
www.wioowszkole.org

**Karol Sieńkowski**

# Laby

## Wstęp do nauki programowania

Publikacja powstała w ramach projektu
**Wolne i Otwarte Oprogramowanie w Szkole**

Dziękuję

***Mateuszowi Mikosowi***

za pomoc i zaangażowanie w tłumaczenie
poleceń Laby na język polski

# Laby

## Wstęp do nauki programowania

## Spis treści

# Wstęp

Laby to program dla systemu Linux stworzony dla dzieci, by zachęcić je do nauki języków programowania. Jeśli ktoś używał języka Logo, to może dostrzec małe podobieństwo. W Logo poruszamy się żółwiem, w Laby występuje mrówka. W obu przypadkach zwierzęta poruszają się po wpisaniu odpowiednich komend. I Logo i Laby zachęca do nauki programowania przez zabawę. Laby choć nie ma takich możliwości jak Logo, to ma jednak jedną zaletę. Występują tu zadania, lub mówiąc językiem bliższym młodzieży, poziomy, czy tez misje, które trzeba spełnić. Uczniowie rozpoczynając zabawę z Laby mają przed sobą konkretny cel, do którego będą dążyć. Wykonanie misji daje dużo frajdy. Zadania mają rosnący stopień trudności, a możliwość tworzenia dodatkowych misji sprawia, że jest do doskonałe narzędzie dla nauczyciela.

# Języki programowania w Laby

Na chwilę obecną Laby umożliwia poznanie składni następujących języków programowania: Ocaml, C, Java, Ruby, Python, Prolog. To chyba wystarczająco dużo. Warunkiem użycia tych języków w zabawie jest zainstalowanie ich kompilatorów w systemie. **Umawiamy się, że w tym opisie będziemy używać składni języka C**. W tym języku prawie każda linijka zakończona jest znakiem „ **;** ”, a dodatkowo po poleceniach stawiamy znaki **( )** (*Ilustracja 1*).

# Mrówka i język polski

W wyniku tłumaczenia Laby na język polski pojawiły się problemy z niektórymi słowami. Przykładowo w oryginale, czyli w języku angielskim po labiryncie porusza się **ant** (*pol. mrówka*) Jednak użycie słowa mrówka w kodzie nie było możliwe ze względu na obecność w nim litery „ó”. Dlatego też tłumacząc program zdecydowaliśmy się poszukać takich polskich słów, które dobrze oddałyby istotę Laby i nie zawierały obecnych w naszym języku tzw. „krzaczków”. Nie było to łatwe, ale udało się zrobić to na tyle dobrze i intuicyjnie, że nie sprawiało uczniom testującym tłumaczenie żadnych problemów.

# Cel gry

W każdej misji celem jest wyprowadzenie mrówki z pułapki. Nie jest to łatwe. Na jej drodze mogą pojawiać się przeszkody-kamienie, które może przenosić. Czekają też na nią zasadzki-pajęczyny. Te z kolei musi omijać, lub niszczyć zrzucając na nie kamień. Im wyższy poziom, tym bardziej skomplikowana misja. Do niektórych misji pojawiają się podpowiedzi, tzn polecenia, których można użyć. Ułatwia to zwłaszcza na początkowym etapie wdrożenie się do gry.

# Podstawowe polecenia

Nazwy poleceń i elementów występujących w języku polskim zostały tak dobrane, by ich zapamiętanie nie było trudne. Zauważysz, że niektóre z nich pisane są dużą literą, a niektóre małą. To nie błąd, takie jest założenie Laby, by pewne obiekty labiryntu wyróżnić dużą literą. Przy niektórych poleceniach pojawia się znak „_”. Zastępuje on spację w zwrotach dwuwyrazowych (spacji nie można używać). Polecenia i obiekty w języku polskim mają następujące brzmienie:

Polecenia sterujące zachowaniem robaka

robak – to jego trzeba wyprowadzić z pułapki

do_przodu – ruch robaka o jedno pole ( o jeden krok) do przodu

w_prawo – obrót robaka o kąt 90 stopni w prawo

w_lewo – obrót robaka o kąt 90 stopni w lewo

zabierz – robak zabiera kamyk leżący przed nim

zostaw – robak zostawia kamyk na pole przed sobą

ucieknij – robak ucieka przez drzwi, które ma na polu przed sobą

Elementy Labiryntu

Kamyk – przeszkoda w postaci kamienia

Zasadzka – przeszkoda w postaci pajęczyny

Mur – element graniczny labiryntu

Drzwi – element labiryntu reprezentujący wyjście z niego

Nic – przed robakiem nic się nie znajduje, jest pusta droga (pusty korytarz)

Polecenie uaktywniające zmysł wzroku robaka

zobacz – komenda pozwalająca robakowi zobaczyć jedno pole przed nim.

I to wszystkie podstawowe polecenia, które będą wykorzystywane w zabawie. Oprócz nich pojawią się też *instrukcje warunkowe*, *pętle* i *funkcje* zapisywane zgodnie ze składnią wybranego języka programowania.

**Uwaga!**

W niektórych wyżej napisanych wyrazach **pierwsza litera jest duża**. Dotyczy to elementów labiryntu i jest bardzo ważne!

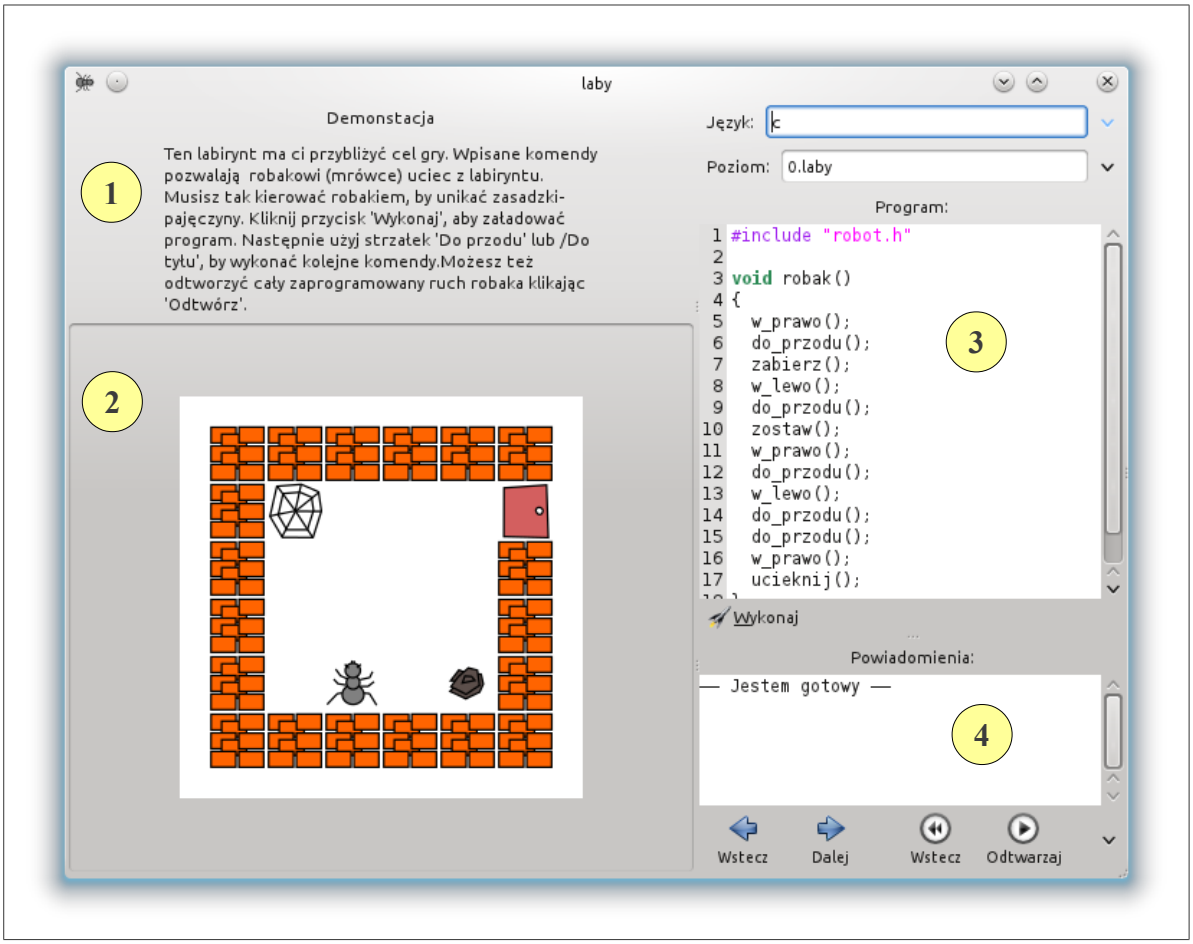
# Trudne początki

Kilka pierwszych poziomów ma za zadanie zapoznać graczy ze składnią poleceń, z komunikatami o występujących w składni błędach itp. Są to plansze proste. Przejście tych misji daje dużo radości dzieciom nie tylko tym zdolniejszym, ale również tym, którzy mają słabiej rozwiniętą wyobraźnię przestrzenną i radzą sobie wyraźnie gorzej.

Program domyślnie znajduje się w grupie **Menu =>Gry**. To dobry pomysł, bo już na samym początku dobrze nastraja. Po jego uruchomieniu otwiera się okno dzielące ekran na 4 obszary (*Ilustracja 1*):

1. Informacje o aktualnie wybranym poziomie
2. Obraz labiryntu
3. Edytor poleceń
4. Okno powiadomień

Dodatkowo w niektórych planszach pod obrazem labiryntu pojawia się jeszcze okno podpowiedzi zawierające wskazówki do przejścia danego poziomu.




*Ilustracja 1: Okno programu Laby witające nas po jego uruchomieniu*

Na samym początku program wita nas planszą demonstracyjną **0.laby** (*Ilustracja 1*). Ma ona za zadanie przybliżyć cel gry i zapoznać z obsługą programu. Wpisane domyślnie polecenia są tak dobrane, by zaobserwować podstawowe czynności, jakie może zrobić mrówka. Aby sprawdzić wpisany kod pod względem błędów należy kliknąć przycisk **Wykonaj**. Jeśli błędów w składni poleceń sterujących mrówką nie ma, to w oknie powiadomień zobaczymy komunikat

„**---Jestem gotowy---**”.

W przeciwnym razie trzeba odnaleźć błędy i je poprawić. Pomocne jest tu okno powiadomień sygnalizujące gdzie tkwi problem. Warto tu dodać, że jeden z najczęstszych błędów jakie się pojawiają, to niewłaściwie zamknięte nawiasy klamrowe. Gdy już uporamy się z błędami, to mamy do wyboru kilka opcji.

- klikamy wielokrotnie przycisk **Dalej.** Mrówka małymi kroczkami będzie przesuwać się do przodu
- klikamy przycisk **Odtwarzaj**. Każda z linijek kodu wykona się automatycznie w normalnym tempie
- klikamy przycisk **Naprzód**. Mrówka będzie poruszać się w zwiększonym tempie.

Program:

```
1 #include "robot.h"
2
3 void robak()
4 {
5
6
7 }
8
```

*Ilustracja 2: Plansza przygotowana do wprowadzenia nowych poleceń.* ***Tych linii, które zostały nie wolno kasować!***

Dostępne są również przyciski pozwalające cofnąć wykonane przez mrówkę czynności. Na początku warto korzystać z opcji pierwszej, gdyż ułatwia ona zrozumienie działania wpisanych poleceń sterujących.

Po przeanalizowaniu pierwszej planszy można przejść do poziomu **1a.laby**. Należy uprzedzić uczniów, by przystępując do rozwiązywania kolejnych poziomów usuwali tylko polecenia, które znajdują się między krańcowymi nawiasami **{ }** (*Ilustracja 2*). Gdy usuną więcej, to pojawią się błędy. Należy wówczas uruchomić Laby ponownie, wtedy pojawią się usunięte linie.

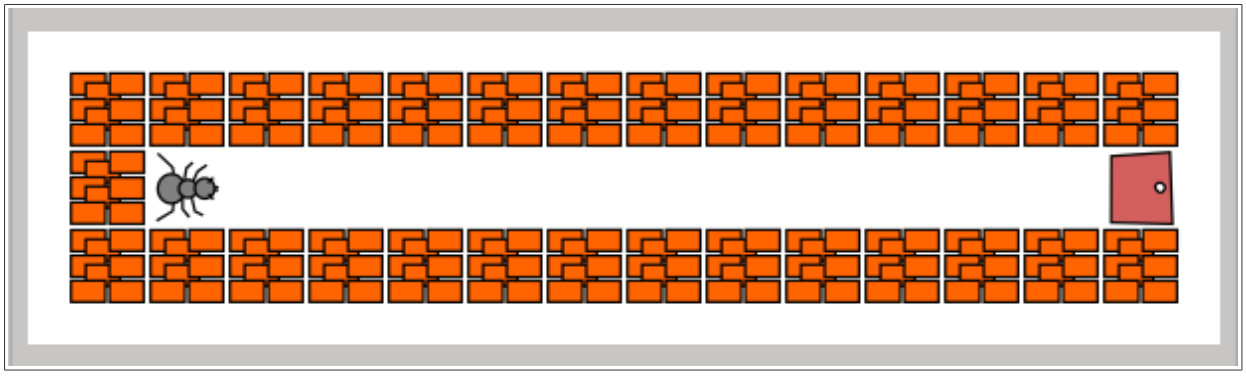
Warto namówić uczniów, by stosowali wcięcia. Dobrze jest by wyrobili sobie właśnie ten nawyk na samym początku. Pozwoli im to szybciej odszukać błąd, albo wybrane miejsce w kodzie i zaowocuje w przyszłości, gdy zaczną swoją prawdziwą przygodę z programowaniem.

W chwili obecnej Laby ma jedną niedogodność. Nie pozwala zapisać kodu programu. Można go oczywiście wkleić do edytora tekstu., ale nie jest to rozwiązanie komfortowe i wygodne.

## Wskazówki do trudniejszych poziomów

Gdy uczniowie zaczynają wykonywać coraz bardziej złożone misje, to zauważają, że ciągłe powtarzanie tych samych komend jest nużące i niezmiernie długo trwa. Pytają wtedy, czy nie da się zrobić tego jakoś szybciej. Jest to znak, że nadszedł czas, by poznali instrukcję pętli. W Laby można wykorzystywać takie pętle jakie są obecne w ustawionym języku programowania. Skupimy się jednak na pętli **while** (*pol. dopóki*), która dobrze nadaje się do rozwiązania mrówkowych problemów. Pierwsza okazja do użycia takiej pętli pojawia się przy poziomie **2a**.

## Poziom 2a



*Ilustracja 3: Plansza z poziomu 2a*

Poziom ten pozornie jest łatwy. Nie ma żadnych przeszkód, kamieni i pajęczyn. Ale jeśli chcemy dać uczniom możliwość zgłębiania tajników programowania, to muszą się przy tej planszy trochę wysilić. Mrówka ma przed sobą pustą drogę, Może więc iść aż do momentu, gdy coś przed sobą zobaczy.

Zatem możemy tę misję wykonać na 2 sposoby:

1. Wpisać 11 razy polecenie `do_przodu( );`

   wpisać polecenie `ucieknij( );`

2. Użyć pętli **while**, która zastąpi 11 krotne wpisanie polecenia `do_przodu( );` jednokrotnym jego wpisaniem.

   Wpisać polecenie `ucieknij( );`

   Zatem rozwiązanie w takim przypadku będzie miało składnię:

```
while (zobacz( ) = = Nic)
  {
     do_przodu( );
  }
ucieknij( );
```

Zapis ten należy rozumieć następująco:

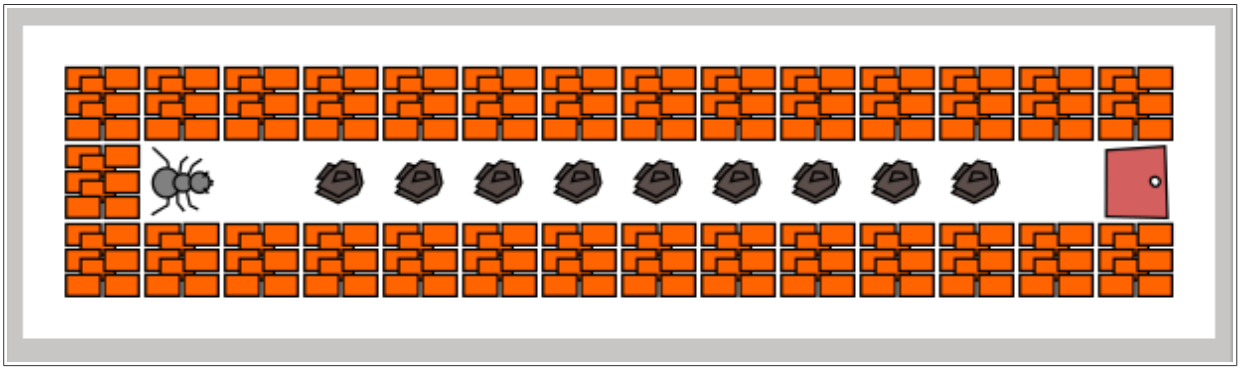
**Dopóki robak nic przed sobą nie widzi, to ma iść do przodu.**

Pętla wykona się 11 razy, bo robak 11 razy będzie miał przed sobą pustą drogę, czyli nic nie będzie widział. W początkowym etapie używania pętli często dochodzi do błędów z nawiasami. Uczniowie albo zapominają domknąć jakiś nawias, albo stawiają ich za dużo. Edytor kodu w Laby ma ciekawą funkcję, która pomaga takie błędy wyłapać. Gdy kursor znajdzie się w okolicy nawiasu, to automatycznie kolorem szarym zaznaczany jest odpowiadający mu nawias otwierający lub zamykający (*Ilustracja 4*).

```
void robak()
{
 while (zobacz() == Mur)
 {
  w_prawo();
  do_przodu();
 }
}
```

*Ilustracja 4: Kursor wskazuje jeden nawias a, kolorem szarym zaznaczony jest nawias mu odpowiadający*

## Poziom 2b



*Ilustracja 5: Plansza poziomu 2b*

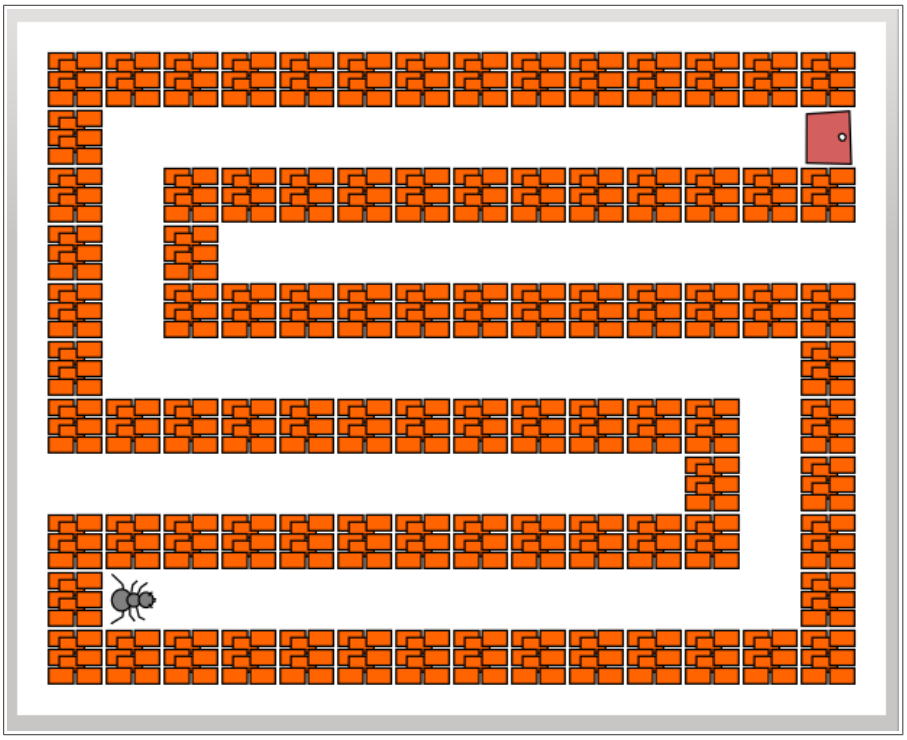
Poziom ten zmusza do użycia innej składni pętli **while**. Tym razem robak musi przerzucać kolejno 9 kamieni za siebie. Tylko to umożliwi mu dotarcie do drzwi. Aby przenieść jeden kamyk robak musi: *zabrać go, dwa razy obrócić się w lewo, zostawić kamyk, 2 razy obrócić się w lewo, podejść pod kolejny kamyk*. Oczywiście można to zrobić wpisując 9 razy ten sam zestaw poleceń. Być może, ktoś tak postąpi. Ale szybko znajdą się tacy uczniowie, którzy mając w pamięci poziom **2a** dostrzegą możliwość użycia pętli **while**. Tyle, że w tym przypadku użycie takiego warunku jak w zadaniu poprzednim, czyli **zobacz( ) = = Nic** nie przyniesie pożądanego efektu. W tym zadaniu pętla musi wykonywać się wtedy, gdy robak zobaczy kamyk. Zatem jej składnia będzie następująca:

```
while (zobacz( ) = = Kamyk)
{
   zabierz( );
   w_lewo( );
   w_lewo( );
   zostaw( );
   w_lewo( );
   w_lewo( );
   do_przodu( );
}
```

Dzięki takiej konstrukcji pętli za każdym razem, kiedy robak dostrzeże przed sobą kamyk, to wykona 7 czynności, których efektem będzie przeniesie go za siebie i stanięcie przed kolejnym kamykiem.

Często uczniowie dobrze zapisują instrukcję pętli w tym poziomie, ale mimo to robak nie przesuwa się. Na początku trudno im zrozumieć, że aby pętla zadziałała, to robak musi znaleźć się przed kamykiem. A więc w przypadku poziomu **2b** należy przed instrukcją **while** wpisać jeszcze polecenie, które przesunie robaka do pierwszego kamienia. Dopiero wtedy pętla zadziała.

## Poziom 2c



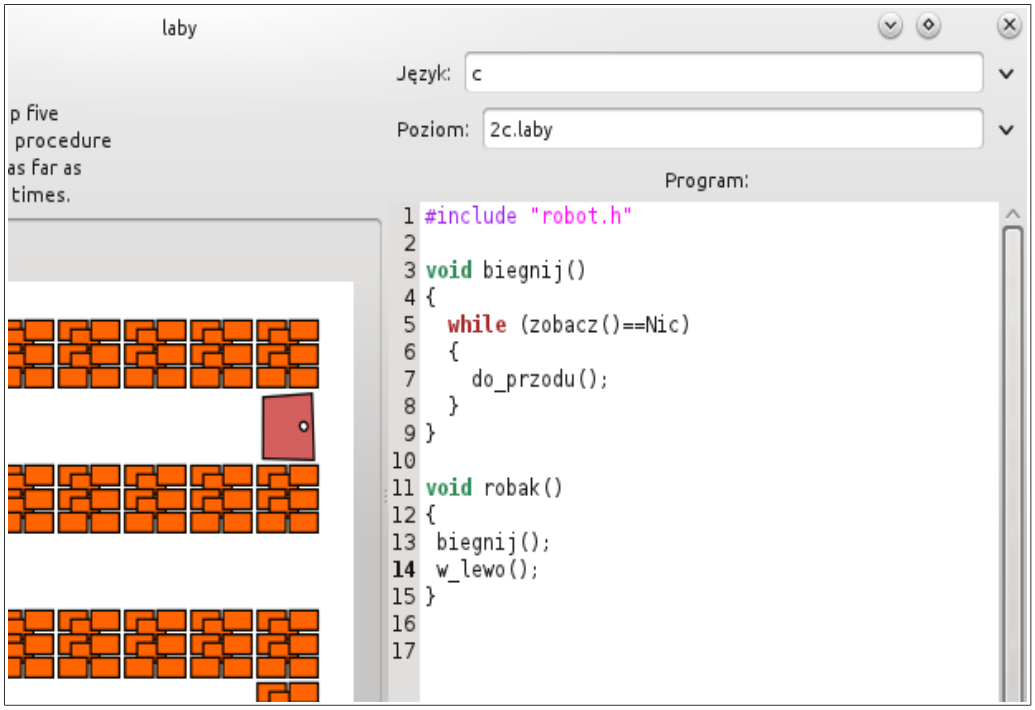
*Ilustracja 6: Plansza poziomu 2c.laby*

W przypadku tego poziomu po raz pierwszy pojawia się okazja do zdefiniowania funkcji. Uczniowie mający za sobą doświadczenia z poziomami wcześniejszymi szybko dostrzegają, że w tym zadaniu można użyć składni pętli **while** z zadania **2a**. Tyle, że tutaj takiej pętli należy użyć aż 5 razy – na każdym prostym odcinku drogi. I znowu można ją po prostu napisać 5 razy. Będzie to dobry sposób dla słabszych uczniów. Ale zdolniejsi mogą spróbować nauczyć robaka reagowania na nowe polecenie. W tym celu należy zdefiniować funkcję. Funkcji można nadać dowolną nazwę, ale umówmy się, że w tym przykładzie funkcja będzie nazywać się **biegnij** (wpisując jej nazwę uważamy jedynie, by nie było w wyrazie „krzaczków" i spacji). Do definiowania funkcji służy słowo **void**, które być może rzuciło się niektórym w oczy (funkcja void robak( `) ). Składnia funkcji będzie więc następująca:

```
void biegnij()
{
   while (zobacz( ) = = Nic)
   {
      do_przodu( );
   }
}
```

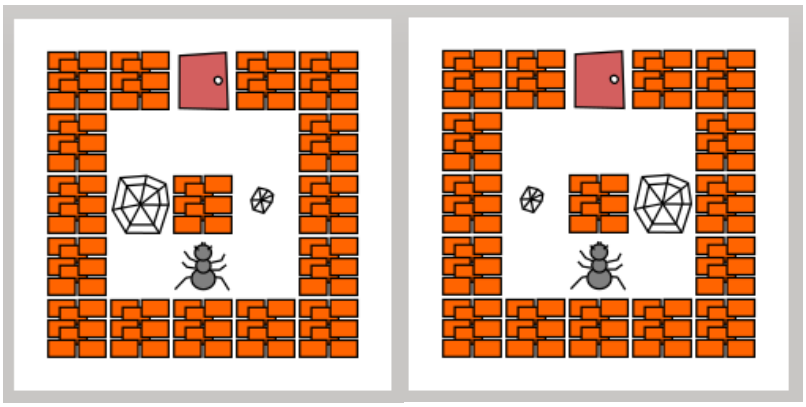
Funkcję należy wpisać zaraz po dyrektywie **#include „robot.h"**, a więc przed funkcją **void robak()** (*Ilustracja 7*)

Od tej pory, gdy wydamy polecenie biegnij( ); to robak będzie przesuwał się do przodu aż do momentu, gdy coś przed sobą zobaczy.




*Ilustracja 7: Przykład użycia funkcji biegnij do rozwiązania poziomu 2c.laby.*
***Uwaga!** Przykład demonstruje miejsce wpisania funkcji **biegnij** oraz sposób jej wywołania.. Przykład nie jest kompletnym rozwiązaniem poziomu 2c.laby.*

## Poziom 3a



*Ilustracja 8: W poziomie 3a.laby losowo zmienia się położenie dużej pajęczyny*

Poziom ten rozpoczyna serię poziomów dynamicznych. Przez dużą pajęczynę robak przejść nie może, natomiast mała pajęczyna nie jest dla niego przeszkodą. Sytuacja w labiryncie zmienia się losowo. Po każdym naciśnięciu przycisku **Wykonaj** zasadzka w postaci dużej pajęczyny zmienia swoje miejsce. To dodatkowe utrudnienie. Aby sobie z nim poradzić należy użyć instrukcji warunkowej.

Robak idąc w jakimś kierunku musi wybadać, co ma przed sobą. Musi sprawdzić, czy przed nim jest mała, nieszkodliwa pajęczyna (przez którą może normalnie przejść). Są dwie możliwości:

1. Jeśli tak, to może iść dalej do wyjścia,
2. Jeśli nie, to znaczy, że ma przed sobą dużą pajęczynę. Zatem musi zawrócić i iść do wyjścia z drugiej strony.

Spróbujmy to, co zapisaliśmy, przełożyć na język zrozumiały dla robaka. Załóżmy, że robak idzie w lewą stronę (pierwsza plansza na *Ilustracji 8*) i podchodzi do dużej pajęczyny. To właśnie w tym miejscu zachodzi kluczowe działanie robaka, czyli badanie, co ma przed sobą.

**Jeśli** zobaczył Zasadzkę, to ma zawrócić i iść do wyjścia drugą stroną.
**W przeciwnym razie** musi podążać do wyjścia tą samą drogą.

Te dwa zdania są chyba zrozumiałe. Zastąpmy teraz słowo ***Jeśli*** jego angielskim odpowiednikiem **If**, a zwrot ***W przeciwnym razie*** odpowiadającym mu słowem słowem **Else**. Zdania będą miały następujące brzmienie:

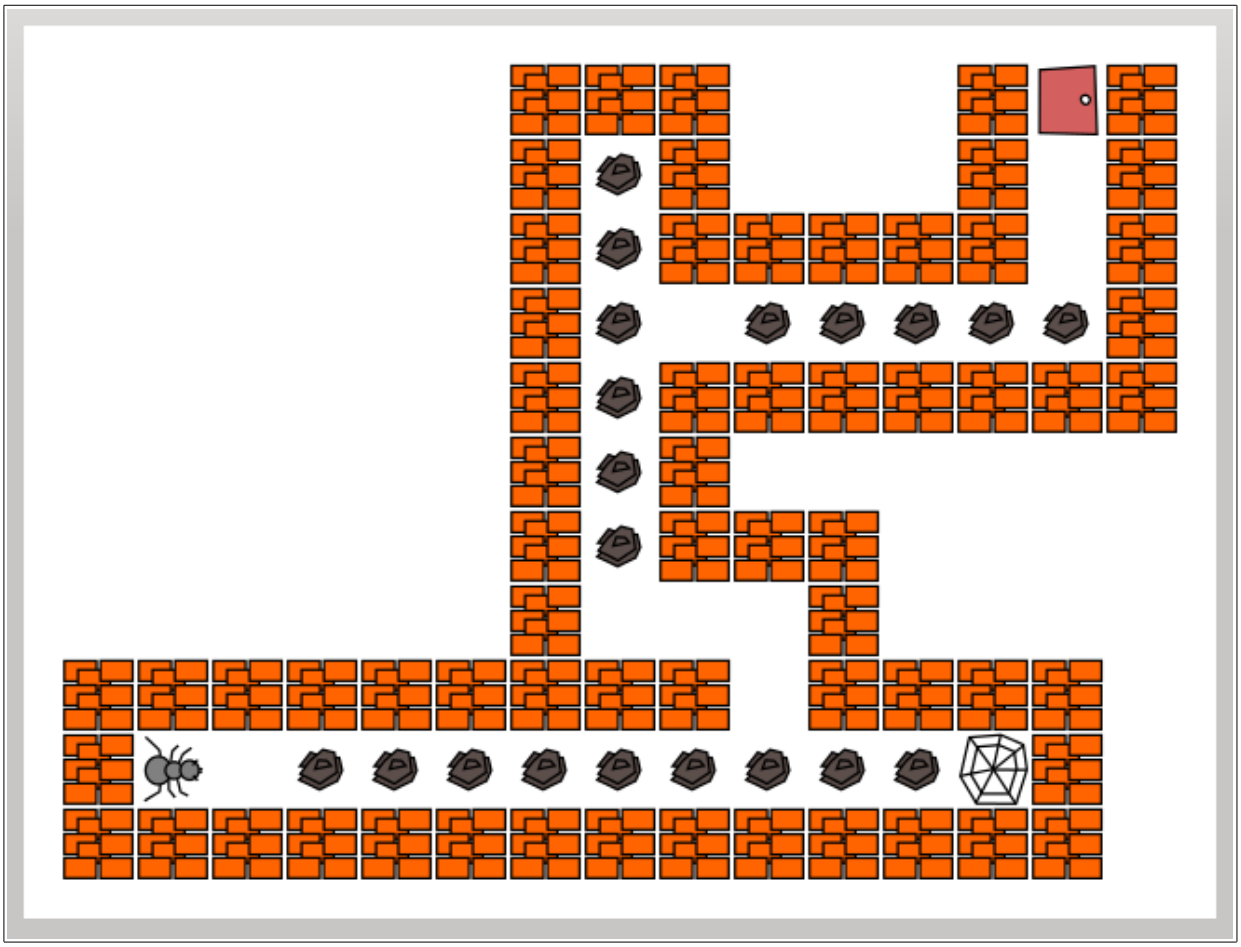
**If** zobaczył Zasadzkę, to ma zawrócić i iść do wyjścia drugą stroną.
**Else** musi podążać do wyjścia tą samą drogą

No i już zupełnie w języku zrozumiałym dla mrówki (dla ułatwienia dopisane są komentarze)

```
w_lewo( );
do_przodu( );
w_prawo( );  #podejście pod lewą pajęczynę
if (zobacz( ) = = Zasadzka)  #jeśli sieć jest duża
  {
     w_prawo( );
     do_przodu( );do_przodu() ;
     w_lewo( );
     do_przodu( );do_przodu( );
     w_lewo( );do_przodu( );w_prawo( );
     ucieknij( );
  }
else    #jeśli sieć jest mała
  {
     do_przodu( );do_przodu( );
     w_prawo( );
     do_przodu( );w_lewo( );
     ucieknij( );
  }
```

Pierwsze 3 polecenia doprowadzają robaka do pajęczyny. Dalej zachodzi sprawdzanie, czy jest ona dla niego groźna. Jeśli tak, to wykona się pierwsza grupa instrukcji między klamrami. Jeśli zaś przed nim nie ma zasadzki, to wykona się druga grupa instrukcji między klamrami (ta po słowie **else**)

## Poziom counting-the-rocks



*Ilustracja 9: Plansza poziomu counting-the-rocks*

Poziom ten nie jest trudny, ale wymaga nieco odmiennego podejścia niż wcześniejsze. Jeśli w tym zadaniu użyje się pętli **while** zbierającej i przerzucającej kamienie jak w poziomie **2b**, to robak dojdzie do samej pajęczyny w dolnym korytarzu i w nią wejdzie. W tym przypadku lepiej do zebrania kamieni znajdujących się w pierwszym korytarzu użyć pętli zliczającej **for** i zdefiniować ją tak by wykonała się 7 razy. Wtedy po jej zakończeniu robak znajdzie się u wejścia do drugiego korytarza i nie dojdzie do pajęczyny (*Ilustracja 10*). Pętla **for** (*pol. dla*) ma bardziej złożoną składnię niż pętla **while**. Wykorzystamy tu po raz pierwszy licznik, który będzie zliczał zebrane i przeniesione kamienie.

Pętla for ma 3 wyrażenia. W naszym przykładzie będą to: wartość początkowa zmiennej **x**, wartość końcowa zmiennej **x** oraz licznik zwiększający się o jeden po każdym wykonaniu pętli. Wygląda to następująco:

```
for (x = 0; x < 7; x++)
{
instrukcje
}
```

Wyjaśnijmy ten zapis. Zauważyłeś, że w powyższym kodzie pojawiła się litera **x**. **Jest to zmienna, której typ należy zdefiniować na początku kodu.** Nasza zmienna **x** będzie przyjmować wartości **od 0 do 7,** a więc będzie liczbą całkowitą (ang. integer), co zapisujemy `int x`; Znaczenie pozostałych zapisów;

`x=0` – oznacza to, że na samym początku wykonywania pętli zmienna ***x*** ma wartość **0**.

`x<7` – ten zapis oznacza, że gdy zmienna **x** ma wartość mniejszą niż **7**, to pętla wykonuje się. Gdy zaś zmienna x przyjmie wartość **7**, to pętla przestanie się wykonywać

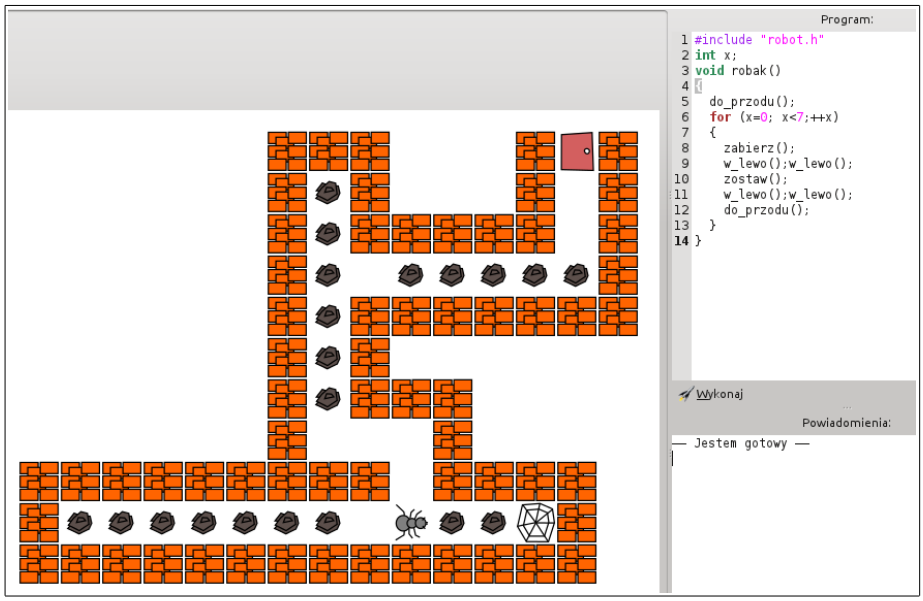
`x++` oznacza to, że po każdym wykonaniu instrukcji pętli wartość zmiennej **x** zwiększana jest o **1**.

Polecenia pozwalające pokonać dolny korytarz kamieni (*Ilustracja 10*) będą więc następujące:

```
int x;
void robak( )
{
  do_przodu( );
  for (x=0; x<7;++x)
    {
      zabierz( );
      w_lewo( );w_lewo( );
      zostaw( );
      w_lewo( );w_lewo( );
      do_przodu( );
    }
}
```

Zapis ten można przetłumaczyć następująco:

**Dla *x* z przedziału od *0* do *6* wykonają się instrukcje zawarte między klamrami.**




*Ilustracja 10: Położenie mrówki po wykonaniu pętli for z kodu zamieszonego wyżej*

# Tworzenie własnych poziomów

Dużą zaletą programu Laby jest możliwość tworzenia własnych poziomów. Ma to znaczenie zwłaszcza wtedy, gdy tempo pracy grupy jest nierówne i jedni kończą pracę szybciej niż drudzy. Z doświadczenia wiem też, że przerobienie dostępnych po instalacji Laby poziomów nie wystarcza do nabycia przez uczniów biegłości w posługiwaniu się składnią danego języka. Warto więc stworzyć sobie nowe plansze treningowe, które pozwolą uczniom utrwalić przyswojone umiejętności. W szkole praktykujemy też tworzenie poziomów sprawdzianowych z podziałem na grupy.

Nowy poziom tworzy się w edytorze tekstowym zwracając uwagę, czy jest on skonfigurowany tak, by wcięcia były ustawione jako tabulatory, a nie spacje. Kodowanie znaków musi być ponadto ustawione jako utf-8. To bardzo ważne, gdyż w przeciwnym razie plansza nie będzie mogła być odczytana.

```
map:
o o o o o o
o w . . . x
o . . . . o
o . . . . o
o . ↑ . r o
o o o o o o

title:
en_US.UTF-8      Demo
fr_FR.UTF-8      Démo
hu_HU.UTF-8      Demó
pl_PL.UTF-8      Demonstacja

comment:
en_US.UTF-8      This labyrinth
out of this labyrinth. It will
spider web. Click the button t
perform individual steps.
```

*Ilustracja 11: Zawartość pliku poziomu 0.laby*

Niezbędne elementy każdego poziomu to (*Ilustracja 11*):

- **map:** obraz mapy labiryntu
- **title:** tytuł poziomu w różnych językach. Dla języka polskiego należy zamieścić wpis pl_PL.UTF-8. Po tym wpisie należy wcisnąć klawisz **Tab** i dodać tytuł poziomu.
- **comment:** Komentarz do poziomu, np. polecenie dla ucznia. Pisząc komentarz postępujemy podobnie jak w przypadku tytułu poziomu

Początkowo nowy poziom najłatwiej jest tworzyć w oparciu o już te istniejące pamiętając, by niechcący nie nadpisać oryginalnej planszy nowo utworzoną.

Przy tworzenie poziomów stosuje się następujące skróty

o – element muru
x – wyjście
**.** – puste miejsce w labiryncie
r – duży kamień
w – duża pajęczyna
R – losowa zamiana kamieni dużego i małego
W – losowa zamiana pajęczyn dużej i małej
↑ ←↓→ – miejsce i kierunek ustawienia mrówki (znaki te można skopiować np. z innej planszy)

Wyżej wymienione elementy oddziela się spacjami. Umieszczenie dwóch lub więcej liter **W** na planszy powoduje losową zmianę miejsc pajęczyny dużej z małą jak np. w poziomie **3a** (*Ilustracja 12*). Podobnie umieszczenie dwóch lub więcej dużych liter **R** powoduje losową zmianę miejsc dużego i małego kamienia. Ostateczny rysunek planszy powinien mieć kształt prostokąta. Plansze należy zapisać na dysku dając jej nazwę z rozszerzeniem **.laby**. Następnie plik planszy należy wkopiować w trybie roota do katalogu Laby (w ubuntu do **/usr/share/laby/levels** )

```
map:
o o x o o
o . . . o
o W o W o
o . ↑ . o
o o o o o
```

*Ilustracja 12: W tym poziomie pajęczyny będą losowo zamieniać się miejscami*

# Dodatek

Tabele zawierają tylko te instrukcje, które są niezbędne do rozwiązania zadań Laby.

## A. Składnia języka C

| | |
|---|---|
| Deklaracja zmiennej całkowitej | **int** nazwa_zmiennej np.<br>**int** x;<br>**int** c, d;<br>**int** a = 5; przykład deklaracji zmiennej z określeniem jej początkowej wartości |
| Instrukcja warunkowa **if** | **if** (wyrażenie)<br>{<br>blok instrukcji (blok instrukcji jest wykonywany, gdy wyrażenie jest prawdziwe)<br>}<br>**else**<br>{<br>blok instrukcji (blok instrukcji jest wykonywany, gdy wyrażenie jest fałszywe)<br>} |
| Pętla **while** | **while** (warunek)<br>{<br>instrukcje (instrukcje te będą się wykonywać, gdy warunek będzie prawdziwy)<br>} |
| Pętla **for** | **for** (wyrażenie1; wyrażenie2; wyrażenie3)<br>{<br>instrukcje (wyrażenie 1 to inicjacja licznika,<br>wyrażenie 2 - jego prawdziwość decyduje o wykonaniu pętli<br>wyrażenie 3 to zwiększenie licznika)<br>} |
| Deklaracja funkcji | **void** nazwa()<br>{instrukcje } |

## B. Składnia języka python

**Uwaga!**

W języku **python** ważne są wcięcia robione **klawiszem tabulacji**. To one decydują o tym, które instrukcje dotyczą np. pętli . W poniższym przykładzie 3 instrukcje wykonują się w pętli, co zostało zaakcentowane wcięciami..

```
while zobacz() == Kamyk:
    zabierz()
    w_lewo()
   do_przodu()
ucieknij()
```

| | |
|---|---|
| Deklaracja zmiennej całkowitej | Zmiennych nie trzeba deklarować w sposób jawny |
| Instrukcja warunkowa **if** | **if** (wyrażenie) **:**<br><br>blok instrukcji (blok instrukcji jest wykonywany, gdy wyrażenie jest prawdziwe)<br>**else :**<br><br>blok instrukcji (blok instrukcji jest wykonywany, gdy wyrażenie jest fałszywe) |
| Pętla **while** | **while** (warunek) **:**<br><br>instrukcje (instrukcje te będą się wykonywać, gdy warunek będzie prawdziwy) |
| Pętla **for** | **for** zmienna **in** range(a,b,c) **:**<br>instrukcje (zmienna to nazwa zmiennej,<br>range- określa zakres wartości zmiennej od wartości **a** do wartości **b,** co wartość **c**. Można podać tylko wartość **a**, wtedy zmienna będzie przyjmowała wartości od 0 do a co 1.<br>Po każdym wykonaniu pętli wartość zmiennej będzie rosła |
| Deklaracja funkcji | **def** nazwa **:**<br>instrukcje |