4
Iteracje

Gdyby programy zawierały tylko proste instrukcje i rozgałęzienia (instrukcje **if** i **switch**), to wykonanie każdego programu kończyłoby się po wykonaniu skończenie wielu instrukcji.

Co więcej, maksymalna liczba instrukcji jakie mógłby wykonać program nie zależałaby zupełnie od danych do niego wprowadzonych.

Rozpatrzmy następujący program, który czyta liczbę n, następnie wczytuje n wartości liczbowych $x_1, x_2, \ldots, x_n$, liczy sumę $\sum_{i=1}^{n} x_i$ i na koniec drukuje wartość wyliczonej sumy.

Nie byłoby możliwe stworzenie takiego programu jedynie z instrukcji prostych i instrukcji rozgałęzień (instrukcje **if** i **switch**). W programie takim liczba wykonywanych instrukcji dosumowujących kolejne wartości byłaby ograniczona długością programu a nie wartością zadanego n.

Wynika z tego konieczność cyklicznego powtarzania pewnych fragmentów programu. W poprzednim rozdziale przedstawiono jak użyć do tego instrukcję skoku. W tym rozdziale zaprezentowane zostaną dostępne w języku C strukturalne instrukcje iteracji.

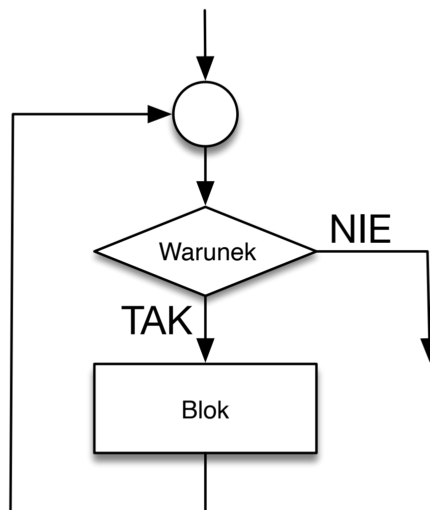
4.1 Instrukcja while

Podstawową instrukcją iteracji jest instrukcja **while**. Ma ona w języku C następującą postać.

```
while(Warunek)
    Blok;
```

Powtarza ona wykonywanie zawartego w niej bloku instrukcji tak długo, dopóki spełniony jest zadany warunek (patrz schemat blokowy przedstawiony na rysunku 4.1).

DRAFT



Rysunek 4.1: Schemat działania instrukcji **while**.

DRAFT

4.1.1 Uzasadnianie poprawności pętli

Rozpatrzmy następującą pętlę:

```
while(B) S;
```

Ma ona tę własność, że dla pewnego warunku logicznego `P` zachodzi:

```
assert(P && B);
S;
assert(P);
```

Jeśli warunek `P` ma taką własność, to dla warunku `P` zachodzi również:

```
assert(P);
while(B) S;
assert(P && !B);
```

Taki warunek zawsze istnieje (np. **true**) ale nie każdy taki warunek jest jednakowo przydatny do opisu działania pętli. Powinniśmy poszukiwać warunku P, który jest spośród wszystkich takich warunków jest najsilniejszy[1], bo tylko taki faktycznie mówi nam coś o analizowanej pętli.

Najsilniejszy warunek `P` zwykło nazywać się *niezmiennikiem pętli*, ponieważ jest on nadal spełniony po każdym przebiegu treści pętli.

Niezmiennika pętli używa się w dowodach poprawności programu.

Warunek P zależy od bieżących wartości zmiennych. Aby to podkreślić będziemy zapisywać go jako $P(x_1, x_2, \ldots, x_n)$, gdzie $x_1, x_2, \ldots, x_n$ są zmiennymi występującymi w programie od których wartości zależy warunek P. Często aby

[1] Mówimy, że warunek P jest silniejszy od warunku Q, jeśli wszystkie dane spełniające warunek P spełniają również warunek Q ale pewne dane spełniające warunek Q nie spełniają warunku P. Zgodnie z tą definicją warunek **true** jest najsłabszy, bo każde dane go spełniają.

uprościć zapis będziemy uwzględniać w wykazie zmiennych tylko te zmienne, które zmieniają swoją wartość podczas wykonywania pętli.

Dowód tego, że warunek $P(x_1, x_2, \ldots, x_n)$ jest niezmiennikiem pętli `while(B) S`, przeprowadza się indukcyjnie. Musimy wykonać dwie rzeczy:

1. Sprawdzić, że kiedy program dochodzi do pętli, to wartości zmiennych $x_1, x_2, \ldots, x_n$ spełniają warunek niezmiennika $P(x_1, x_2, \ldots, x_n)$ (krok pierwszy dowodu indukcyjnego).

2. Udowodnić, że jeśli zmienne $x_1, x_2, \ldots, x_n$ spełniają warunek niezmiennika $P(x_1, x_2, \ldots, x_n)$ i spełniony jest warunek sterujący pętlą B, to nowe wartości zmiennych $x'_1, x'_2, \ldots, x'_n$, jakie przyjęły zmienne $x_1, x_2, \ldots, x_n$ w wyniku wykonania bloku instrukcji `S`, również spełniają warunek niezmiennika $P(x'_1, x'_2, \ldots, x'_n)$ (krok drugi dowodu indukcyjnego). Primami rozróżniamy nowe wartości zmiennych po wykonaniu bloku `S` od starych przed wykonaniem tego bloku.

Dowód jest indukcyjny po wykonywanych iteracjach pętli, zatem warunek $P(x_1, x_2, \ldots, x_n)$, warunek B i zależności między starymi wartościami $x_1, x_2, \ldots, x_n$ a nowymi $x'_1, x'_2, \ldots, x'_n$ stanowią założenie indukcyjne, natomiast warunek $P(x'_1, x'_2, \ldots, x'_n)$ stanowi tezę indukcyjną.

Na wybranych przykładach zaprezentujemy teraz użycie niezmiennika w dowodzeniu poprawności programu.

DRAFT

Iloraz i reszta z dzielenia

Rozpoczniemy od przykładu z książki [1]. Na listingu 4.1 przedstawiono program wyliczający iloraz i resztę z dzielenia całkowitego.

Listing 4.1: Iloraz i reszta z dzielenia całkowitego.

```c
// niezmiennik.c

#include <stdio.h>
#include <assert.h>

int main(void)
{
  int x;
  printf("         Podaj liczbę naturalną: ");
  scanf("%d", &x);
  int y;
  printf("Podaj liczbę całkowitą dodatnią: ");
  scanf("%d", &y);
  assert(x >= 0 && y > 0);
  int q = 0;
  int r = x;
  assert(x == q * y + r && 0 <= r);
  while(r >= y)
  {
    assert(x == q * y + r && 0 < y && y <= r);
    r = r - y;
    q = q + 1;
```

```
    }
    assert(x == q * y + r && 0 <= r && r < y);
    printf("%d div %d = %d\n", x, y, q);
    printf("%d mod %d = %d\n", x, y, r);
    printf("test: %d * %d + %d = %d\n", q, y, r, q * y + r);
    return 0;
}
```

Niezmiennikiem pętli **while** jest:

$$P(q, r) \equiv x = q \cdot y + r \wedge 0 \leq r.$$

Dowód, że powyższy warunek jest niezmiennikiem przeprowadzimy indukcyjnie.

W kroku pierwszym sprawdzimy, czy warunek jest spełniony gdy dochodzimy do pętli. Zmienne q i r posiadają wówczas następujące wartości: $q = 0, r = x$. Podstawiając je w formule $P(q, r)$ otrzymujemy:

$$P(0, x) \equiv x = 0 \cdot y + x \wedge 0 \leq 0,$$

co jest spełnione.

W kroku drugim pokażemy, następującą implikację:

$$P(q, r) \wedge r \geq y \rightarrow P(q', r'),$$

gdzie q' i r' są nowymi wartościami zmiennych q i r, które zostały wyliczone w kolejnym przebiegu pętli, gdy warunek nią sterujący $r \geq y$ był spełniony.

Wypiszmy przesłanki implikacji, którą będziemy dowodzić:

$$x = q \cdot y + r, \tag{4.1}$$
$$0 \leq r, \tag{4.2}$$
$$r \geq y. \tag{4.3}$$

Dopiszemy do nich zależności między starymi wartościami zmiennych q i r a nowymi wyliczonymi w jednym przebiegu pętli:

$$r' = r - y, \tag{4.4}$$
$$q' = q + 1. \tag{4.5}$$

Zauważmy, że

$$\begin{aligned} x &= q \cdot y + r && \text{z (4.1)} \\ &= (q' - 1) \cdot y + r && \text{z (4.5)} \\ &= (q' - 1) \cdot y + (r' + y) && \text{z (4.4)} \\ &= q' \cdot y + r' \end{aligned}$$

$$\begin{aligned} r' &= r - y && \text{z (4.4)} \\ &\geq 0 && \text{z (4.3)} \end{aligned}$$

Zatem nowe wartości q' i r' również spełniają warunek $P(q', r')$. Co kończy dowód indukcyjny.

Po wyjściu z pętli zachodzi warunek $P(q, r) \wedge r < y$, zatem

$$x = q \cdot y + r \wedge 0 \leq r < y,$$

czyli q jest całkowitym ilorazem x przez y a r jest resztą z dzielenia x przez y.

DRAFT

Potęgowanie

Na listingu 4.2 przedstawiono program podnoszący wartość x do potęgi n. W komentarzu i na wydruku użyto symbolu ** do oznaczenia operacji potęgowania.

Listing 4.2: Podnoszenie do potęgi.

```
#include <stdio.h>
#include <assert.h>

int main(void)
{
  int x;
  printf(" podaj dodatnie x: ");
  scanf("%d", &x);
  assert(x > 0);
  int n;
  printf("podaj nieujemne n: ");
  scanf("%d", &n);
  assert(n >= 0);

  int y = 1;
  int p = x;
  int i = n;
  while(i > 0)
  {
    // niezmiennik: y * p ** i = x ** n
    if(i % 2 == 0)
    {
      p = p * p;
      i = i / 2;
    }
    else
    {
      y = y * p;
      i = i - 1;
    }
  }

  printf("%d ** %d = %d\n", x, n, y);
  return 0;
}
```

DRAFT

Niezmiennikiem pętli **while** jest warunek $P(y, p, i) \equiv y \cdot p^i = x^n$.

Gdy dochodzimy do pętli, wówczas $y = 1, p = x, i = n$, zatem warunek $P(y, p, i) \equiv 1 \cdot x^n = x^n$ jest spełniony.

Udowodnimy teraz, że z założeń:

$$y \cdot p^i = x^n, \quad (4.6)$$

$$i > 0, \quad (4.7)$$

wynika teza:

$$y' \cdot p'^{i'} = x^n.$$

Zależność między nowymi wartościami y', p', i' a starymi y, p, i jest teraz trochę bardziej skomplikowana, bo zależy od tego czy i jest wartością parzystą czy nie.

Dowód zatem będzie składał się z dwóch przypadków.

Gdy i jest liczbą parzystą, wówczas:

$$y' = y, \tag{4.8}$$
$$p' = p^2, \tag{4.9}$$
$$i' = \frac{i}{2}. \tag{4.10}$$

Policzmy teraz ile wynosi $y' \cdot p'^{i'}$:

$$\begin{aligned} y' \cdot p'^{i'} &= y \cdot p'^{i'} && \text{z (4.8)} \\ &= y \cdot (p^2)^{i'} && \text{z (4.9)} \\ &= y \cdot (p^2)^{\frac{i}{2}} && \text{z (4.10)} \\ &= y \cdot p^{2 \cdot \frac{i}{2}} \\ &= y \cdot p^i \\ &= x^n && \text{z (4.6)} \end{aligned}$$

W tym przypadku teza została udowodniona.

Pozostał do udowodnienia drugi przypadek, w którym i jest liczbą nieparzystą. Wówczas:

DRAFT

$$y' = y \cdot p, \tag{4.11}$$
$$p' = p, \tag{4.12}$$
$$i' = i - 1. \tag{4.13}$$

Policzmy teraz ile wynosi $y' \cdot p'^{i'}$:

$$\begin{aligned} y' \cdot p'^{i'} &= y \cdot p \cdot p'^{i'} && \text{z (4.11)} \\ &= y \cdot p \cdot p^{i'} && \text{z (4.12)} \\ &= y \cdot p \cdot p^{i-1} && \text{z (4.13)} \\ &= y \cdot p^i \\ &= x^n && \text{z (4.6)} \end{aligned}$$

Jak widać, w tym przypadku teza również została udowodniona, zatem warunek $P(y, p, i) \equiv y \cdot p^i = x^n$ jest faktycznie niezmiennikiem pętli.

Gdy program opuszcza pętlę **while** zmienna i ma wartość równą 0, ponieważ jest to pierwsza wartość całkowita niespełniająca warunku sterującego pętlą $i > 0$ (nie mogliśmy wartości 0 „przeskoczyć” bo za każdym razem albo dzieliliśmy parzystą liczbę przez dwa albo odejmowaliśmy od nieparzystej liczby jedynkę).

Po wyjściu z pętli zachodzi warunek $P(y, p, i) \wedge i = 0$, czyli:

$$y \cdot p^i = y \cdot p^0 = y \cdot 1 = y = x^n.$$

Zatem program drukując wartość zmiennej y, wydrukuje obliczoną potęgę x^n.

Rozszerzony algorytm Euklidesa

Teraz trochę bardziej złożony przykład. Na listingu 4.3 przedstawiono rozszerzony algorytm Euklidesa. Służy on do rozwiązywania równania diofantycznego

(wielomian ze współczynnikami całkowitymi i zmiennymi o wartościach całkowitych):

$$m \cdot x + n \cdot y = z, \tag{4.14}$$

gdzie stałe m stałe n są liczbami całkowitymi, natomiast x, y i z są poszukiwanymi wartościami całkowitymi spełniającymi to równanie.

Listing 4.3: Rozszerzony algorytm Euklidesa

```
// rozeuklides.c

// Program przedstawia wartość największego wspólego dzielnika
// dwóch liczb w postaci ich liniowej kombinacji. Wykorzystano
// rozszerzony algorytm Euklidesa.

#include <stdio.h>
#include <assert.h>

int main(void)
{
  int m, n;
  printf("Podaj pierwszą liczbę dodatnią: ");
  scanf("%d", &m);
  assert(m > 0);
  printf("   Podaj drugą liczbę dodatnią: ");
  scanf("%d", &n);
  assert(n > 0);

  int a = m;
  int b = n;
  int x = 1;
  int y = 0;
  int r = 0;
  int s = 1;
  while (b > 0)
  {
    // niezmiennik: NWD(m, n) = NWD(a, b)
    // niezmiennik: a = m * x + n * y
    // niezmiennik: b = m * r + n * s
    int reszta = a % b;
    int iloraz = a / b;
    a = b;
    b = reszta;
    int rr = r;
    int ss = s;
    r = x - iloraz * r;
    s = y - iloraz * s;
    x = rr;
    y = ss;
  }
  int z = a;

  printf("%d * %d + %d * %d = %d\n", m, x, n, y, z);
  return 0;
}
```

DRAFT

Aby równanie (4.14) miało rozwiązanie, wartość z musi być równa całkowitej wielokrotności największego wspólnego dzielnika liczb m i n. Program wylicza $z = NWD(m,n)$ i przy okazji znajduje szukane wartości x i y spełniające równanie (inne rozwiązania są wielokrotnościami tych wartości).

Pokażemy, że

$$P(a,b,x,y,r,s) \equiv NWD(m,n) = NWD(a,b) \wedge a = m \cdot x + n \cdot y \wedge b = m \cdot r + n \cdot s,$$

jest niezmiennikiem pętli **while**.

Gdy program dochodzi do pętli, wówczas $a = m, b = n, x = 1, y = 0, r = 0, s = 1$, zatem warunek niezmiennika zachodzi:

$$P(m,n,1,0,0,1) \equiv NWD(m,n) = NWD(m,n) \wedge m = m \cdot 1 + n \cdot 0 \wedge n = m \cdot 0 + n \cdot 1.$$

W ten sposób udowodniliśmy pierwszy krok indukcyjny.

W drugim kroku indukcyjnym pokażemy, że z następujących założeń (warunek niezmiennika na starych wartościach zmiennych, warunek sterujący i zależności nowych wartości zmiennych od starych):

$$\begin{aligned}
NWD(m,n) &= NWD(a,b), && (4.15)\\
a &= m \cdot x + n \cdot y, && (4.16)\\
b &= m \cdot r + n \cdot s, && (4.17)\\
b &> 0, && (4.18)\\
a' &= b, && (4.19)\\
b' &= a \mod b, && (4.20)\\
r' &= x - \lfloor a/b \rfloor \cdot r, && (4.21)\\
s' &= y - \lfloor a/b \rfloor \cdot s, && (4.22)\\
x' &= r, && (4.23)\\
y' &= s, && (4.24)
\end{aligned}$$

DRAFT

wynikają warunki niezmiennika na nowych wartościach zmiennych:

$$\begin{aligned}
NWD(m,n) &= NWD(a',b'), && (4.25)\\
a' &= m \cdot x' + n \cdot y', && (4.26)\\
b' &= m \cdot r' + n \cdot s'. && (4.27)
\end{aligned}$$

W dowodzie skorzystamy z następujących czterech własności, ktrych proste dowody pozostawiam Państwu jako ćwiczenie:

$$\begin{aligned}
x \mod b &= x, \quad \text{dla dowolnego } 0 \le x < b, && (4.28)\\
NWD(b, a \mod b) &= NWD(a,b), && (4.29)\\
m \cdot r' + n \cdot s' &< b, && (4.30)\\
(x+y) \mod z &= (x \mod z + y \mod z) \mod z. && (4.31)
\end{aligned}$$

Dowiedziemy teraz trzy równania składające się na tezę indukcyjną:

$$
\begin{aligned}
NWD(a',b') &= NWD(b, a \mod b) && \text{z (4.19) i (4.20)}\\
&= NWD(a,b) && \text{z (4.29)}\\
&= NWD(m,n) && \text{z (4.15)}
\end{aligned}
$$

$$
\begin{aligned}
a' &= b && \text{z (4.19)}\\
&= m \cdot r + n \cdot s && \text{z (4.17)}\\
&= m \cdot x' + n \cdot y' && \text{z (4.23) i (4.24)}
\end{aligned}
$$

$$
\begin{aligned}
b' &= a \mod b && \text{z (4.20)}\\
&= (m \cdot x + n \cdot y) \mod b && \text{z (4.16)}\\
&= (m \cdot (r' + \lfloor a/b \rfloor \cdot r) + n \cdot (s' + \lfloor a/b \rfloor \cdot s)) \mod b\\
&= (m \cdot r' + n \cdot s' + \lfloor a/b \rfloor \cdot (m \cdot r + n \cdot s)) \mod b\\
&= (m \cdot r' + n \cdot s' + \lfloor a/b \rfloor \cdot b) \mod b\\
&= ((m \cdot r' + n \cdot s') \mod b + 0) \mod b && \text{z (4.31)}\\
&= ((m \cdot r' + n \cdot s') \mod b) \mod b\\
&= (m \cdot r' + n \cdot s') \mod b && \text{z (4.28)}\\
&= m \cdot r' + n \cdot s' && \text{z (4.30) i (4.28)}
\end{aligned}
$$

Gdy opuszczamy pętlę **while** spełniony jest warunek $P(a,b,x,y,r,s) \wedge b = 0$. Gdy podczas wykonywania pętli, po raz ostatni b było większe od 0, to musiało być równe a (bo $b' = a \mod b = 0$). Z tego, że $NWD(m,n) = NWD(a,b) = NWD(a,a) = a$ wynika, że $z = NWD(m,n) = a$.

Z kolei ze spełnionego równania $a = m \cdot x + n \cdot y$ wynika, że wyliczone wartości x i y, wraz z $z = a$, są rozwiązaniami równania (4.14).

DRAFT

4.2 Instrukcja for

Często mamy do czynienia z sytuacją, w której pewien fragment kodu należy wykonać dla pewnego ciągu wartości $(a_j)_{j=1}^{n}$. W takiej sytuacji przydatna jest instrukcja **for**.

W instrukcji tej, kolejne wartości ciągu przechowywane są w tzw. *zmiennej sterującej pętlą*.

Załóżmy, że rozpatrywanym ciągiem jest ciąg arytmetyczny

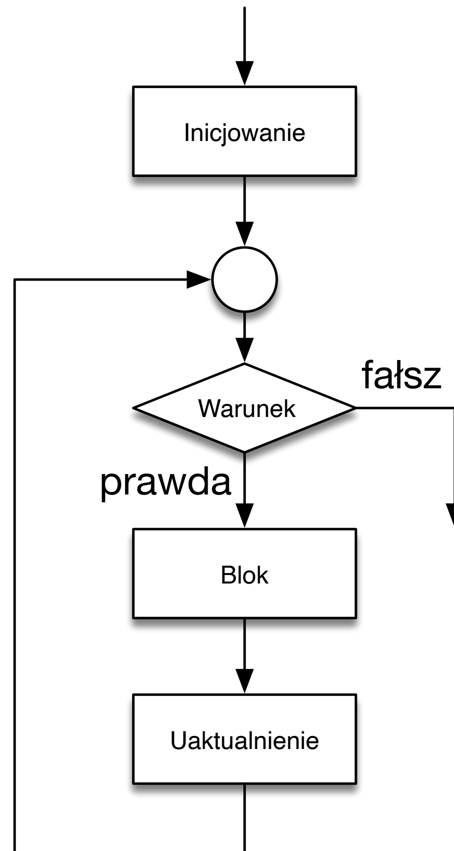
$$(a_j) = (0, 1, 2, \ldots, n-1). \tag{4.32}$$

Jego kolejne wyrazy możemy wyliczyć ze wzoru:

$$
\begin{cases}
a_0 &= 0\\
a_{j+1} &= a_j + 1
\end{cases}
$$

Jeśli i jest zmienną sterującą pętlą, to dla powyższego ciągu powinna przyjąć ona początkowo wartość 0 a następnie po każdej iteracji powinna ona rosnąć o 1.

Wykonywanie pętli dla ciągu (4.32) powinno zakończyć się gdy i przekroczy wartość $n-1$, zatem powtarzanie jej powinno trwać dopóki wartość zmiennej sterującej jest mniejsza od n.



Rysunek 4.2: Schemat działania instrukcji **for**.

Jak widać istotne jest odpowiednie **zainicjowanie** wartości zmiennej sterującej, odpowiednie jej **uaktualnianie** i podanie odpowiedniego **warunku**, mówiącego jak długo ma się pętla powtarzać.

W języku C instrukcja pętli **for** ma następującą postać:

```
for(Inicjowanie; Warunek; Uaktualnienie)
    Blok;
```

Schemat jej działania przedstawiono na rysunku 4.2.

Rozpatrzmy następujący prosty przykład użycia instrukcji **for**. Drukujemy w nim kolejne potęgi liczby 2, które są mniejsze od miliarda.

```
#include <stdio.h>

int main(void)
{
  for(int i = 1; i < 1000000000; i*=2)
    printf("%d\n", i);
  return 0;
}
```

Gdy przyjrzymy się dokładniej schematowi z rysunku 4.2, zauważymy, że działanie pętli **for** odpowiada następującej pętli **while**:

```
Inicjowanie;
while(Warunek)
{
    Blok;
    Uaktualnienie;
}
```

Jednak zalecam korzystanie z instrukcji **for**, ponieważ trzy istotne elementy opisu ciągu wartości sterujących przebiegiem pętli są w niej umieszczone obok siebie w nagłówku pętli. Łatwiej jest wtedy zauważyć co to za ciąg a jest to istotne dla zrozumienia jak pętla działa.

4.2.1 Wielokrotne inicjowanie i uaktualnianie

W pętli **for** podaje się w nawiasach trzy sekcje oddzielone średnikami.

Pierwsza dotyczy inicjowania danych przed wykonaniem pętli i może zawierać wiele przypisań (ale nie deklaracji lokalnych[2]) oddzielonych przecinkami.

Podobnie trzecia sekcja, odpowiadająca za uaktualnienie danych przed kolejnym przebiegiem pętli, może zawierać wiele instrukcji przypisania i (in/de)-krementacji oddzielonych przecinkami.

Rozpatrzmy następujący przykład:

```
int i, j;
for(i = 0, j = n; i < j; i++, j--)
    printf("i = %d, j = %d\n", i, j);
```

DRAFT

Wykonując powyższą pętlę dla $n = 10$ otrzymujemy następujący wynik:

```
i = 0, j = 10
i = 1, j = 9
i = 2, j = 8
i = 3, j = 7
i = 4, j = 6
```

Pamiętając o tym, że pętla **for** jest w języku C jedynie skróconym zapisem pętli **while**, można również analizować jej działanie za pomocą niezmiennika pętli.

Niezmiennikiem powyższej pętli jest warunek:

$$P(i, j) \equiv i + j = n. \tag{4.33}$$

Udowodnimy go, korzystając z tego, że powyższa pętla odpowiada następującej pętli **while**:

```
int i, j;
i = 0;
j = n;
```

[2]To dlatego, że przypisanie wartości jest wyrażeniem i można łączyć wyrażenia operatorem przecinka, który narzuca kolejność wyliczania tych wyrażeń, natomiast deklaracje z inicjowaniem są instrukcjami i nie mogą być oddzielane operatorem przecinka.

```
while(i < j)
{
  printf("i = %d, j = %d\n", i, j);
  i = i + 1;
  j = j - 1;
}
```

W pierwszym kroku dowodu indukcyjnego sprawdzamy czy warunek niezmiennika zachodzi dla zainicjowanych danych:

$$P(i,j) \equiv P(0,n) \equiv 0+n=n \equiv n=n.$$

W drugim pokażemy prawdziwość implikacji:

$$P(i,j) \wedge i<j \implies P(i',j'),$$

gdzie i', j' są nowymi wartościami zmiennych i oraz j.

Przesłankami są warunki:

$$i+j = n, \quad (4.34)$$
$$i < j, \quad (4.35)$$
$$i' = i+1, \quad (4.36)$$
$$j' = j-1. \quad (4.37)$$

DRAFT

Tezę indukcyjną dowodzimy następująco:

$$\begin{aligned} i'+j' &= i+1+j' && \text{z (4.36)} \\ &= i+1+j-1 && \text{z (4.37)} \\ &= i+j \\ &= n && \text{z (4.34)} \end{aligned}$$

4.2.2 Nieskończona pętla

W szczególnym przypadku można opuścić wszystkie trzy sekcje oddzielone średnikami:

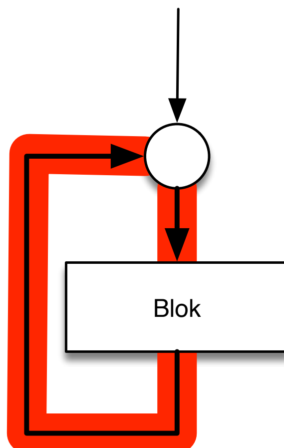
```
for(;;)
    Blok;
```

Taka instrukcja odpowiada nieskończonej pętli, której schemat blokowy przedstawiono na rysunku 4.3.

Pętle takie znajdują zastosowanie w programach, których działanie polega na nieskończonym oczekiwaniu na zdarzenia i obsługiwaniu tych zdarzeń. Przykładami takich programów są systemy operacyjne i programy w mikrokontrolerach sterujących pracą urządzeń.

4.3 Instrukcja do-while

Gdy przyjrzymy się schematowi działania pętli **while**, przedstawionemu na rysunku 4.1, zauważymy, że jeśli warunek sterujący pętlą nie jest spełniony przy dojściu do niej, to blok wewnątrz pętli nie wykona się ani razu.



Rysunek 4.3: Działanie nieskończonej pętli **for**.

Zdarza się jednak, że mamy czasami do czynienia z sytuacją, w której blok wewnątrz pętli powinien wykonać się przynajmniej raz a warunek sterujący powtarzaniem się pętli powinien być sprawdzany po a nie przed wykonaniem bloku.

Przydatna w takich sytuacjach jest pętla **do-while**:

DRAFT

```
do
   Blok
while(Warunek);
```

Schemat działania pętli **do-while** przedstawiono na rysunku 4.4.

Poniższy przykład ilustruje jej użycie. Załóżmy, że chcemy wczytać liczbę całkowitą n z zakresu od a do b. Możemy to zrobić poniższą pętlą:

```
do
{
  printf("podaj liczbę z zakresu od %d do %d: ", a, b);
  scanf("%d", &n);
} while(n < a || n > b);
```

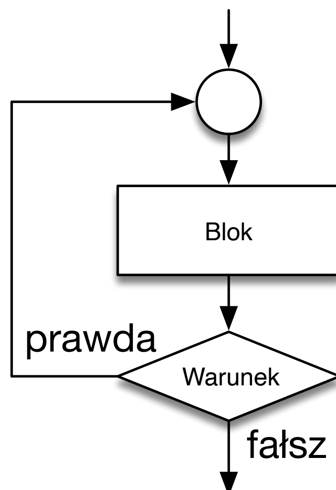
4.4 Instrukcja break

Jeśli zaistniała sytuacja, w której należy natychmiast opuścić pętlę, należy skorzystać z instrukcji **break**.

Użycie tej instrukcji pokażemy na przykładzie programu sprawdzającego czy dana liczba jest liczbą pierwszą. Jego kod przedstawiono na listingu 4.4.

Listing 4.4: Test na bycie liczbą pierwszą

```
// pierwsza.c
//
// Program sprawdza czy dana liczba jest pierwsza. Jeśli nie,
// to podaje jeden z jej właściwych dzielników.
```



Rysunek 4.4: Schemat działania pętli **do-while**.

DRAFT

```
#include <stdio.h>
#include <math.h>
#include <assert.h>

int main(void)
{
  int n;
  printf("Podaj liczbę do zbadania (większą od 1): ");
  scanf("%d", &n);
  assert(n > 1);
  int limit = sqrt(n);
  int i = 2;
  while(i <= limit)
  {
    if(n % i == 0)
      break;
    i++;
  }
  if(i > limit)
    printf("%d jest liczbą pierwszą.\n", n);
  else
    printf("%d nie jest liczbą pierwszą (np. %d|%d).\n",
           n, i, n);
  return 0;
}
```

Program czyta liczbę $n > 1$ i w pętli **while** sprawdza czy dzieli się ona przez kolejne liczby $i = 2, 3, 4, \ldots$. Sprawdzanie może zakończyć się w jednym z dwóch przypadków:

1. Wartość zmiennej i przekroczyła $\sqrt{n}$, co oznacza, że liczba n jest liczbą pierwszą. Jeśli liczba n nie ma dzielnika właściwego mniejszego lub równego $\sqrt{n}$, to również nie ma dzielnika właściwego większego od $\sqrt{n}$.

Gdyby n miało dzielnik d większy od $\sqrt{n}$, to miałoby również dzielnik mniejszy od $\sqrt{n}$ równy n/d.

2. Znaleziono wartość i nie przekraczającą $\sqrt{n}$, która jest właściwym dzielnikiem n.

Po wyjściu z pętli **while** wystarczy sprawdzić jaki był powód opuszczenia pętli **while**. Jeśli po wyjściu z pętli zmienna i ma wartość większą niż $\sqrt{n}$, to znaczy, że n jest liczbą pierwszą. W przeciwnym przypadku n nie jest liczbą pierwszą gdyż ma co najmniej jeden właściwy dzielnik (np. i).

4.5 Instrukcja continue

W pewnych sytuacjach zachodzi konieczność aby opuścić wykonywanie pozostałej części bloku instrukcji i przejść do kolejnej iteracji. Służy do tego instrukcja **continue**.

Poniższy przykład jest zbyt prosty by docenić w pełni użyteczność tej instrukcji ale ilustruje jej ideę. Program czyta kolejne liczby rzeczywiste. Kończy pracę gdy wprowadzi się liczbę zero. W każdym innym przypadku iteracje są kontynuowane ale dla liczb dodatnich drukowane są dodatkowo wartości ich pierwiastków kwadratowych.

DRAFT

```
#include <stdio.h>
#include <math.h>

int main(void)
{
  float x;
  do
  {
    scanf("%f", &x);
    if(x <= 0.0)
      continue;
    printf("%f\n", sqrt(x));
  } while(x != 0.0);
  return 0;
}
```

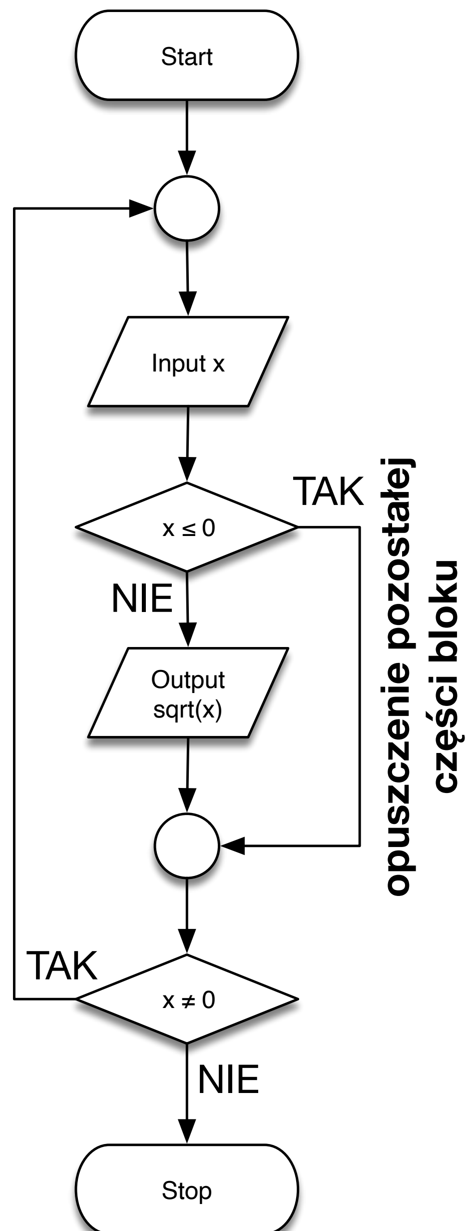
Schemat działania powyższego programu przedstawiono na rysunku 4.5.

4.6 Wybór odpowiedniej pętli

Jeśli zastanawiasz się, którą pętlę spośród dostępnych w języku C pętli **for**, **while** i **do-while** zastosować, to pomoże ci schemat przedstawiony na rysunku 4.6.

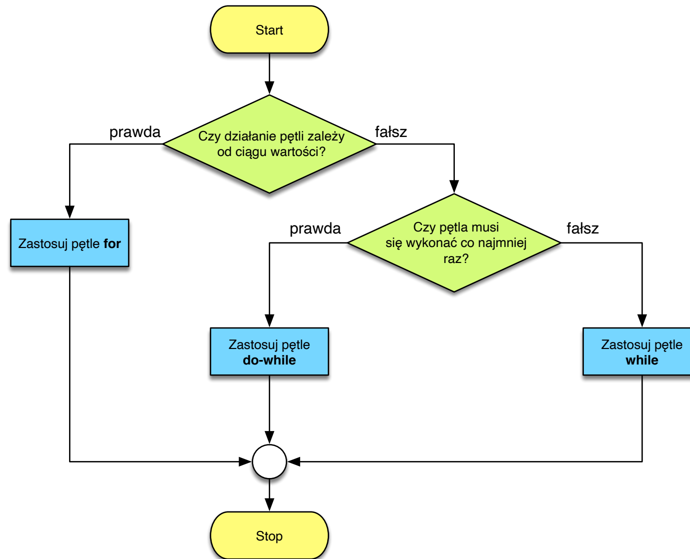
4.7 Ciekawostka

Nie zawsze instrukcja iteracji jest wykonywana przez powtarzanie fragmentu kodu. Oto prosty program, który po przeczytaniu liczby n liczy sumę liczb od 1 do n:



Rysunek 4.5: Warunkowy skok omijający część bloku instrukcji.

.



Rysunek 4.6: „System ekspercki” doradzający wybór odpowiedniej pętli.

.

DRAFT

```
// main.c

#include <stdio.h>

int main(void)
{
  int n;
  printf("Podaj liczbę n: ");
  scanf("%d", &n);
  int s = 0;
  for(int i = 1; i <= n; i++)
    s = s + i;
  printf("1 + 2 + ... + %d = %d\n", n, s);
  return 0;
}
```

Kiedy skompilujemy go z opcję `-O3` zalecającą wysoki poziom optymalizacji kodu, to instrukcja pętli zostanie zastąpiona kodem wyliczającym wartość $\frac{n\cdot(n+1)}{2}$.

Możemy przekonać się o tym podając kompilatorowi opcję `-S`, która poleca kompilatorowi zapisanie przekładu programu na język assemblera:

```
$ clang -O3 main.c -S
$
```

W powstałym w ten sposób pliku `main.s`, zawierającym przekład na język assemblera, znajdziemy następujący fragment:

```
    leal    -1(%rsi), %eax
    leal    -2(%rsi), %ecx
```

```
imulq   %rax, %rcx
shrq    %rcx
leal    -1(%rcx,%rsi,2), %ebx
```

Przeanalizujmy krok po kroku powyższy kod. Przeczytana wartość n znajduje się w rejestrze RSI. W kolejnych instrukcjach wyliczane są następujące wartości[3]:

$$
\begin{aligned}
EAX &= RSI - 1 = n - 1 \\
ECX &= RSI - 2 = n - 2 \\
RCX &= RCX \cdot RAX = (n-1)\cdot(n-2) \\
RCX &= \frac{RCX}{2} = \frac{(n-1)\cdot(n-2)}{2} \\
EBX &= RCX + 2\cdot RSI - 1 = \frac{(n-1)\cdot(n-2)}{2} + 2\cdot n - 1 = \frac{n\cdot(n+1)}{2}
\end{aligned}
$$

Jak widać po wykonaniu tego kodu, w rejestrze EBX zostaje wyliczona wartość:

$$\sum_{i=1}^{n} i = \frac{n\cdot(n+1)}{2}.$$

Jeśli kompilator zauważy, że może wykonać pętlę wyliczając wynik jej działania bez powtarzania kodu, to optymalizując kod zastąpi pętlę takim wyliczeniem.

Wybrane instrukcje assemblera i dostępne w nich tryby adresacji opisano w Dodatku A.5.

DRAFT

[3]Zapisując wartość w 32-bitowym rejestrze EAX ustala się mniej znaczącą połowę 64-bitowego rejestru RAX. Podobnie jest z rejestrami ECX i RCX