogólnienia). Powody dla których równanie nie ma rozwiązań też są różne, gdy mamy lub nie mamy do czynienia z trójmianem. Algorytm byłby niepoprawny, gdybyśmy chcieli wyświetlać także wartości rozwiązań równania.

Czy schemat ten można zapisać w sposób bardziej przejrzysty? Otóż jeśli zastosujemy konwencję oznaczania *wyboru* podkreślając nie tylko początek wyboru (pytanie, od odpowiedzi na które zależy którą ścieżką powędrujemy dalej), ale także koniec — miejsce od którego odpowiedź na ostatnio zadane pytanie nie ma już znaczenia. Zyskamy wówczas nie tylko na czytelności schematu, ale również uzyskamy dodatkową informację o sposobie zagnieżdżenia poszczególnych warunków, co w szczególności dla początkujących programistów wydaje się być dość cenne. Porównajmy fragment schematu ze strony 6 z fragmentem narysowanym z uwzględnieniej wyżej opisanej konwencji.

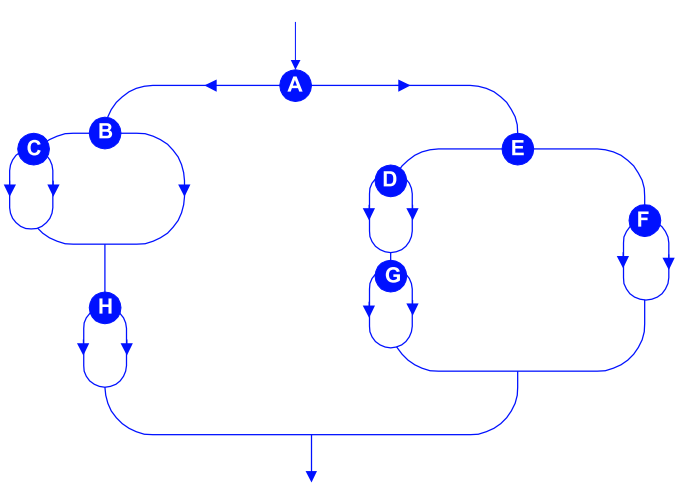


Pozornie nie zmieniło się zbyt wiele, ale na rysunku *b)* widać, że warunek sprawdzający, czy wyróżnik jest równy zero, jest **zagnieżdżony** w warunku, gdzie na pytanie czy wyróżnik jest mniejszy od zera uzyskaliśmy odpowiedź NIE. Na którym schemacie (tym ze strony 6 czy poniższym) łatwiej doliczyć, ile warunków jest zagnieżdżonych jednokrotnie, dwukrotnie czy trzykrotnie?



Podsumowując, każdy schemat blokowy, którego przebieg sterowania będzie przypominał *sieć równolegle i szeregowo połączonych oporników* da się w prosty sposób

przenieść z postaci projektu programu do postaci kodu, w którym jako instrukcji sterującej używa się wyłącznie instrukcji warunkowej (patrz przykład poniżej).




```
if A then
  do
    if B then
      do
        if C then
          [ ]
      end;
    else
      [ ]
    if H then
      [ ]
  end;
else
  if E then
    do
      if D then
        [ ]
      else
        [ ]
      if G then
        [ ]
    end;
  else if F then
         [ ]
```

Czy da się w powyższy sposób zapisać schemat ze strony szóstej?

2.4 Petle, czyli powtórki nie tylko z rozrywki...

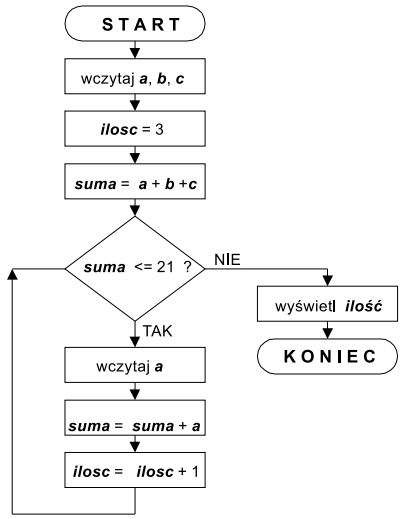
Napisaliśmy, że w programach sekwencyjnych każda instrukcja wykonywana jest dokładnie raz. A gdybyśmy chcieli wykonać pewne instrukcje kilka razy albo wcale? A nawet gorzej, jeśli nie wiemy ile razy mają być wykonane wybrane instrukcje, choć wiemy, że mają być wykonane przynajmniej raz... Rozważmy zadanie, którego pierwowzorem była karciana gra w oczko.

- **zadanie:**
  Wczytaj trzy liczby. Następnie wczytuj liczby dopóki suma liczb wczytanych nie przekroczy 21. Wówczas wyświetl ile liczb wczytałeś.

- **algorytm:**

  1. wczytaj pierwszą liczbę;
  2. wczytaj drugą liczbę;
  3. wczytaj trzecią liczbę;
  4. zapamiętaj, że wczytałeś trzy liczby;
  5. oblicz sumę wczytanych liczb;
  6. jeśli suma jest większa od 21 idź do punktu 11, w przeciwnym razie do punktu 7;
  7. wczytaj od użytkownika liczbę;
  8. zwiększ ilość liczb;
  9. dodaj liczbę do sumy;
  10. idź do punktu 6;
  11. wyświetl ile wczytano liczb;

- **schemat blokowy:**

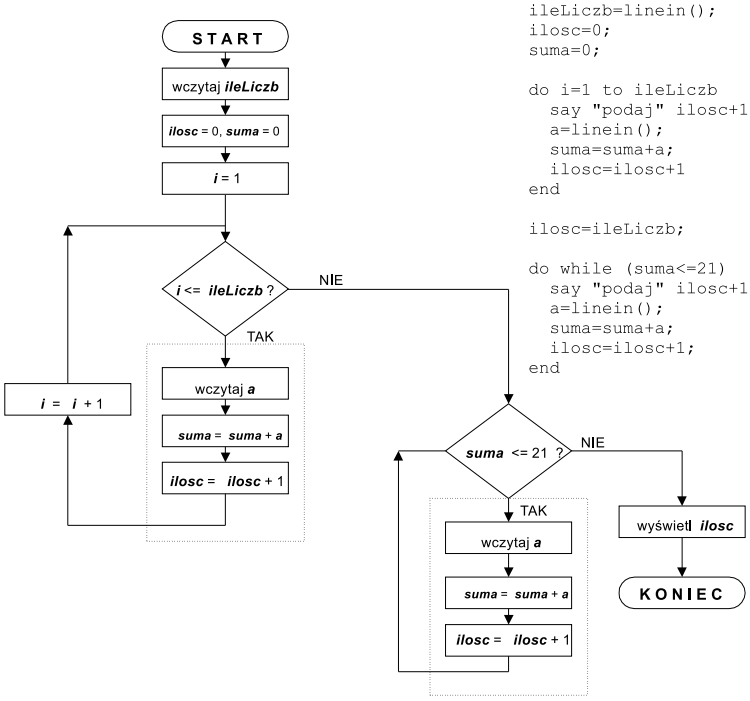


A gdybyśmy chcieli wczytywać obowiązkowo nie trzy lecz 4, 5 lub 8 liczb, albo tyle ile użytkownik podczas działania programu sobie zażyczy? Zmodyfikujmy więc zadanie.

- **algorytm:**

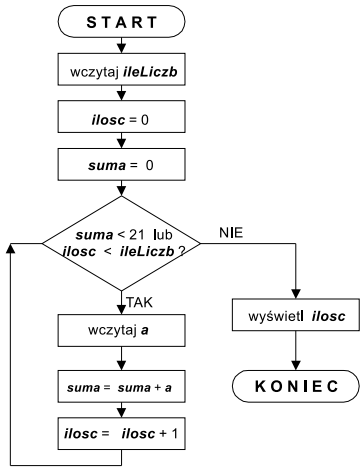
1. wczytaj ile najmniej liczb musi być wczytanych (*ileLiczb*);
2. suma wczytanych liczb początkowo wynosi 0;
3. ilość wczytanych liczb początkowo wynosi 0;
4. powtórz odpowiednią ilość razy punkty 5–7;
5. wczytaj liczbę;
6. dodaj liczbę do sumy;
7. zwiększ ilość liczb o jeden;
8. jeśli suma jest większa od 21 idź do punktu 13, w przeciwnym razie do punktu 9;
9. wczytaj liczbę;
10. dodaj liczbę do sumy;
11. zwiększ ilość liczb o jeden;
12. idź do punktu 8;
13. wyświetl ile wczytano liczb;

- **schemat blokowy:**




```
say "podaj minimalną ilość liczb:";
ileLiczb=linein();
ilosc=0;
suma=0;

do i=1 to ileLiczb
  say "podaj" ilosc+1 "liczbę";
  a=linein();
  suma=suma+a;
  ilosc=ilosc+1
end

ilosc=ileLiczb;

do while (suma<=21)
  say "podaj" ilosc+1 "liczbę";
  a=linein();
  suma=suma+a;
  ilosc=ilosc+1;
end
```

Jak dobrze się przyjrzymy powyższemu schematowi, to okaże się, że nie jest to jeszcze najpiękniej napisany algorytm realizujący postawione przed nami zadanie. Rysowanie schematów początkujących programistom bardzo często pozwala szybciej dostrzec, że pewne bloki się powtarzają. Rozwiązanie powyżej jest prawidłowe, ale nieeleganckie ze względu na niepotrzebne powielenie instrukcji: *wczytaj a; suma=suma+1; ilosc=ilosc+1*. Poniżej przedstawiono schemat, który przedstawia najkrótszą i najbardziej czytelną wersję algorytmu realizującego zmodyfikowaną wersję gry w oczko.

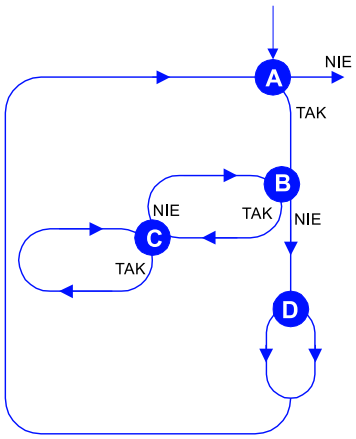


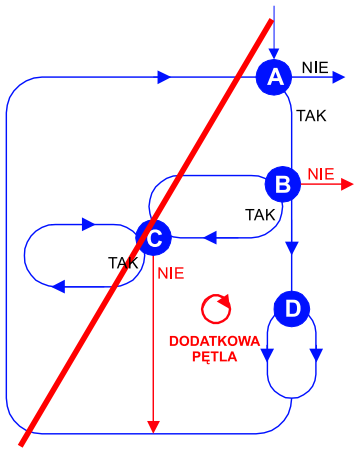

```
say "podaj minimalną ilość liczb:";
ileLiczb=linein();

ilosc=0;

suma=0;


do while (suma<21) | (ilosc<ileLiczb)


  say "podaj liczbę";

  a=linein();

  suma=suma+a;

  ilosc=ilosc+1;
end;

say "wczytalem" ilosc "liczb";
```

Powtórzenia można zagnieżdżać. Oznacza to jeszcze jedną okazję do tworzenia schematów, które niekoniecznie dadzą się bezpośrednio zapisać w postaci kodu. Jest jednak bardzo łatwy sposób na wykrycie nieprawidłowo zagnieżdżanych *powtórzeń*.

Poniżej przedstawiono schematycznie poprawne i nieprawidłowe zagnieżdżanie pętli. Jeśli spojrzymy na schemat jak na graf, to dostrzeżemy, że złe zagnieżdżenie powoduje automatyczne utworzenie się dodatkowej pętli.

```
do while A
  do while B
    do while C
      [ ]
    end;
  end;
  if D then
    [ ]
  else
    [ ]
end;
```





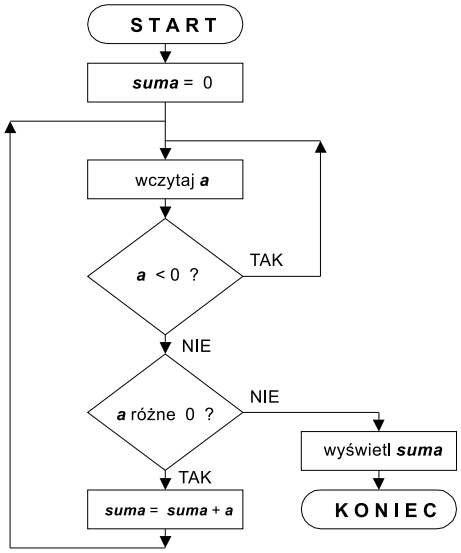
2.5 Zapętlone warunki — warunkowe pętle...

Mamy już nienajgorszy arsenał narzędzi do pisania programu, możemy wykonywać pewne instrukcje, nie wykonywać innych lub powtarzać określoną lub nieznaną liczbę razy jeszcze innych instrukcje. Możliwości do napisania schematu, na którego podstawie nie da się w prosty sposób napisać kodu programu, jest aż nadto. Czy są jakieś metody poprawiania schematu?

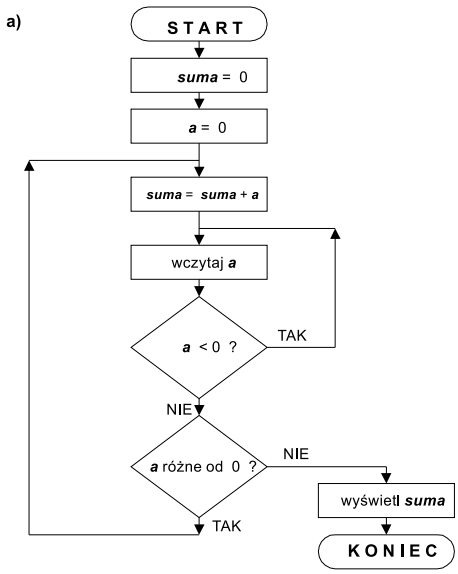
Prześledzmy to na schematach realizujących poniższe zadanie.

- **zadanie:**
  Wczytuj liczby tak długo, dopóki nie wczytasz liczby 0. Następnie wyświetl sumę wczytanych dotychczas liczb dodatnich.

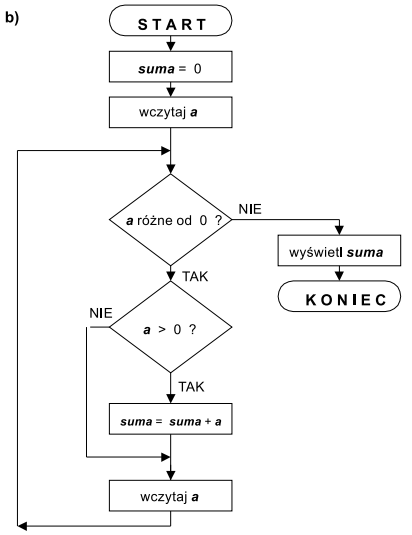
- **schemat blokowy:**



Powyższego schematu nie da się napisać bez użycia instrukcji *goto*, ze względu na to, że zewnętrzna pętla ani się nie zaczyna, ani nie kończy warunkiem. Czy to oznacza, że schemat jest zupełnie do wyrzucenia? Na szczęście nie. Jeśli mamy schemat i już wiemy, że bez instrukcji *goto*, póki co, nie da się nic zrobić, trzeba się przyjrzeć możliwościom przesunięć pewnych instrukcji zgodnie z zaznaczonym na schemacie kierunkiem sterowania w inne miejsca (patrz schemat poniżej *a)* porównaj szczegóły z powyższym schematem).



Druga i łatwiejsza metoda wypływa z następującego faktu. Niepoprawny schemat oznacza najczęściej nieprawidłową kolejność warunków, nieprawidłową kolejność zagnieżdżania pętli w warunkach, czy warunków w pętlach. Należy więc zastanowić się, czy być może ma sens zamiana sprawdzanych po kolei warunków; może warto któryś z warunków zanegować, a następnie zobaczyć, czy poprawiło to czytelność schematu, czy nie. I tak dochodzimy do schematu *b)*, który jest nie tylko poprawny.




```
suma=0;
say "podaj liczbe (0 koniec)";
a=linein();

do while a<>0

    if a>0 then

      suma=suma+a;

    say "podaj liczbe (0 koniec)";
    a=linein();
end;

say suma;
```

3 Warto czy nie warto? — Podsumowanie

Wróćmy więc do pytania postawionego na początku. Opracowanie to miało na celu przedstawienie schematów blokowych, jako zgrabnego i taniego narzędzia (wystarczy papier i ołówek) nauki programowania dla bardzo początkujących — nauki myślenia o tworzonym algorytmie, a nie nauki pisania kodu. Następujące cechy schematów powodują, że warto się nimi zająć na początku drogi programistycznej, nawet jeśli zęby same się zaciskają:

- **przepływ sterowania**
  graficzne przedstawienie algorytmu pozwala początkującemu na przyjrzenie się przebiegowi sterowania w programie i dokonanie oceny poprawności uruchamiając algorytm palcem na kartce (warte przemyślenia: Turing nie miał komputera na biurku, a jednak parę rzeczy wymyślił);

  Schematy zmuszają początkującego do prawdziwego zrozumienia elementów, które czynią programowanie użytecznym — sprawdzanie warunków i pętle. Zrozumienie to jest niezbędne w przypadku gdy mamy możliwość zagnieżdżania tych elementów. Zastosowanie opisanych wcześniej konwencji rysowania schematu pozwala uniknąć kłopotów związanych z pisaniem kodu z zagnieżdzonymi warunkami czy pętlami (postawić *do* czy nie?, do czego jest ten *end*?);

- **analogowy kompilator**
  jeżeli schemat nie daje się przedstawić w postaci planarnego grafu, gdzie jakieś strzałki wciąż się krzyżują, pomimo pomysłowego przestawiania elementów składowych; to bez czytania treści umieszczonej na schemacie wiadomo, że algorytm ma niewłaściwą kolejność wykonywanych czynności i należy się nad tą kolejnością pochylić i ją przemyśleć;

- **szczegółowość**
  schematy można stosować na różnych poziomach szczegółowości;

- **natura cyfrowego świata**
  schematy (różnego typu i z różnymi zasadami ich tworzenia) są jednym z podstawowych elementów opisu informatycznego świata, poza wzorami;

- **rysunek**
  schemat to wciąż rysunek a powiadają, że „*dobry rysunek wart jest tysiąc słów*“

Do wad schematów blokowych w przedstawionej postaci należy zaliczyć fakt, że odzwierciedlają one tylko przebieg sterowania w algorytmie, natomiast nie opisują struktury wykorzystywanych danych. Nie byłoby etycznym jednak przysypać początkującego podwójnym ciężarem... wszystko w swoim czasie! Na świecie jest wiele pożytecznych notacji, które tej wady nie mają przy zachowaniu wszelkich zalet schematów blokowych.

Schematy są stare prawie jak świat, a może nawet i starsze, uwzględniając fakt, że w informatyce wszystko co ma więcej niż dwa lata to przeżytek. Warto jednak zajrzeć do księgarni i obejrzeć instrukcję obsługi dzieła autorytetu — w świecie, w którym bark autorytetów daje nam się we znaki — Donalda E. Knuth'a pt. „*Sztuka programowania*“. A jakże... jest i algorytm i schemat!