3 Rozgałęzienia

3.1 Ścieżka obliczeń

DRAFT

Przez ścieżkę obliczeń rozumieć będziemy ciąg wykonanych przez program instrukcji.

Gdyby programy składały się jedynie z instrukcji podstawienia oraz czytania i pisania, to ich wykonanie odbywałoby się zawsze tylko jedną ścieżką obliczeń.

Byłyby to jednak zbyt proste programy aby liczyć ciekawsze rzeczy.

Już przy wyliczaniu pierwiastków trójmianu kwadratowego, zachodzi konieczność by część instrukcji wykonywać tylko gdy wyróżnik trójmianu jest ujemny, część gdy jest on równy zeru a część gdy jest dodatni.

Na rysunku 3.1 przedstawiono dwa schematy blokowe. Na pierwszym jest tylko jedna możliwa ścieżka obliczeń a na drugim ścieżek takich są trzy.

3.2 Instrukcja złożona

Przez instrukcję złożoną mamy na myśli instrukcję, która zawiera w sobie blok innych instrukcji.

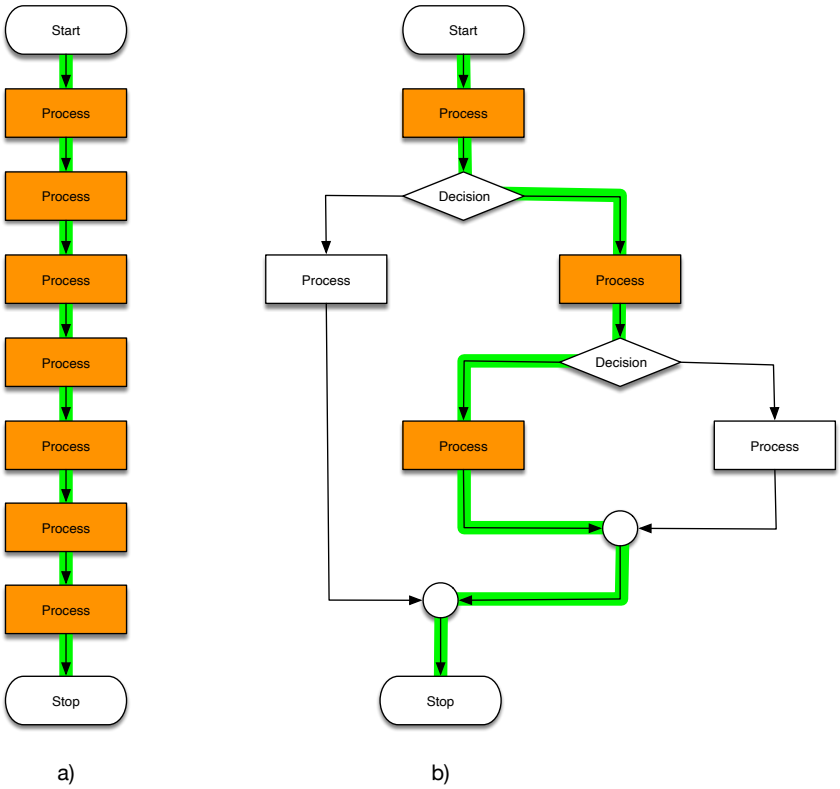
Instrukcje można zagnieżdżać. Tworzą one strukturę hierarchiczną tj. dla każdej instrukcji można wskazać instrukcje w niej umieszczone oraz instrukcję, w której została ona umieszczona.

Każdą instrukcję złożoną można zagnieździć w innej jeszcze bardziej złożonej instrukcji.

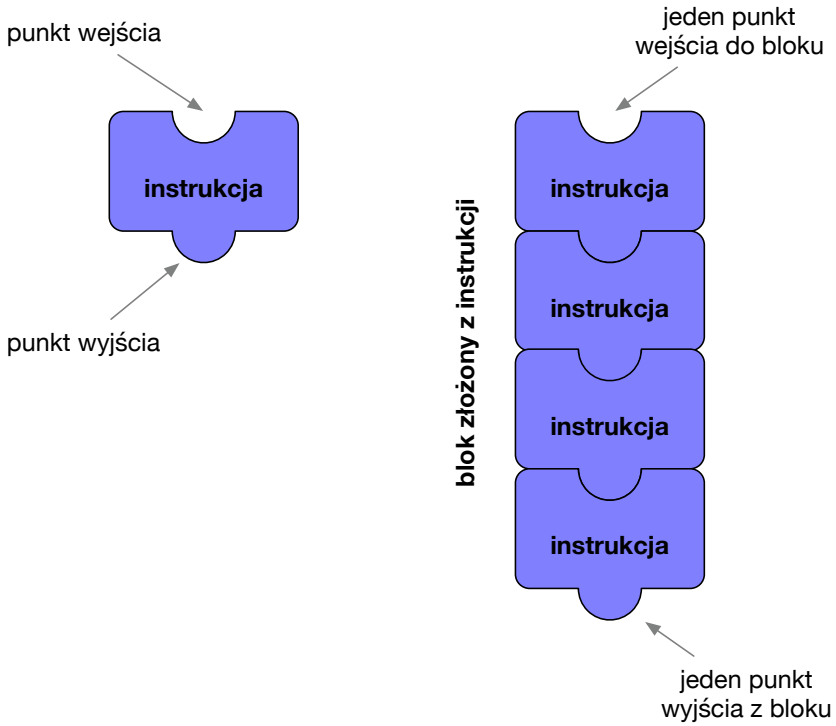
Tworzenie programów z hierarchicznie zagnieżdżonych instrukcji jest cechą programowania strukturalnego.

Składowe algorytmu wyraża się instrukcjami złożonymi.

W programowaniu strukturalnym wyróżnia się dwie metody projektowania algorytmu:



Rysunek 3.1: a) Jedyna ścieżka obliczeń. b) Jedna z trzech ścieżek obliczeń.



Rysunek 3.2: Instrukcja i blok instrukcji.

- **z góry na dół** (ang. top-down) gdy początkowe złożone składowe dzieli się hierarchicznie na coraz prostsze, tj. wychodzi się od ogólnej koncepcji algorytmu a następnie coraz bardziej ją uszczegóławia (analiza).
- **z dołu do góry** (ang. bottom-up) gdy proste składowe algorytmu łączy się hierarchicznie w coraz bardziej złożone, tj. wychodzi się od drobnych elementów i przeprowadza z nich syntezę całości.

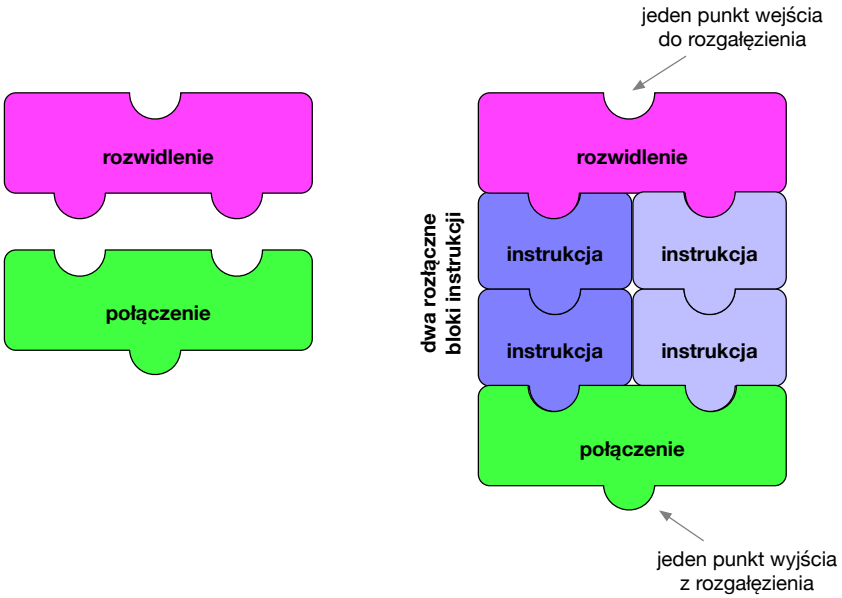
3.2.1 Punkt wejścia i wyjścia

Aby było możliwe zagnieżdżanie instrukcji, przyjęło się, że każda z nich ma jeden punkt wejścia (miejsce gdzie rozpoczynamy jej wykonywania) i jeden punkt wyjścia (miejsce gdzie kończymy jej wykonywanie).

Na rysunku 3.2 przedstawiono instrukcję prostą i ciąg instrukcji połączony w jeden blok.

3.2.2 Rozwidlenie i połączenie

Na rysunku 3.3 przedstawiono blok instrukcji w postaci rozgałęzienia na dwa bloki. Takie rozgałęzienie również posiada jedno wejście (rozwidlające się na dwie ścieżki) i jedno wyjście (łączące dwie ścieżki). Ważne alby rozwidlenia i



Rysunek 3.3: Strukturalne rozgałęzienie.

DRAFT

połączenia występowały parami (każdemu rozwidleniu musi odpowiadać połączenia łączące dwie rozdzielone ścieżki).

Rysując schematy blokowe można utworzyć taki schemat, któremu nie będzie odpowiadać poprawnie zagnieżdżona instrukcja złożona. Przykład takiego schematu przedstawiono na rysunku 3.4.

Proszę zauważyć, że rozwidleniu **Decision 2** nie odpowiada żadne połączenie. Nie jest nim połączenie **A**, ponieważ nie łączy ono ścieżki zawierającej blok **Process 3**. Nie jest nim również połączenie **B**, bo przechodzi przez nie ścieżka zawierająca blok **Process 1** a blok ten nie leży na żadnej ścieżce rozwidlającej się w **Decision 2**.

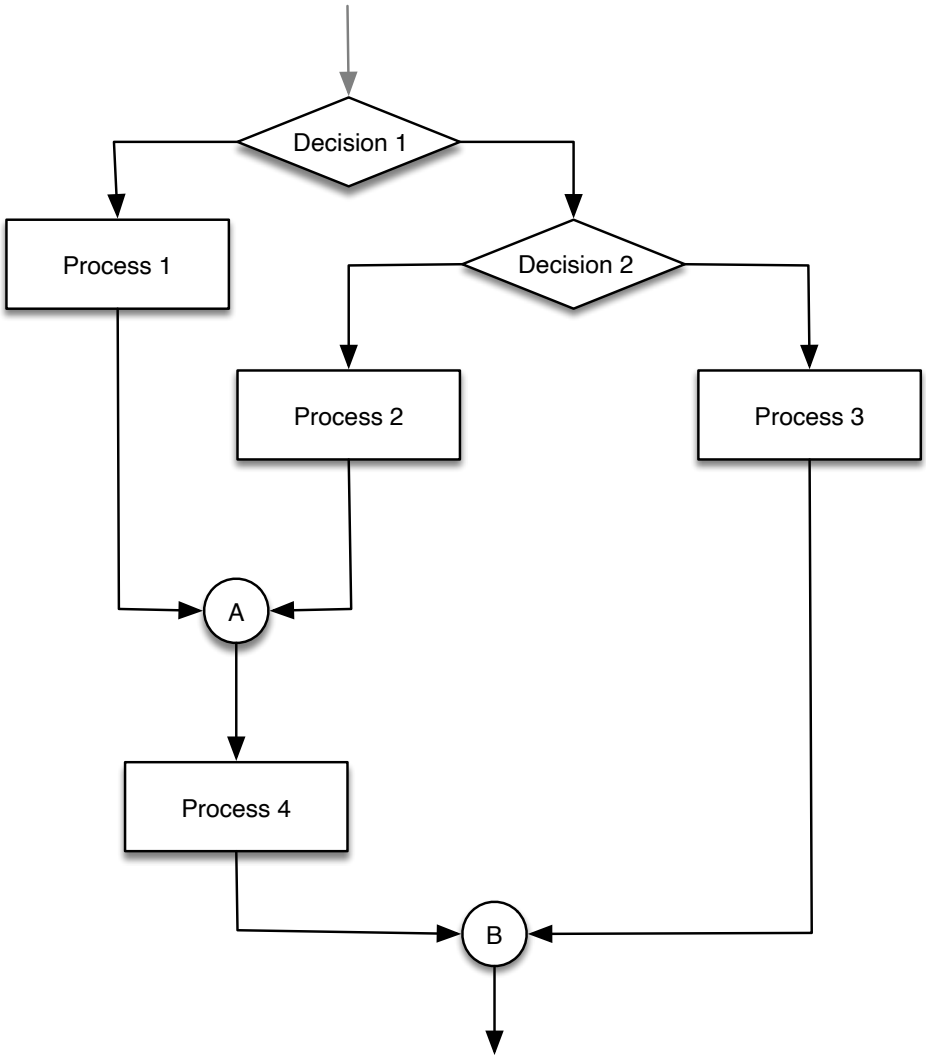
3.3 Instrukcja if

Instrukcja warunkowa w języku C ma następującą postać:

```
if(Warunek)
{
  // instrukcje wykonywane gdy warunek jest prawdziwy
}
else
{
  // instrukcje wykonywane gdy warunek jest fałszywy
}
```

Jeśli w którymś z dwóch bloków ujętych w klamrowe nawiasy występuje tylko jedna instrukcja, to nawiasy otaczające tę jedną instrukcję można pominać:

```
if(Warunek)
```



Rysunek 3.4: Fragment schematu blokowego, którego nie da się zapisać instrukcją strukturalną.

```
  // jedna instrukcja wykonywana gdy warunek jest prawdziwy
else
{
  // instrukcje wykonywane gdy warunek jest fałszywy
}
```

Jeśli blok po słowie kluczowym **else** nie zawiera instrukcji, to można go pominąć:

```
if(Warunek)
{
  // instrukcje wykonywane gdy warunek jest prawdziwy
}
```

Rozpatrzmy prosty przykład programu, który rozpoznaje czy zadany rok jest przestępny czy nie.

```
// if.c

#include <stdio.h>

int main(void)
{
  int rok;
  printf("Podaj rok: ");
  scanf("%d", &rok);
  if((rok % 4 == 0 && rok % 100 != 0) || rok % 400 == 0)
    printf("%d jest rokiem przestępnym i ma 366 dni\n", rok);
  else
    printf("%d jest rokiem zwykłym i ma 365 dni\n", rok);
  return 0;
}
```

DRAFT

Często mamy do czynienia z sytuacją, w której pod zmienną podstawiana jest jedna z dwóch wartości, w zależności czy pewien warunek jest spełniony czy nie:

```
if(Warunek)
  zmienna = Pierwsza_Wartość;
else
  zmienna = Druga_Wartość;
```

W takiej sytuacji możemy skorzystać z dostępnego w języku C wyrażenia warunkowego:

```
zmienna = Warunek ? Pierwsza_Wartość : Druga_Wartość;
```

Dla przykładu, poniższa instrukcja podstawia pod zmienną y wartość bezwzględną zmiennej x:

```
y = (x >= 0)?x:-x;
```

Jeśli użyje się najwyższego poziomu optymalizacji kodu (opcja kompilacji `-O3`), to powyższa instrukcja podstawienia będzie miała dokładnie taki sam przekład, jak instrukcja warunkowa:

```
if(x >= 0)
  y = x;
else
  y = -x;
```

3.4 Instrukcja goto

Szczególną instrukcją jest instrukcja służąca do przeniesienia wykonywania z jednego miejsca programu do innego.

Instrukcją tą jest instrukcja skoku **goto**. Ma ona postać:

```
goto Label;
```

gdzie `Label` jest etykietą miejsca, do którego należy przenieść dalsze wykonywanie programu.

Etykietę w języku C definiuje się pisząc jej nazwę a następnie dwukropek oddzielający ją od instrukcji, którą poprzedza:

```
Label: statement;
```

Gdybyśmy chcieli zapisać w języku C fragment programu opisany schematem blokowym z rysunku 3.4, to albo musielibyśmy dwukrotnie zapisać w nim blok **Process 4**:

```
if(Decision 1)
{
  Process 1;
  Process 4;
}
else
  if(Decision 2)
  {
    Process 2;
    Process 4;
  }
  else
    Process 3;
```

DRAFT

albo zastosować instrukcję skoku:

```
if(Decision 1)
{
  Process 1;
  goto Label;
}
else
  if(Decision 2)
  {
    Process 2;
Label:
    Process 4;
  }
  else
    Process 3;
```

Aby unikać takich niezręcznych sytuacji[1] stosujmy przy projektowaniu algorytmu zasadę **z góry na dół** albo **z dołu do góry** a programy nasze zawsze będą poprawnym programami strukturalnymi. Unikniemy w nich niepotrzebnych powtórzeń albo, co gorsze, instrukcji skoku.

[1] Szczególnie użycie instrukcji skoku jest niezręczne, bo instrukcja warunkowa **if(Decision 2)** ma teraz dwa punkty wejścia: gdy warunek jest spełniony oraz w miejscu etykiety **Label**.

Rozpatrzmy jeszcze przykład zastosowania instrukcji skoku do wielokrotnego wykonania tego samego fragmentu programu:

```c
// goto.c

#include<stdio.h>

int main(void)
{
  int n;
  printf("Podaj ile liczb wydrukować: ");
  scanf("%d", &n);
  printf("\n");
  int i = 1;

Początek:

  if(i > n)
    goto Koniec;
  printf("%d\n", i);
  i = i + 1;
  goto Początek;

Koniec:

  return 0;
}
```

DRAFT

Na rysunku 3.5 przedstawiono schemat blokowy powyższego programu. Zaznaczono na nim cykl, w którym drukowane są kolejne wartości zmiennej `i` od 1 do zadanego `n`. Gdy wartość zmiennej `i` przekroczy wartość `n`, wtedy program opuszcza cykl przechodząc do etykiety `Koniec`.

Przykład uruchomienia powyższego programu:

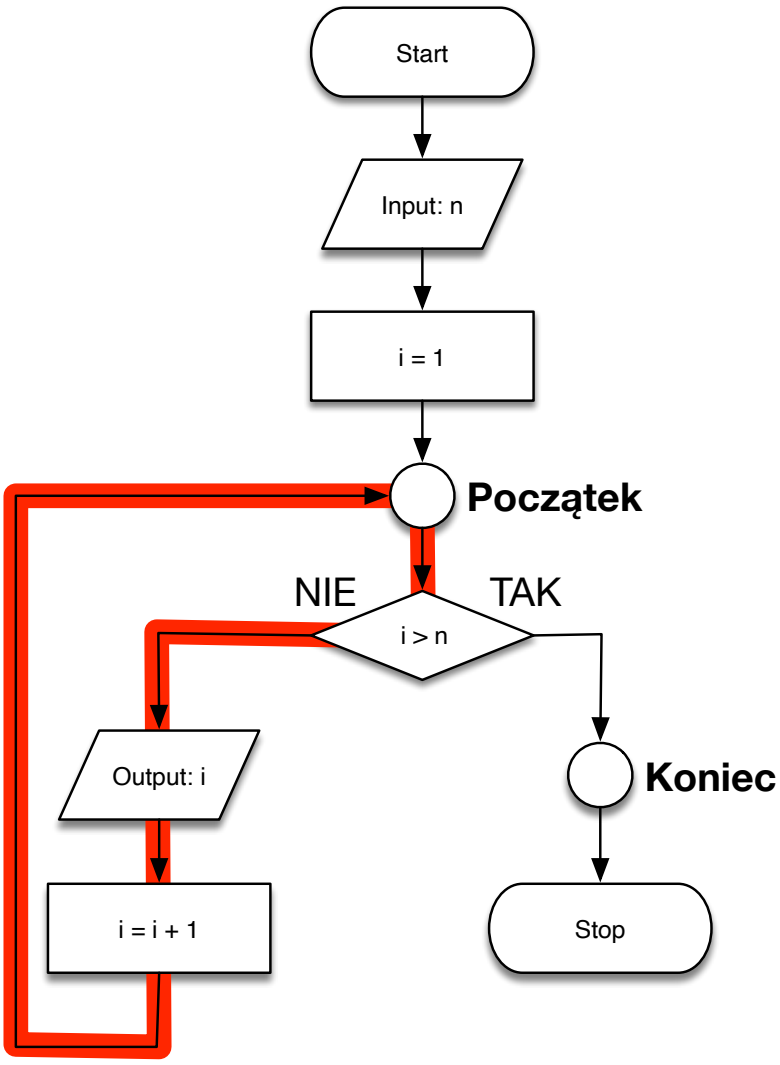
```
$ ./goto
Podaj ile liczb wydrukować: 5

1
2
3
4
5
```

W języku C dostępne są trzy różne sposoby zapisania iteracji w programie (przedstawimy je w następnym rozdziale), dzięki którym nie jest konieczne użycie w takich sytuacjach instrukcji skoku.

Proszę zapamiętać, że podczas zajęć ze *Wstępu do Informatyki i Programowania* **NIGDY WIĘCEJ** nie będziemy używać instrukcji **goto**[2]. Programy korzystające z tej instrukcji będą wymagały poprawy i usunięcia jej.

[2]Inne instrukcje dostępne w języku C, między innymi **break** i **continue**, o których będzie dalej, pozwalają unikać użycia instrukcji **goto**.



Rysunek 3.5: Cykl drukujący kolejne wartości zmiennej `i`.

3.5 Instrukcja switch

Wyobraźmy sobie sytuację, w której w zależności od tego jaka jest wartość wyrażenia `Wyr` zostają wykonane następujące bloki instrukcji:

`Blok1` gdy wartość wynosi `Wart1`

`Blok2` gdy wartość wynosi `Wart2`

`Blok3` gdy wartość wynosi `Wart3`

`Blok4` gdy wartość wynosi `Wart4`

Możemy wyrazić to zagnieżdżonymi instrukcjami warunkowymi:

```
if(Wyr == Wart1)
  Blok1;
else
  if(Wyr == Wart2)
    Blok2;
  else
    if(Wyr == Wart3)
      Blok3;
    else
      if(Wyr = Wart4)
        Blok4;
```

DRAFT

Rozwiązanie takie ma jednak dwie wady:

1. Wielokrotnie liczy wartość tego samego wyrażenia.
2. Zagłębienie instrukcji (jej złożoność) rośnie wraz ze wzrostem liczby przypadków.

Możemy to zapisać prościej stosując dostępną w języku C instrukcję **switch**:

```
switch(Wyr)
{
  case Wart1: Blok1;
  case Wart2: Blok2;
  case Wart3: Blok3;
  case Wart4: Blok4;
}
```

w której dalsze wykonywanie zostaje przeniesione do miejsca odpowiadającemu danemu przypadkowi wartości.

Jednak powyższa instrukcja wymaga jeszcze drobnej poprawki. Otóż jeśli sterowanie zostanie przeniesiona w dane miejsce, to zostaną wykonane wszystkie dalsze bloki.

Musimy po każdym bloku (poza ostatnim) umieścić instrukcję **break**, która spowoduje opuszczenie instrukcji **switch**:

```
switch(Wyr)
{
  case Wart1: Blok1;
              break;
  case Wart2: Blok2;
```

```
                break;
  case Wart3: Blok3;
                break;
  case Wart4: Blok4;
}
```

Instrukcja **break** służy do opuszczenia instrukcji złożonej, w której została wykonana. Odpowiada ona instrukcji skoku ale w jedno ustalone miejsce: punkt wyjścia instrukcji złożonej. Zatem nawet w przypadku użycia instrukcji **break**, instrukcja złożona, w której ją użyto, ma nadal tylko jeden punkt wyjścia.

W instrukcji **switch** możemy użyć jeszcze szczególnego przypadku **default**, który zostanie wybrany, jeśli wartość wyrażenia jest inna niż każda z wartości wymienionych we wszystkich przypadkach **case**:

```
switch(Wyr)
{
  case Wart1: Blok1;
                break;
  case Wart2: Blok2;
                break;
  case Wart3: Blok3;
                break;
  case Wart4: Blok4;
                break;
  default:    Blok5; // gdy żaden z powyższych przypadków
}
```

DRAFT

W poniższym przykładzie, instrukcja **switch** może służyć do zamiany liter na kody alfabetu Morse'a:

```
switch(ch)
{
  case 'a': printf(".-"   ); break;
  case 'b': printf("-..."); break;
  case 'c': printf("-.-."); break;
  case 'd': printf("-.." ); break;
  case 'e': printf("."    ); break;
  ...
  case 'z': printf("--..");
}
```

W instrukcji **switch** można blok instrukcji związać z więcej niż jedną wartością. W poniższym przykładzie, w przypadku kiedy `Wyr` ma wartość 2, 3 lub 4 wówczas wykona się `Blok2`:

```
switch(Wyr)
{
  case 1: Blok1;
          break;
  case 2:
  case 3:
  case 4: Blok2;
          break;
  case 5: Blok3;
}
```

3.6 Ciekawostka

W języku C instrukcja `switch` jest tylko pewną formą skoku. Fraza `case Value:` pełni rolę etykiety oznaczającej miejsce, do którego następuje skok w przypadku gdy wyrażenie w instrukcji `case` ma wartość `Value`. Ważne jest tylko aby „etykieta” `case Value:` znajdowała się w zasięgu instrukcji `case`.

Można się o tym przekonać kompilując następujący program poprawny składniowo ale błędny semantycznie:

```c
// switch.c

#include <stdio.h>

int main(void)
{
  switch(1)
  {
    for(int i = 0; i < 10; i++)
      case 1: printf("i=%d\n", i);
  }
  return 0;
}
```

Zaskakujący wynik działania powyższego programu:

```
$ ./switch
i = 263647286
```

DRAFT

Nawet kompilując program z opcją `-Wextra` nie zostaje wykryty błąd:

```
$ clang -Wextra --pedantic -std=c11 -o switch switch.c
$
```

Dopiero statyczna analiza kodu ostrzega nas o użyciu niezainicjowanej zmiennej:

```
$ clang --analyze switch.c
switch.c:10:15: warning: 2nd function call argument is an uninitialized value
      case 1: printf("i=%d\n", i);
              ^~~~~~~~~~~~~~~~~~
1 warning generated.
```