Język maszyn

Jakub Jernajczyk
Akademia Sztuk Pięknych
im. E. Gepperta we Wrocławiu
Wydział Grafiki i Sztuki Mediów

Bartłomiej Skowron
Uniwersytet Wrocławski
Wydział Nauk Społecznych

Jarosław Drapała
Politechnika Wrocławska
Wydział Informatyki i Zarządzania

Akademia Młodych Uczonych i Artystów

Autorzy

Jakub Jernajczyk – rozdział pierwszy (KoLo)
Artysta grafik, matematyk. Działalność: sztuka mediów, programowanie multimediów, związki sztuki z nauką, popularyzacja nauki, badanie zjawisk dyskretnych.

Bartłomiej Skowron – wstęp i redakcja
Filozof, matematyk. Działalność: ontologia, teoria całości i części, matematyczne modelowanie struktur filozoficznych, logika i etyka.

Jarosław Drapała – rozdział drugi (mrówka)
Informatyk, statystyk. Działalność: modelowanie matematyczne, systemy złożone, metody numeryczne, sieci neuronowe, systemy wspomagania decyzji.

Autorzy są członkami **Akademii Młodych Uczonych i Artystów**. Akademia jest miejscem interdyscyplinarnej, swobodnej i żywej wymiany intelektualnej naukowców i artystów. Powstała we Wrocławiu w 2010 roku, jako pierwsza tego typu inicjatywa w Polsce. Pośród wielości specjalizacji i wielości perspektyw poznawczych, idee jedności wiedzy i jedności poznania stały się ideami całościującymi Akademię. W szczególności jednym z zadań Akademii stało się przełamywanie społecznie utwierdzonego podziału humanista-ścisłowiec. Członkowie Akademii tworzą prawdziwie interdyscyplinarne grupy badawcze (np. psychiatria i informatyka, sztuki plastyczne i informatyka, filozofia i matematyka) i prowadzą w swoim gronie projekty naukowe i naukowo-artystyczne.

www.akademia.wroc.pl

Projekt koordynuje Wrocławskie Centrum Doskonalenia Nauczycieli
Zespół koordynujący: e-mail: jezyk.maszyn@gmail.com
Jerzy Wachowicz – koordynator, tel. (71)796 45 69
Henryk Ożóg, tel. (71)795 50 81

Wrocław 2013

Spis treści

Wstęp

Język maszyn

O tym jak ważną rolę pełnią w naszym świecie maszyny a w szczególności komputery, nie trzeba nikogo przekonywać. Główną motywacją dla przedmiotu język maszyn jest dbałość o jakość kontaktu maszynami. Użytkownik maszyn powinien znać reguły nimi rządzące. Idzie zatem o świadome użytkowanie maszyn. Świadomość ta pozwala na efektywne i twórcze ich wykorzystanie.

Programowanie w obrębie współczesnej wiedzy zajmuje szczególne miejsce. Z jednej strony rozwija podstawowe umiejętności logiczno-algorytmiczno-matematyczne, z drugiej zaś programowanie ma praktyczne zastosowanie w życiu społecznym, gospodarczym i biznesowym. Trudno o taką sferę społeczną, naukową czy artystyczną, w której pewne techniki programistyczne nie odgrywałyby żadnej roli. Ponadto, w obrębie wiedzy programistycznej rozważa się wiele ważnych poznawczo pojęć, choćby takich jak: znak, język, wyrażenie, funkcja, algorytm, zmienna, złożoność obliczeniowa, rozstrzygalność itd. Swoisty teoretyczno-praktyczny charakter programowania sprawia, że staje się ono ważnym i potrzebnym a nawet koniecznym elementem edukacji człowieka.

Wartość poznawcza przedmiotu język maszyn nie polega tylko, jak można by domniemywać, na nauce języka formalnego. Powszechne rozumienie funkcji języka jako takiego ogranicza się do komunikacji. Jest to oczywiście bardzo ważna funkcja języka, lecz nie jedyna. Język należy rozważać w trzech jego – przenikających się nawzajem – aspektach: syntaktycznym, semantycznym oraz pragmatycznym. Syntaksa języka to podstawowe symbole i reguły tworzenia poprawnych wyrażeń w tym języku. Semantyka to sfera znaczeń języka, sfera przedmiotowa do której znaki się odnoszą, którą oznaczają, którą wyrażają. Warstwa znaczeń jest niemniej ważna niż sfera składni. To właśnie sfera znaczeń umożliwia funkcje komunikacyjne języka, jest niejako podstawą komunikacji. Sfera pragmatyczna zaś języka to przestrzeń jego użytkownika, przestrzeń w której rozumiemy dane wyrażenia i na ich podstawie podejmujemy działania.

Język maszyn nie jest tak bardzo skomplikowanym językiem jak języki naturalne, niemniej wszystkie trzy wymienione aspekty są w nim obecne i łatwo zauważalne. Syntaksa to sfera znaków i wyrażeń. Nie należy przywiązywać się

do jej formy, gdyż może być on inna w różnych językach programowania. Zdecydowaliśmy, aby na początkowych etapach nauczania sztuki programowania wybrać takie symbole, których sama forma wskazuje na ich znaczenie, w szczególności znak ruchu „w prawo" na scenie oznaczyliśmy symbolem „→". Symbol ten jest jednocześnie nośnikiem informacji o swoim działaniu. W przypadku bardziej złożonych problemów piktogramy tego typu zastąpiliśmy bardziej abstrakcyjnymi symbolami, zbliżając się w ten sposób do klasycznego programowania. Wobec różnorodności języków programowania, proponujemy aby już od pierwszych etapów nauczania sztuki programowania nie przywiązywać się do określonej formy zapisu, do określonych symboli, tylko kształtować postawę otwartości na płynną zmianę nawyków co do używania takich a nie innych znaków. Wartościowym przykładem dydaktycznym może być zaprezentowanie programów zakodowanych w kilku różnych językach, ale dających taki sam efekt. Uczniowie widzą dzięki temu, że choć formy zapisu są różne, struktura logiczna tych programów jest w zasadzie taka sama.

Znaki, napisy, wyrażenia odpowiednio do potrzeb i efektywności mogą i powinny się zmieniać. Sfera znaczeń (semantyka), sfera procedur i konstrukcji stojących za tymi napisami, nie jest już aż tak zmienna i umowna, niemniej jest niezwykle złożona. Jedną i tę samą procedurę można wyrazić w różny sposób – nie tracąc jej tożsamości. Tę samą procedurę można opisać przy pomocy różnych wyrażeń w jednym języku bądź różnych wyrażeń pochodzących z różnych języków formalnych. Można jednak również odnieść się do tej procedury przy pomocy znaków mniej abstrakcyjnych, np. piktogramów. Ten sposób odnoszenia się do przedmiotu znaku został potraktowany jako wzorcowy w pierwszych etapach nauczania sztuki komunikacji z maszyną. Odpowiednim komendom odpowiadają schematy, rysunki. Rysunki te są modelami formuł języka, są tym do czego komendy się odnoszą, tym co one oznaczają. Sfera znaczeń w tym sensie jest dla nas szczególnie ważna a nawet kluczowa. Uwaga uczącego się programowania ucznia powinna być właśnie nastawiona przede wszystkim na przedmiot, na to „o co chodzi", a nie na takie-a-nie-inne napisy i znaczki formalne. Jest prawdą, że komunikacja z maszyną odbywa się na poziomie znaków i napisów. Efektywna komunikacja wymaga jednak zrozumienia znaczeń nadanych owym znakom i napisom. W tym też sensie nazwanie sztuki programowania tylko „rzemiosłem", jest krzywdzącym uproszczeniem.

Trzecią sferą języka jest pragmatyka. Reguły składni i znaczenie to za mało,

aby mówić w pełni o języku. Potrzebny jest użytkownik. Przestrzeń pomiędzy użytkownikiem a składnią i znaczeniem w przypadku sztuki programowania przybiera różne formy. Na początku przygody z programowaniem może być nią kartka papieru, tablica czy odpowiednio przygotowana podłoga w klasie. Na dalszym etapie wprowadzane są konkretne środowiska programistyczne. W ramach języka maszyn proponujemy dwa takie środowiska: KoLo operujące „graficznym językiem strzałek" oraz Octave z autorskim narzędziem dydaktycznym w postaci programu `mrowkaGo`. Twórcą środowiska KoLo jest Jakub Jernajczyk, autorem programu `mrowkaGo` jest Jarosław Drapała.

Technika nauczania przyjęta w tym projekcie, polega na odsłanianiu coraz bardziej zaawansowanych poziomów programowania. Punktem wyjścia jest intuicyjne operowanie na obiektach, przypominających przedmioty rzeczywiste. W kolejnych etapach kształcenia przechodzimy do coraz bardziej abstrakcyjnych, a przy tym dających coraz większe możliwości, sposobów komunikacji z maszyną.

Już na tym początkowym etapie chcielibyśmy, aby nauka programowania nie polegała – jak to często bywa – wyłącznie na zapisywaniu w danym języku gotowych algorytmów.

Ważną poznawczo i niezwykle twórczą częścią programowania jest odnalezienie sposobu rozwiązania danego problemu. Proces rozwiązywania problemu polega zazwyczaj na rozłożeniu go na podproblemy, znalezieniu odpowiedniego algorytmu oraz jego optymalizacji. W zaproponowanym materiale porządek nauczania odpowiada porządkowi poznania. Oznacza to, że punktem wyjścia jest sam problem a celem kod programu (rozwiązanie problemu). Dopiero w trakcie nauki uczeń stopniowo odkrywa rolę i istotę algorytmów. Algorytmika nie stanowi tu więc początku nauczania, choć wciąż pozostaje istotnym elementem języka maszyn.

W trakcie zajęć uczniowie, jak już wspomnieliśmy, będą ćwiczyć umiejętności logiczno-matematyczne a w tym umiejętności analizowania problemów, szukania rozwiązań (czyli też stawiania hipotez i wyciągania wniosków) oraz optymalizacji tych rozwiązań. Nierzadko też umiejętności te ćwiczone będą w twórczy sposób, wymaga tego bowiem wieloaspektowość i złożoność problemów programistycznych. Jeśli dodatkowo weźmiemy pod uwagę wysoką informatyzację społeczną, czyli m.in. fakt, że problemy algorytmiczno-programistyczne pojawiają się niemalże w każdej dziedzinie życia, to z powodzeniem i wbrew powszechnie panującym opiniom, można nazwać ten przedmiot humanistycznym. Umiejętności rozumnej ana-

lizy problemów, optymalizacji rozwiązań itd., są bowiem umiejętnościami typowo ludzkimi, maszyny zaś są tylko narzędziem i wsparciem tych ludzkich działań.

Skrypt ten składa się zasadniczo z dwóch części. Zadaniem pierwszej jest wyrobienie u uczniów podstawowych intuicji programistycznych w oparciu o uproszczony „obrazkowy" język programowania. W drugiej części stopniowo wprowadzamy klasycznie rozumiane kodowanie programu. W jawny sposób wprowadzane są podstawowe konstrukcje programistyczne: zmienne, funkcje, instrukcje warunkowe i pętle.

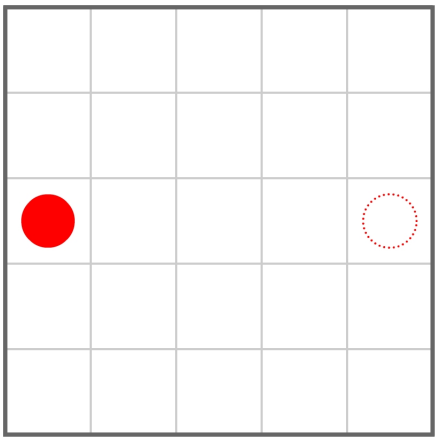
Skrypt ma charakter otwarty i będzie rozbudowywany o nowe elementy. Dotyczyć one będą zarówno technik programowania, jak i propozycji wykorzystania umiejętności programowania w różnych dziedzinach życia, sztuki i nauki. Na treść kolejnych wydań skryptu duży wpływ będą miały rezultaty pilotażu przeprowadzonego we wrocławskich szkołach podstawowych i gimnazjach. Autorzy spodziewają się wręcz, że w trakcie pracy nauczycieli z uczniami pojawią się nowe pomysły i propozycje uzupełnień.

1 Do celu! Programowanie ruchu kółka przy pomocy strzałek

Aby zapoznać się z podstawami programowania, nie musimy posługiwać się abstrakcyjnym kodem. Graficzny język programowania, w którym sens każdego znaku jest oczywisty, pozwala skupić się uczniowi tylko na rozwiązaniu problemu. Problemy, które będziemy rozwiązywać w tym rozdziale z założenia są bardzo proste. Polegają one na doprowadzeniu kółka do wyznaczonego celu. Jednak te proste i klarownie sformułowane wytyczne, pozwolą nam w możliwie przystępny sposób, wprowadzić prototypy podstawowych pojęć i konstrukcji programistycznych, takich jak zmienne, procedury, pętle, czy instrukcje warunkowe. Dodajmy, że pomysł graficznego języka programowania opiera się na tzw. *myśleniu wzrokowym*, które stanowi bazową formę ludzkiego poznania.

1.1 Zapoznanie z elementami na scenie

Obszar, na którym widoczne będą efekty naszych działań, nazywać będziemy *sceną*. Scena podzielona jest na kwadratowe pola. W najprostszym wariancie zaczynamy od sceny o rozmiarze 5×5 pól:



Na starcie w jednym z pól umieszczony jest obiekt (czerwone kółko), który za pomocą odpowiednich instrukcji będziemy starali się doprowadzić do celu (wykropkowany okrąg).

- ● – obiekt, który będziemy przemieszczać
- ○ – cel, do którego obiekt będzie zmierzał

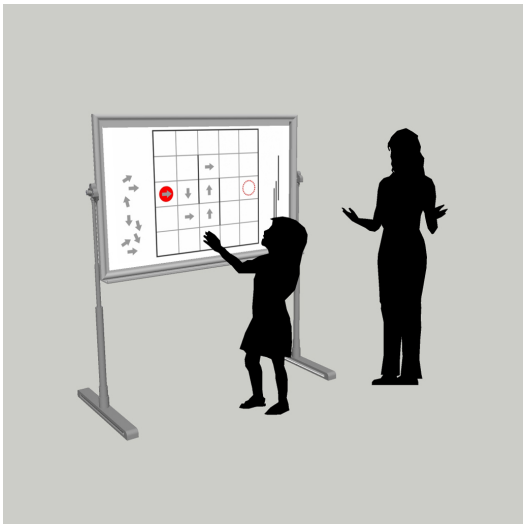
Scena może zostać zaprezentowana uczniom w wielu postaciach:

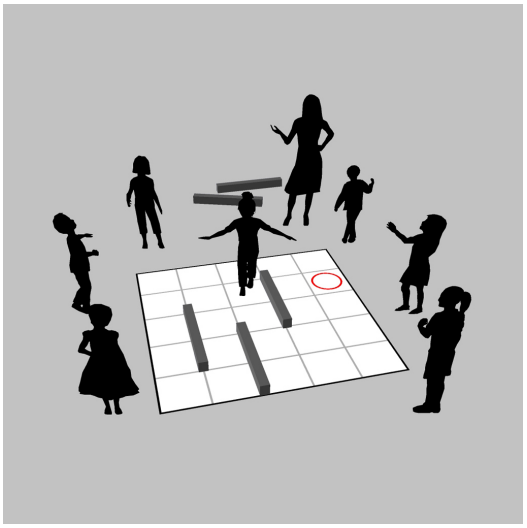
(a) na kartkach A4 – każdy uczeń dostaje własne kopie sceny, na których może rysować symbole; sceny na papierze mogą również służyć do testowania postępów uczniów (kartkówki);

(b) na tablicy magnetycznej – nauczyciel dysponuje specjalnie zaprojektowaną tablicą, na której może umieszczać magnetyczne obiekty (kółka, strzałki, itd.);

(c) na macie – nauczyciel rozkłada w sali dużą matę z wyrysowaną sceną, jeden z uczniów wciela się w rolę sterowanego obiektu (czerwonego kółka), reszta zaś steruje nim wydając odpowiednie komendy;

(d) na komputerze – uczniowie wprowadzają odpowiednie symbole przy pomocy interfejsu; wersja komputerowa sceny pozwala wykonać zaprogramowany ciąg poleceń.

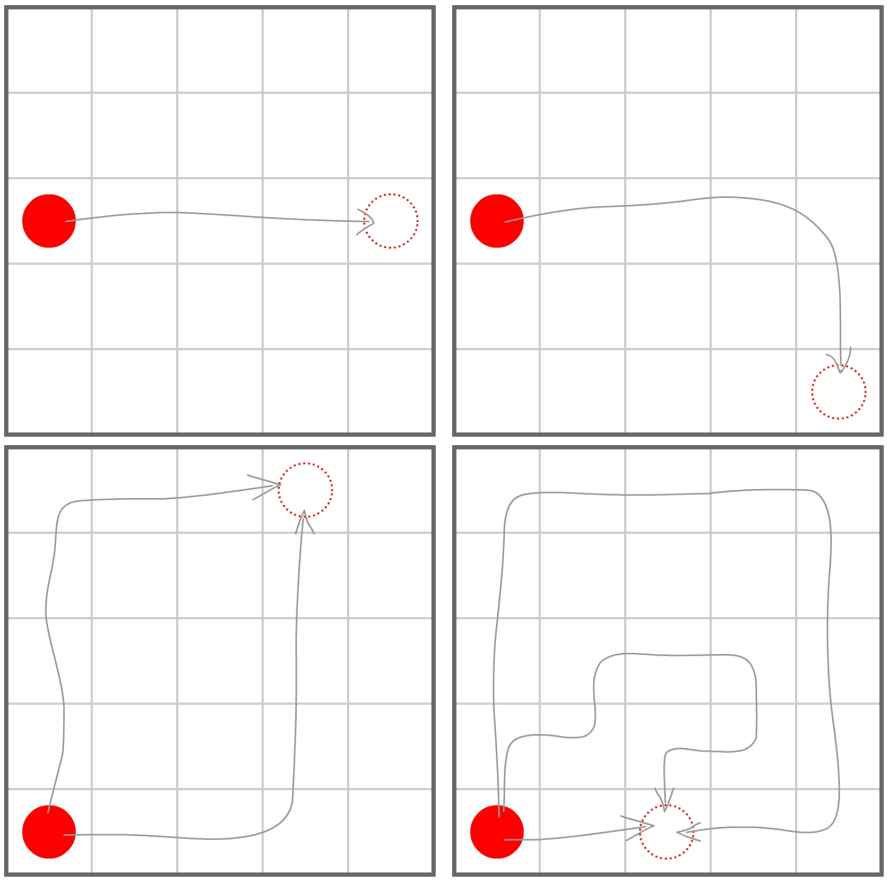




1.2 Rysowanie ścieżki kółka do celu

Ćwiczenie 1.0. Wyrysuj drogę kółka do celu.

To wstępne ćwiczenie polega na naszkicowaniu drogi obiektu (czerwonego kółka) do celu. Można je zrealizować na tablicy lub na kartkach A4. Poniżej przedstawiamy możliwe ćwiczenia z rozwiązaniami.



Na trzecim i czwartym obrazku widać, że ścieżkę kółka do celu można wyznaczyć na wiele różnych sposobów. Większość proponowanych w tym rozdziale ćwiczeń ma wiele poprawnych rozwiązań. Nie można zatem opracować gotowego zestawu poprawnych odpowiedzi. Dlatego też nauczyciel musi z uwagą prześledzić wszystkie rozwiązania proponowane przez uczniów. Już na tym etapie warto zwracać uwagę na potrzebę optymalizacji rozwiązania danego problemu. Wiele rozwiązań jest dobrych, ale nie wszystkie są optymalne.

1.3 Instrukcje. Przeprowadzanie kółka do celu krok po kroku.

Prezentowane w tej sekcji ćwiczenia realizowane mogą być przez uczniów na kartkach, na tablicy magnetycznej lub grupowo na rozłożonej macie.

Do przemieszczania obiektu służą następujące instrukcje (strzałki):

$$\rightarrow \quad \leftarrow \quad \downarrow \quad \uparrow \tag{1}$$

$\rightarrow$ oznacza ruch o jedno pole w prawo

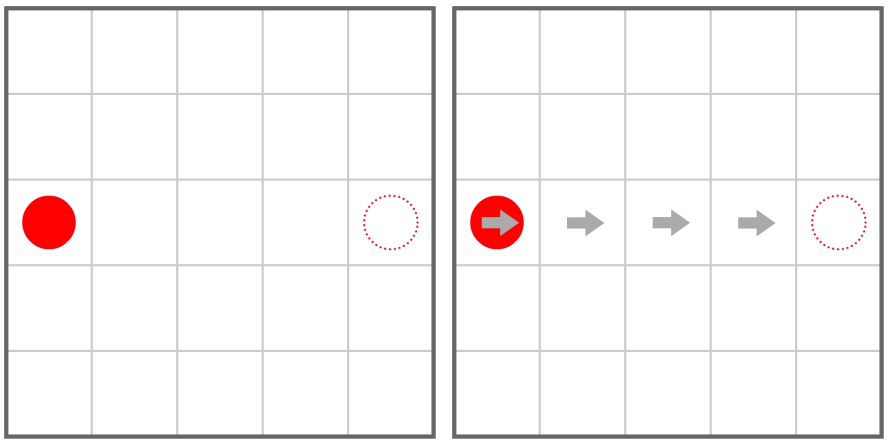
$\leftarrow$ oznacza ruch o jedno pole w lewo

$\downarrow$ oznacza ruch o jedno pole w dół

$\uparrow$ oznacza ruch o jedno pole w górę

Ćwiczenie 1.1. Za pomocą instrukcji $\rightarrow$ przeprowadź kółko do celu.

Pierwsze programistyczne ćwiczenie polega na przeprowadzeniu kółka do celu, który jest oddalony o 4 pola w linii prostej. Umieszczenie na początku instrukcji symbolu poruszającego się obiektu oznacza, że następujące po tym symbolu polecenia dotyczą tego właśnie obiektu. Polecenia wydajemy dla czerwonego kółka, zatem instrukcje nasze poprzedzać będziemy znakami „●:”.

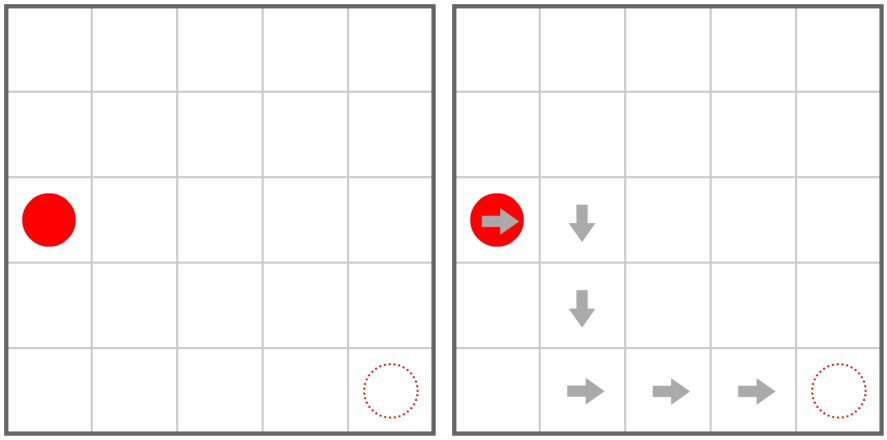


Zadanie to najprościej można zrealizować wpisując następujące instrukcje:

$$\bullet :\rightarrow\rightarrow\rightarrow\rightarrow \tag{2}$$

Powyższą sekwencję czytamy: „kółko ma: iść w prawo, iść w prawo, iść w prawo, iść w prawo".

Ćwiczenie 1.2. Za pomocą instrukcji $\rightarrow$ i $\downarrow$ przeprowadź kółko do celu.

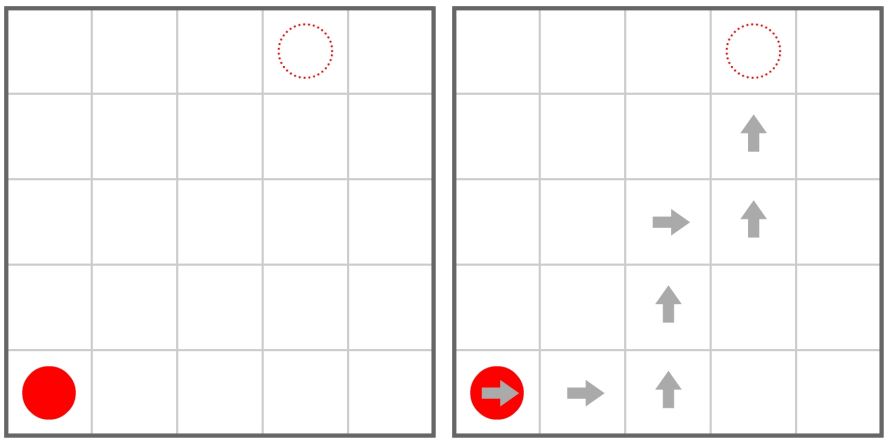


Jednym z poprawnych rozwiązań jest ciąg instrukcji:

$$\bullet :\rightarrow\downarrow\downarrow\rightarrow\rightarrow\rightarrow \tag{3}$$

Powyższą sekwencję czytamy: „kółko ma: iść w prawo, iść w dół, iść w dół, iść w prawo, iść w prawo, iść w prawo".

Ćwiczenie 1.3. Za pomocą instrukcji → i ↑ przeprowadź kółko do celu.



Jednym z poprawnych rozwiązań jest ciąg instrukcji:

$$\bullet : \rightarrow\rightarrow\uparrow\uparrow\rightarrow\uparrow\uparrow \tag{4}$$

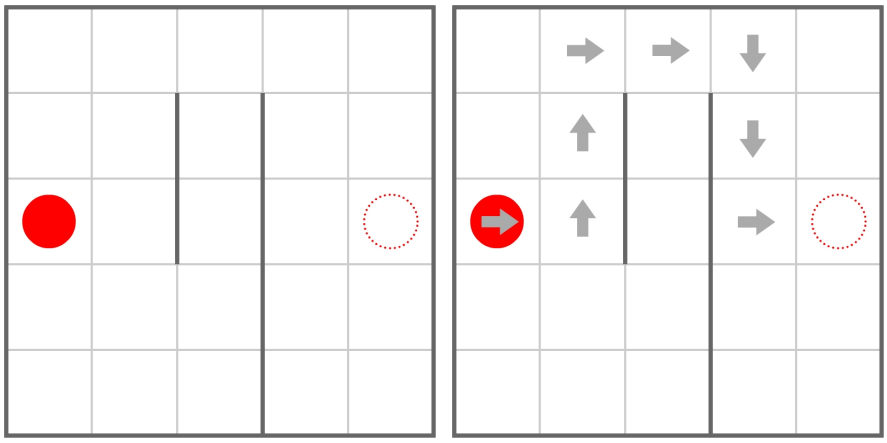
Powyższą sekwencję czytamy: „kółko ma: iść w prawo, iść w prawo, iść w górę, iść w górę, iść w prawo, iść w górę, iść w górę".

Przedstawione wyżej ćwiczenia można powtórzyć całą grupą na podłodze w klasie. Jeden z uczniów wciela się w rolę kółka i przemieszcza zgodnie z instrukcjami dyktowanymi przez resztę grupy. Dodatkowo można zliczać wykonane kroki (instrukcje) w celu wybrania optymalnej – złożonej z najmniejszej liczby kroków ścieżki (takich ścieżek może być kilka). Do zliczania kroków mogą służyć specjalne przedmioty, np. żetony. „Uczeń-obiekt" trzyma w ręku określoną liczbę żetonów i za każdy swój krok (wejście na sąsiadujące pole) płaci jednym żetonem. Grupa musi zatem znaleźć najlepszą, czyli najtańszą i najefektywniejszą drogę obiektu do celu.

1.4 Przeszkody (ściany)

Teraz na scenie pojawiają się przeszkody, które ograniczają swobodę przemieszczania obiektu. Kółko nie może przejść przez przeszkodę (ścianę), tylko musi ją ominąć.

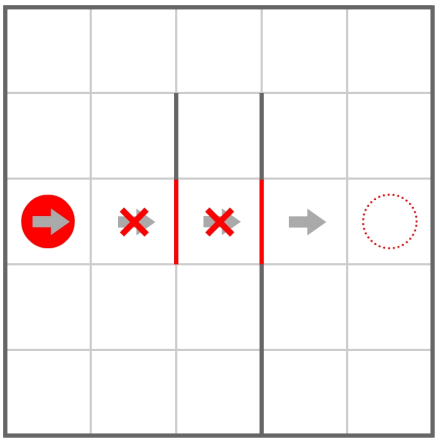
Ćwiczenie 1.4. Za pomocą instrukcji $\rightarrow$, $\uparrow$ i $\downarrow$ przeprowadź kółko do celu, nie przechodząc przez ściany.



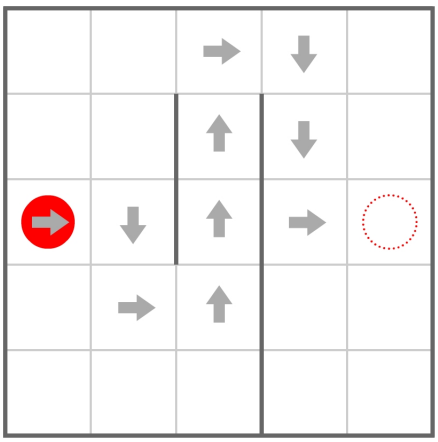
Jednym z poprawnych rozwiązań jest ciąg instrukcji:

$$\bullet : \rightarrow\uparrow\uparrow\rightarrow\rightarrow\downarrow\downarrow\rightarrow \tag{5}$$

Kółko nie może przechodzić przez ściany, ani wychodzić poza scenę. Instrukcja wymuszająca na kółku przejście przez przeszkodę, traktowana jest jako błąd.

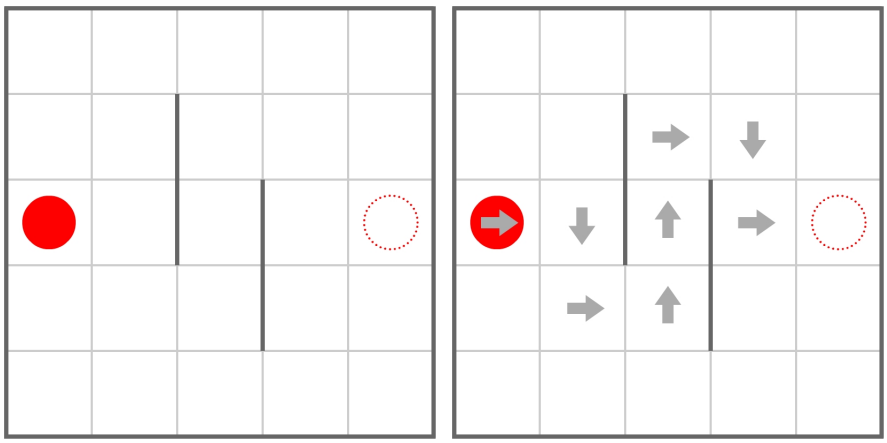


Poniższe zaś rozwiązanie jest poprawne, ale liczba wykonanych kroków jest większa (czyli jest ono mniej wydajne):



● :→↓→↑↑↑→↓↓→ (6)

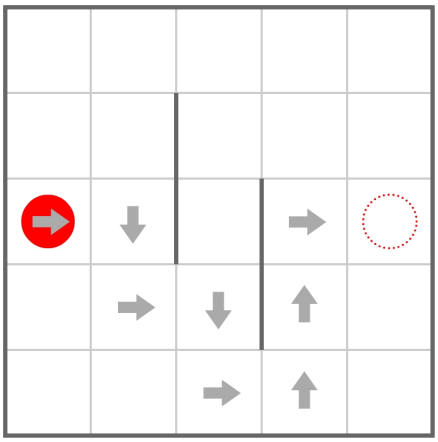
Ćwiczenie 1.5. Za pomocą instrukcji $\rightarrow$, $\uparrow$ i $\downarrow$ przeprowadź kółko do celu, nie przechodząc przez ściany.



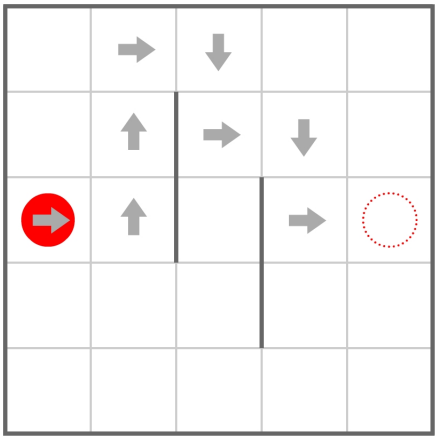
Jednym z poprawnych rozwiązań jest ciąg instrukcji:

$$\bullet : \rightarrow\downarrow\rightarrow\uparrow\uparrow\rightarrow\downarrow\rightarrow \tag{7}$$

Poniższe rozwiązania również są poprawne i składają się z tej samej liczby instrukcji (kroków):



$$\bullet : \rightarrow\downarrow\rightarrow\downarrow\rightarrow\uparrow\uparrow\rightarrow \tag{8}$$



● :→↑↑→↓→↓→ (9)

Ćwiczenia 1.4. i 1.5. można przeprowadzić na tablicy, na kartkach i na podłodze.

1.5 Praca z komputerem

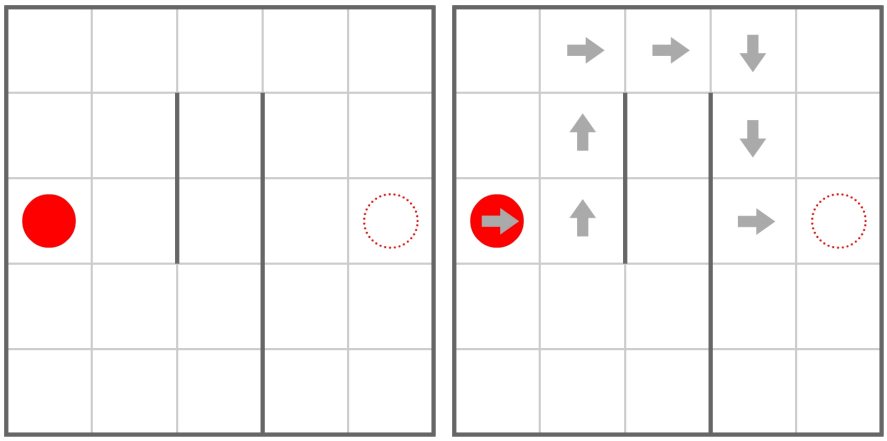
Wszystkie powyższe ćwiczenia należy powtórzyć z pomocą komputera. Służy do tego specjalnie opracowana aplikacja KoLo, operująca graficznym językiem programowania. Przeprowadzenie ćwiczeń na komputerze ma kilka zalet:

1. Każdy uczeń może prześledzić animację ruchu kółka wynikającą z zaprogramowanej przez siebie sekwencji ruchów.
2. Animację zaprogramowanej sekwencji można prześledzić w całości lub krok po kroku.
3. Doprowadzenie obiektu do wyznaczonego celu nagradzane jest pozytywnym sygnałem dźwiękowym i wizualnym, co zwalnia nauczyciela z obowiązku sprawdzania rozwiązania krok po kroku.
4. Komputer nie pozwoli na przejście kółka przez przeszkodę (ścianę lub krawędź sceny) i zasygnalizuje błąd programu.

Dedykowany edytor scen KoLes pozwala na tworzenie wielu nowych zadań.

1.6 Powtórzenia pojedynczych kroków

Ćwiczenie 1.6. Za pomocą instrukcji $\rightarrow$, $\uparrow$ i $\downarrow$ przeprowadź kółko do celu.



Pozornie nie ma w tym zadaniu nic nowego. Poprawne rozwiązanie podane już było w ćwiczeniu 1.4.

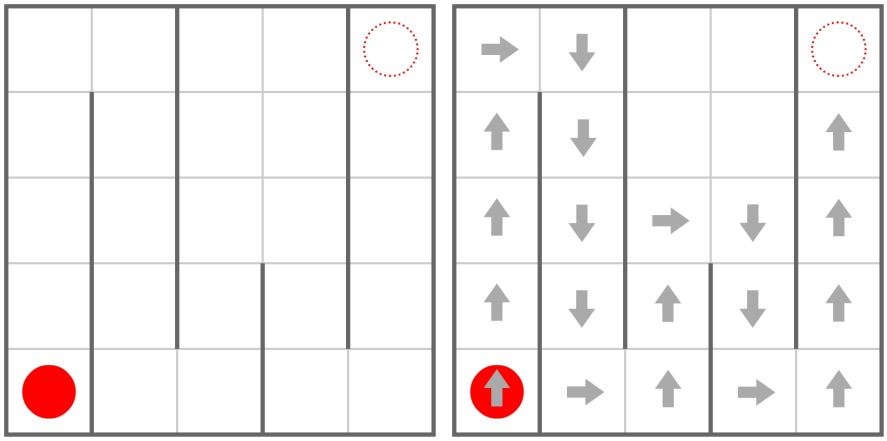
$$\bullet : \rightarrow\uparrow\uparrow\rightarrow\rightarrow\downarrow\downarrow\rightarrow \tag{10}$$

Warto jednak zwrócić uwagę na powtarzające się po sobie instrukcje.

Zapis programu można skrócić, podając po strzałce liczbę, która określa ile razy ta strzałka ma się powtórzyć. Jeśli przy strzałce nie pojawi się żadna liczba, oznacza to, że występuje ona tylko jeden raz. Formule (10) odpowiadał będzie zatem zapis:

$$\bullet : \rightarrow\uparrow_2 \rightarrow_2 \downarrow_2 \rightarrow \tag{11}$$

Ćwiczenie 1.7. Zapisz program przejścia kółka do celu w formie pełnej oraz skróconej (z powtórzeniami).



Zapis pełny:

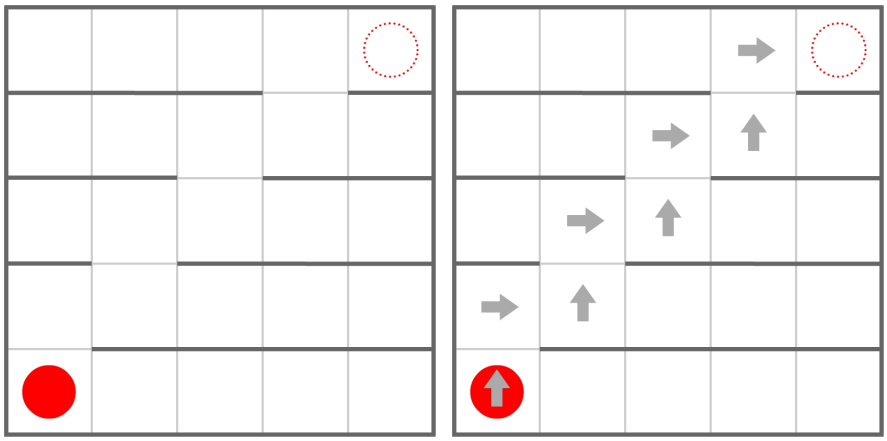
$$\bullet : \uparrow\uparrow\uparrow\uparrow \rightarrow \downarrow\downarrow\downarrow\downarrow \rightarrow \uparrow\uparrow \rightarrow \downarrow\downarrow \rightarrow \uparrow\uparrow\uparrow\uparrow \tag{12}$$

Zapis skrócony (z liczbą powtórzeń instrukcji):

$$\bullet : \uparrow_4 \rightarrow \downarrow_4 \rightarrow \uparrow_2 \rightarrow \downarrow_2 \rightarrow \uparrow_4 \tag{13}$$

1.7 Powtórzenia sekwencji kroków – paczki

Ćwiczenie 1.8. Przeprowadź kółko do celu.



Najefektywniejszym rozwiązaniem tego zadania jest ciąg instrukcji:

$$\bullet : \uparrow \rightarrow \uparrow \rightarrow \uparrow \rightarrow \uparrow \rightarrow \tag{14}$$

Zapytajmy teraz uczniów, czy widzą tu coś szczególnego. Być może ktoś zauważy, że w powyższym kodzie cztery razy powtarza się sekwencja dwóch symboli „$\uparrow\rightarrow$”. Gdybyśmy zatem mogli jakoś spakować razem te dwie instrukcje ($\uparrow$ i $\rightarrow$), wystarczyłoby taką paczkę powtórzyć czterokrotnie, by uzyskać powyższy program.

W tym właśnie celu wzbogacamy nasz graficzny język programowania o kolejny symbol „$\boxplus$”, oznaczający paczkę, w której przechowywane mogą być sekwencje instrukcji. Jeśli teraz zdefiniujemy zawartość paczki jako:

$$\boxplus = \uparrow \rightarrow \tag{15}$$

i dodatkowo zastosujemy powtórzenia, to nasz kod możemy przedstawić w bardzo skróconej formie:

$$\bullet : \boxplus_4 \tag{16}$$

Zatem rozwiązaniem ćwiczenia będą dwie sekwencje znaków:

$$\begin{array}{l} \boxplus = \uparrow \rightarrow \\ \bullet : \boxplus_4 \end{array} \tag{17}$$

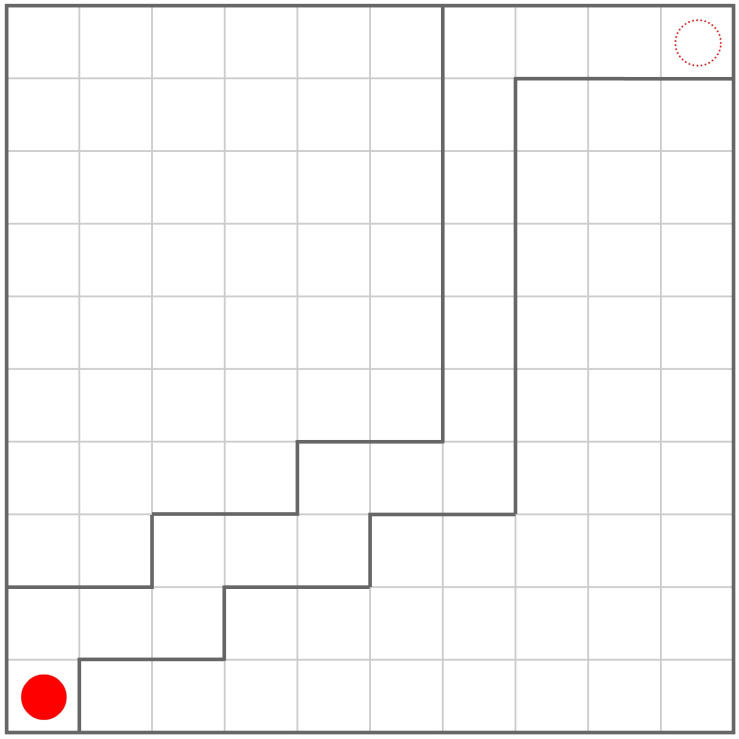
Pod względem wykonywanych instrukcji jest to tożsame z zapisem

$$\bullet : \uparrow \rightarrow \uparrow \rightarrow \uparrow \rightarrow \uparrow \rightarrow \tag{18}$$

jednak zapis z zastosowaniem paczki jest krótszy.

W kolejnych ćwiczeniach pracować będziemy już na większej scenie o wymiarze 10×10 pól. Od tej pory nie będziemy już używać maty ani tablicy magnetycznej. Głównym narzędziem pracy będzie komputer. Pomocne mogą być również wydruki scen na kartkach A4. Od tego momentu uczniowie powinni zapisywać kod programu w liniowej sekwencji znaków pod sceną, nie notując już nic na scenie.

Ćwiczenie 1.9. Przeprowadź kółko do celu. Odszukaj powtarzającą się sekwencję znaków i skróć zapis korzystając z paczki.



Zapis pełny:

$$\bullet : \uparrow \rightarrow \rightarrow \uparrow \rightarrow \rightarrow \uparrow \rightarrow \rightarrow \uparrow\uparrow\uparrow\uparrow\uparrow\uparrow \rightarrow \rightarrow \rightarrow \tag{19}$$

Odnajdujemy powtarzający się wzór „$\uparrow \rightarrow \rightarrow$” i zamykamy go w paczce:

$$\boxplus = \uparrow \rightarrow \rightarrow \tag{20}$$

Tworzymy zapis skrócony z wykorzystaniem paczki:

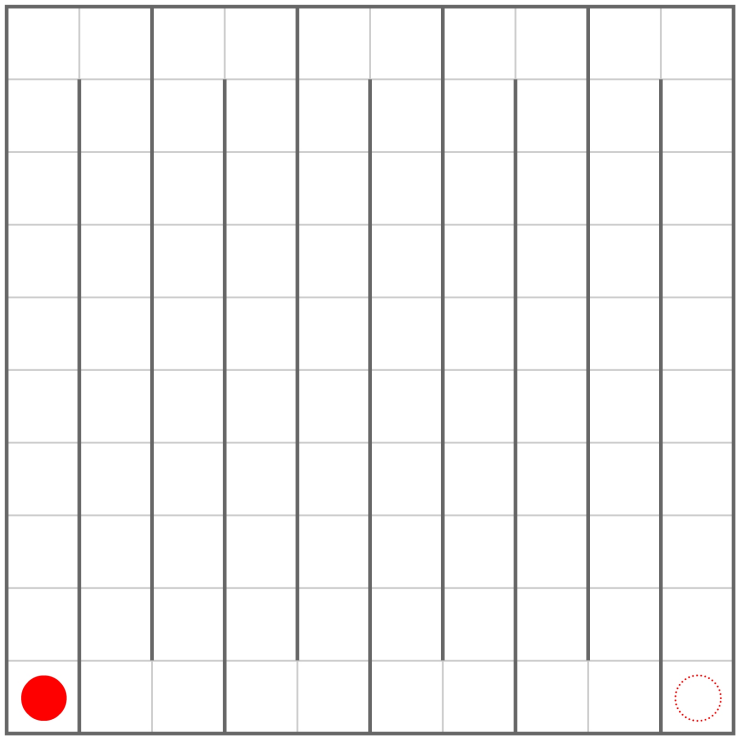
$$\begin{aligned} \boxplus &= \uparrow \rightarrow \rightarrow \\ \bullet &: \boxplus_3 \uparrow_5 \boxplus \rightarrow \end{aligned} \tag{21}$$

Omawiane w tej sekcji paczki stanowią prototyp procedur. Właściwie są one czymś pomiędzy zmiennymi a procedurami. Powtórzenia zaś są uproszczonym modelem pętli.

1.8 Paczki złożone

Wewnątrz paczek możemy również stosować powtórzenia.

Ćwiczenie 1.10. Przeprowadź kółko do celu. Skróć zapis korzystając z paczki.



Zapis skrócony bez paczek:

$$\bullet : \uparrow_9 \rightarrow \downarrow_9 \rightarrow \uparrow_9 \rightarrow \downarrow_9 \rightarrow \uparrow_9 \rightarrow \downarrow_9 \rightarrow \uparrow_9 \rightarrow \downarrow_9 \rightarrow \uparrow_9 \rightarrow \downarrow_9 \tag{22}$$

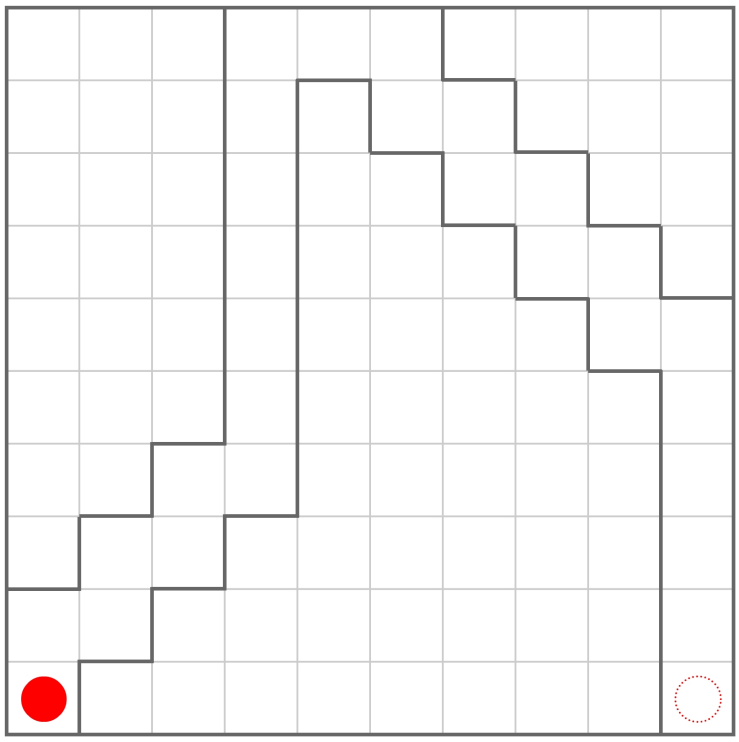
Odnajdujemy powtarzający się wzór „$\uparrow_9 \rightarrow \downarrow_9 \rightarrow$” i zamykamy go w paczce:

$$\boxplus = \uparrow_9 \rightarrow \downarrow_9 \rightarrow \tag{23}$$

Tworzymy zapis skrócony z wykorzystaniem paczki:

$$\begin{aligned} \boxplus &= \uparrow_9 \rightarrow \downarrow_9 \rightarrow \\ \bullet &: \boxplus_4 \uparrow_9 \rightarrow \downarrow_9 \end{aligned} \tag{24}$$

Ćwiczenie 1.11. Przeprowadź kółko do celu. Skróć zapis korzystając z paczek.



Zapis bez paczek:

$$\bullet : \uparrow\rightarrow\uparrow\rightarrow\uparrow\rightarrow\uparrow\uparrow\uparrow\uparrow\uparrow\uparrow\rightarrow\rightarrow\downarrow\rightarrow\downarrow\rightarrow\downarrow\rightarrow\downarrow\rightarrow\downarrow\downarrow\downarrow\downarrow\downarrow \tag{25}$$

Odnajdujemy tu dwa powtarzające się wzory „$\uparrow\rightarrow$” i „ $\rightarrow\downarrow$” i pakujemy je do dwóch różnych paczek:

$$\boxplus = \uparrow\rightarrow \tag{26}$$

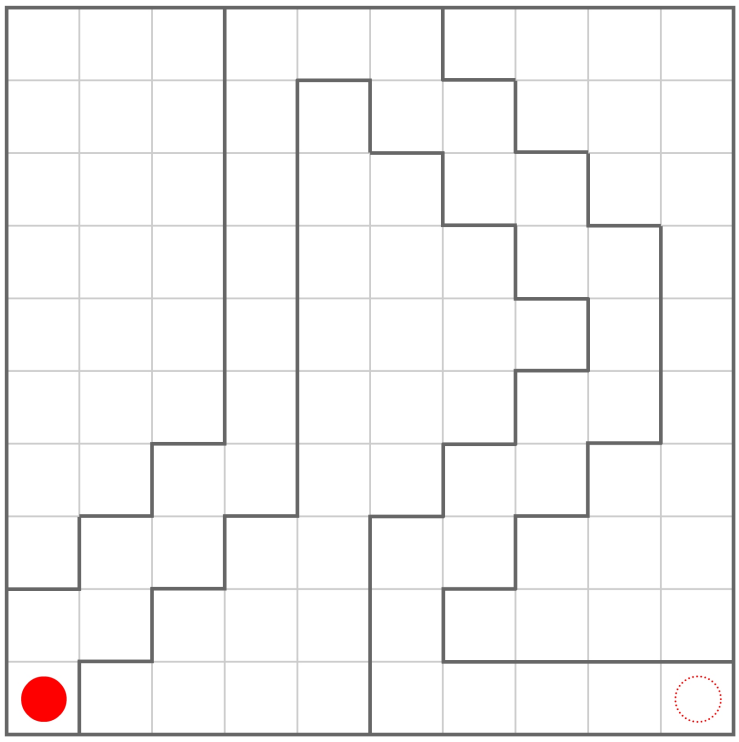
oraz

$$\boxtimes = \rightarrow\downarrow \tag{27}$$

Tworzymy zapis skrócony z wykorzystaniem paczek:

$$\begin{aligned} &\boxplus = \uparrow\rightarrow \\ &\boxtimes = \rightarrow\downarrow \\ &\bullet : \boxplus_3 \uparrow_6 \rightarrow \boxtimes_5 \downarrow_4 \end{aligned} \tag{28}$$

Ćwiczenie 1.12. Przeprowadź kółko do celu. Skróć zapis korzystając z paczek.



Zapis bez paczek:

$$\bullet : \uparrow\rightarrow\uparrow\rightarrow\uparrow\rightarrow\uparrow\uparrow\uparrow\uparrow\uparrow\uparrow\rightarrow\rightarrow\downarrow\rightarrow\downarrow\rightarrow\downarrow\rightarrow\downarrow\downarrow\leftarrow\downarrow\leftarrow\downarrow\leftarrow\downarrow\downarrow\rightarrow\rightarrow\rightarrow\rightarrow \tag{29}$$

Można odnaleźć tu aż trzy powtarzające się wzory „$\uparrow\rightarrow$”, „$\downarrow\rightarrow$” i „$\leftarrow\downarrow$”. Spakujmy je więc do trzech różnych paczek:

$$\boxplus = \uparrow\rightarrow \quad \boxtimes = \downarrow\rightarrow \quad \boxminus = \leftarrow\downarrow \tag{30}$$

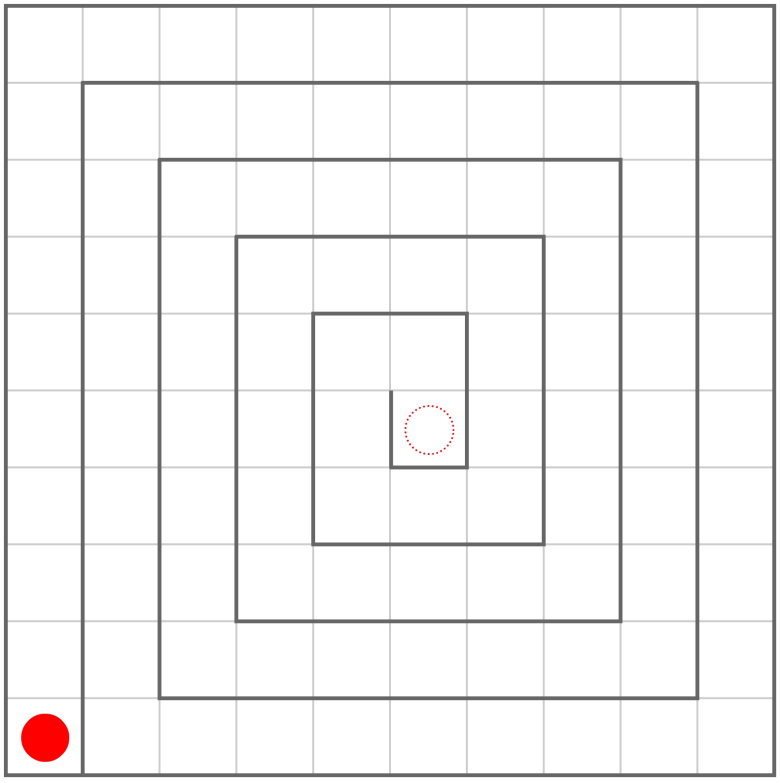
Tworzymy zapis skrócony z wykorzystaniem paczek:

$$\begin{aligned} \boxplus &= \uparrow\rightarrow \\ \boxtimes &= \downarrow\rightarrow \\ \boxminus &= \leftarrow\downarrow \\ \bullet &: \boxplus_3\ \uparrow_6\rightarrow_2\ \boxtimes_3\ \downarrow_2\ \boxminus_3\ \downarrow\rightarrow_4 \end{aligned} \tag{31}$$

Ostatnią linijkę można zakodować również w następujący sposób:

$$\bullet : \boxplus_3 \uparrow_6 \rightarrow_2 \boxtimes_3 \downarrow_2 \boxminus_3 \boxtimes \rightarrow_3 \tag{32}$$

Ćwiczenie 1.13. Przeprowadź kółko do celu. Skróć zapis korzystając z paczki.



Zapis podstawowy:

$$\bullet :\uparrow_9 \rightarrow_9 \downarrow_9 \leftarrow_8 \uparrow_8 \rightarrow_7 \downarrow_7 \leftarrow_6 \uparrow_6 \rightarrow_5 \downarrow_5 \leftarrow_4 \uparrow_4 \rightarrow_3 \downarrow_3 \leftarrow_2 \uparrow_2 \rightarrow \downarrow \tag{33}$$

Jest to przykład, dla którego nie można zdefiniować paczki, przy pomocy poznanych dotąd instrukcji.

1.9 Ruchy warunkowe

Możliwości naszego graficznego języka programowania znacznie się powiększą, jeśli wzbogacimy go o następujące instrukcje strzałek warunkowych:

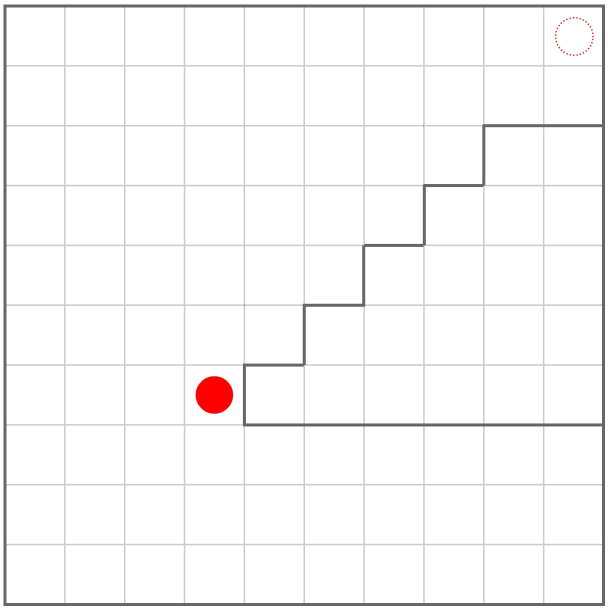
$\rightarrow^{?}$ jeśli to możliwe, idź w prawo

$\leftarrow^{?}$ jeśli to możliwe, idź w lewo

$\downarrow^{?}$ jeśli to możliwe, idź w dół

$\uparrow^{?}$ jeśli to możliwe, idź w górę

Jeśli wykonanie instrukcji warunkowej miałoby spowodować zderzenie kółka z przeszkodą (ścianą), to nie zostanie ona wykonana.



Przeanalizujmy powyższą scenę porównując dwa kody:

1) • : $\rightarrow$

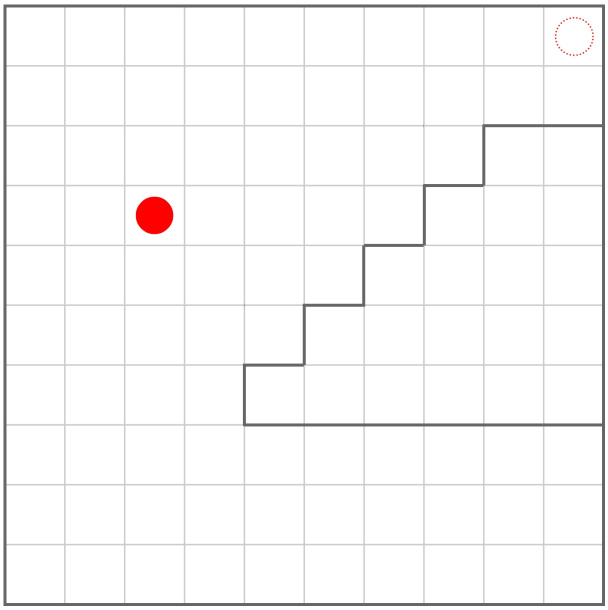
2) • : $\rightarrow^{?}$

Pierwszy kod jest błędny, gdyż wymusza na kółku ruch w prawo, pomimo tego, że na jego drodze znajduje się przeszkoda. W drugim przypadku strzałka warunkowa sprawdza najpierw czy ruch w prawo jest możliwy. Jeśli nie, to kółko nie wykonuje ruchu; pozostaje w miejscu.

Strzałki warunkowe stanowią prototyp ważnej konstrukcji programistycznej, jaką jest instrukcja warunkowa (*if*).

Strzałki warunkowe możemy również stosować razem z powtórzeniami.

Polecenie: $\bullet : \rightarrow_7^?$
oznacza, że kółko będzie siedem razy próbowało przesunąć się w prawo, za każdym razem sprawdzając czy ruch ten jest możliwy.



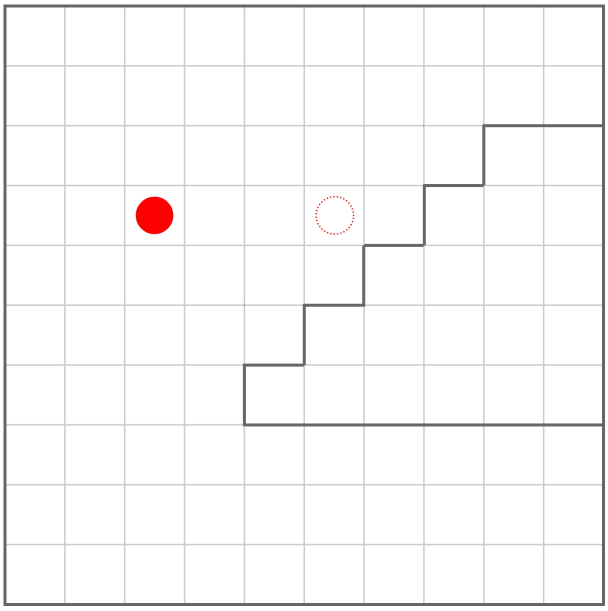
W przypadku tej sceny powyższy kod spowoduje, że kółko przesunie się cztery razy w prawo, potem zaś trzykrotnie stwierdzi, że dalszy ruch w prawo jest niemożliwy i będzie stało w miejscu.

Jeśli choć raz nie uda się przejść dalej, możemy być pewni, że powtarzanie tego samego ruchu warunkowego również zakończy się niepowodzeniem. Dlatego też mo-

żemy bezpiecznie przyjąć, że w takim przypadku kółko przerwie dalsze powtórzenia strzałek warunkowych. Od teraz więc nasze kółko będzie odrobinę mądrzejsze.

Strzałki warunkowe z powtórzeniami są prototypem pętli warunkowej (*while*).

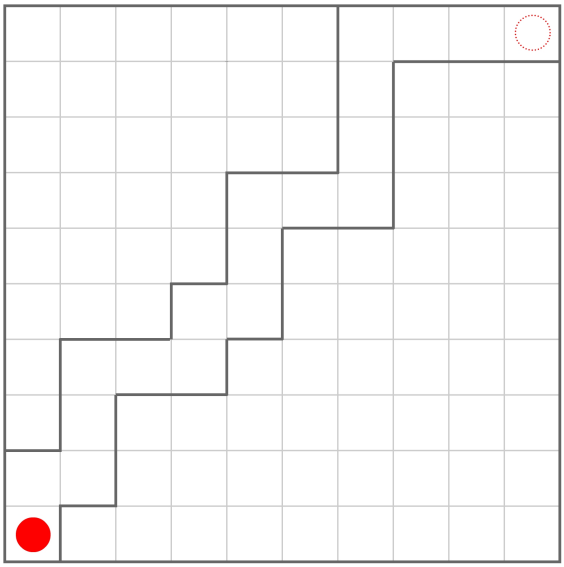
Stosując strzałki warunkowe musimy wprowadzić jeszcze jedno dodatkowe założenie. Odtąd będziemy przyjmować, że dojście kółka do celu (wejście na pole ○) zawsze kończy program, niezależnie od tego, czy zostały wykorzystane wszystkie zakodowane instrukcje ruchu.



Jeśli w powyższej sytuacji zakodujemy następujący ruch kółka: ● $:\rightarrow^{?}_{5} \uparrow \uparrow \rightarrow$ to i tak po trzech krokach w prawo (wykonanych dzięki powtarzanej strzałce warunkowej) program zakończy się sukcesem, gdyż kółko dotrze do celu. Dalsze instrukcje nie będą miały już znaczenia.

Wprowadzone tu założenie nie unieważnia przykładów rozważanych w poprzednich ćwiczeniach. Dotychczas wpisywaliśmy dokładnie tyle instrukcji, ile było potrzeba, aby kółko dotarło do celu, zaś osiągnięcie celu zawsze kończyło program. Teraz stwierdzamy tylko, że nawet jeśli tych instrukcji będzie więcej niż potrzeba, program również wykona się prawidłowo.

Ćwiczenie 1.14. Przeprowadź kółko do celu z wykorzystaniem dwóch strzałek warunkowych: $\uparrow^?$ i $\rightarrow^?$ oraz powtórzeń.



Bez wykorzystania strzałek warunkowych pełny zapis tego ruchu wygląda następująco:

• : ↑→↑↑→→↑→↑↑→→↑↑↑→→→

Widać, że powtarzają się w nim podobne sekwencje instrukcji:

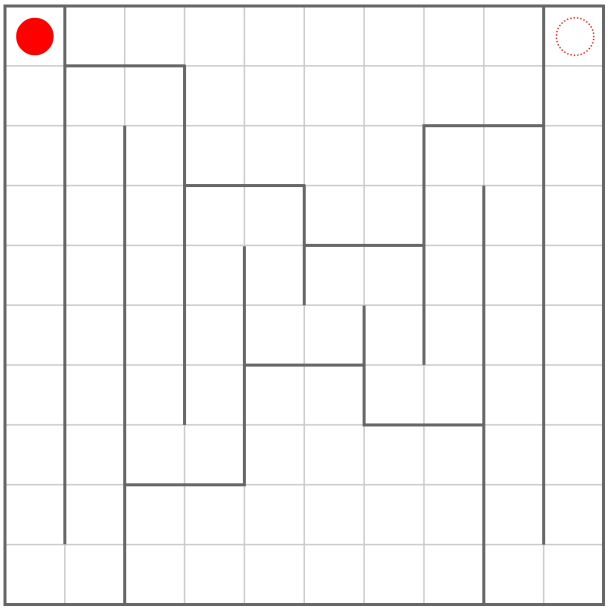
• : ↑→ ↑↑→→ ↑→ ↑↑→→ ↑↑↑→→→

Jednak ze względu na różną liczbę strzałek występujących w tych sekwencjach, nie da się zdefiniować pojedynczej paczki przy pomocy zwykłych strzałek.
Stosując strzałki warunkowe, ruch ten możemy zakodować w zwięzły sposób:

$\boxplus = \uparrow^?_3 \rightarrow^?_3$
• : $\boxplus_5$

Ustalając w powyższej paczce liczbę powtórzeń strzałek warunkowych wzięto pod uwagę ich maksymalną potrzebną ilość (3 ruchy w grę i 3 ruchy w prawo).

Ćwiczenie 1.15. Przeprowadź kółko do celu z wykorzystaniem dwóch strzałek warunkowych: $\downarrow^?$ i $\uparrow^?$ oraz powtórzeń.

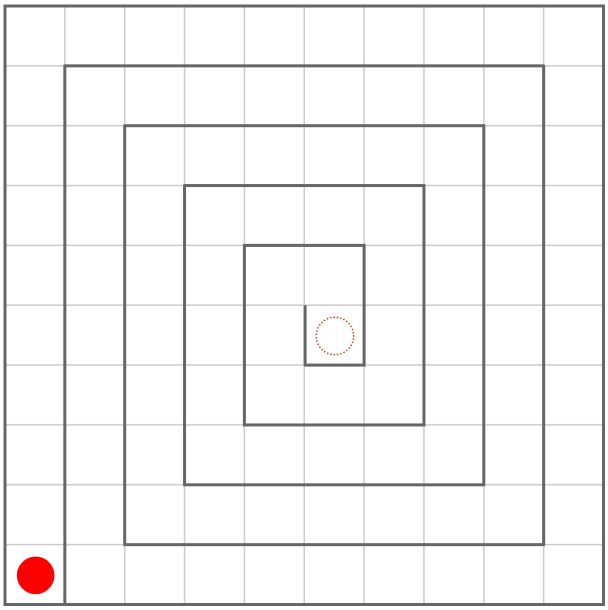


Rozwiązanie:

$\boxplus = \downarrow^?_9 \rightarrow \uparrow^?_9 \rightarrow$
$\bullet : \boxplus_5$

Program zakończy się warunkowo, zanim wykonana zostanie ostania instrukcja ($\rightarrow$), ponieważ kółko wcześniej osiągnie cel.

Ćwiczenie 1.16. Przeprowadź kółko do celu z wykorzystaniem czterech strzałek warunkowych: $\uparrow^{?}$ $\downarrow^{?}$ $\rightarrow^{?}$ $\leftarrow^{?}$ oraz powtórzeń.

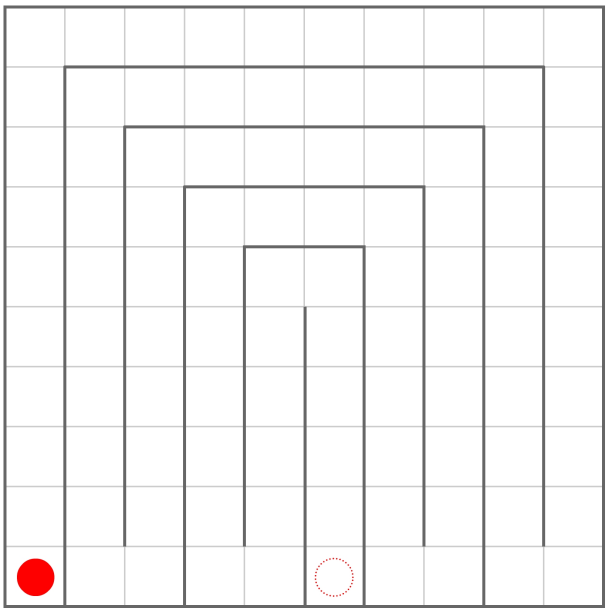


Zadanie to przedstawione już było w ćwiczeniu 1.13. Można je łatwo rozwiązać przy pomocy strzałek warunkowych:

$\boxplus = \uparrow^{?}_{9} \rightarrow^{?}_{9} \downarrow^{?}_{9} \leftarrow^{?}_{9}$

$\bullet : \boxplus_{5}$

Ćwiczenie 1.17. Przeprowadź kółko do celu z wykorzystaniem strzałek warunkowych, przy jak najmniejszej liczbie powtórzeń. Tam gdzie to możliwe użyj zwykłych strzałek.



Poprawne rozwiązanie:

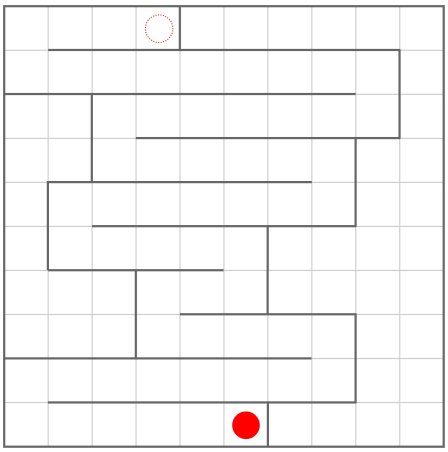
$\boxplus = \uparrow^{?}_{9} \rightarrow^{?}_{9} \downarrow^{?}_{9} \leftarrow \uparrow^{?}_{8} \leftarrow^{?}_{7} \downarrow^{?}_{8} \rightarrow$

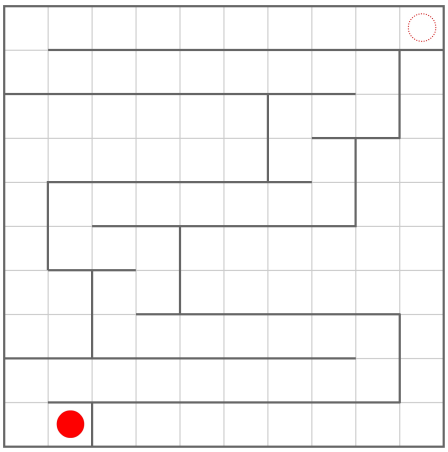
$\bullet : \boxplus_3$

Powyższe przykłady pokazują, że choć strzałki warunkowe znacząco zwiększają możliwości graficznego języka programowania, nie rozwiązują one wszystkich problemów automatycznie. Cały czas niezbędna jest wnikliwa analiza problemu, prowadząca do znalezienia optymalnego rozwiązania.

Strzałki warunkowe pozwalają na rozwiązanie wielu zadań przy pomocy tego samego kodu.

Ćwiczenie 1.18. Napisz jeden program rozwiązujący dwa poniższe zadania. Zoptymalizuj kod wykorzystując jak najmniej strzałek warunkowych oraz ich powtórzeń.



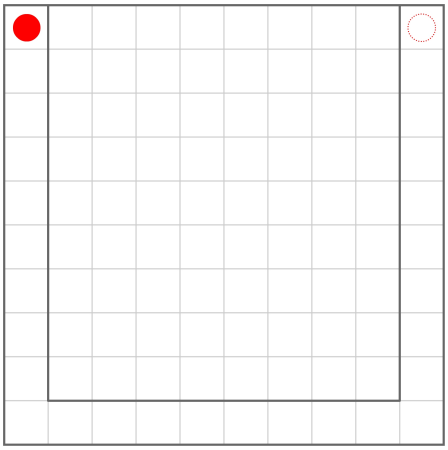


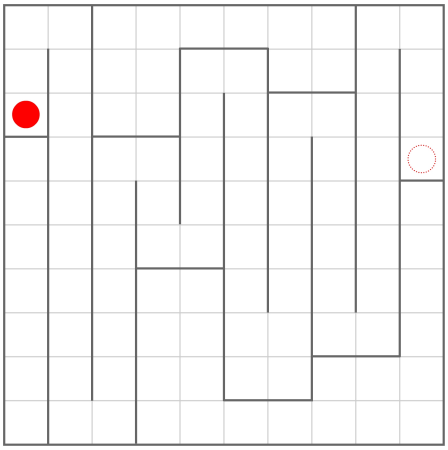
Poprawne rozwiązanie:

$\boxplus = \overset{?}{\underset{8}{\leftarrow}} \uparrow \overset{?}{\underset{9}{\rightarrow}} \uparrow$

$\bullet$: $\boxplus_5$

Ćwiczenie 1.19. Napisz jeden program rozwiązujący dwa poniższe zadania. Zoptymalizuj kod wykorzystując jak najmniej strzałek warunkowych oraz ich powtórzeń.



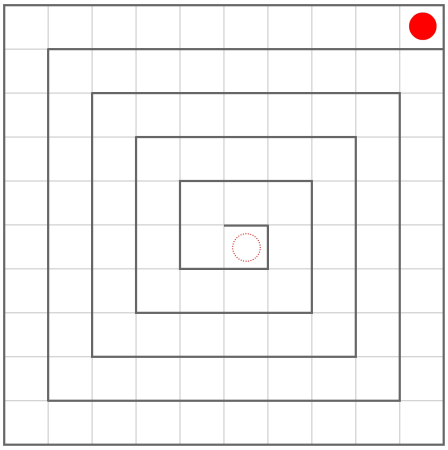


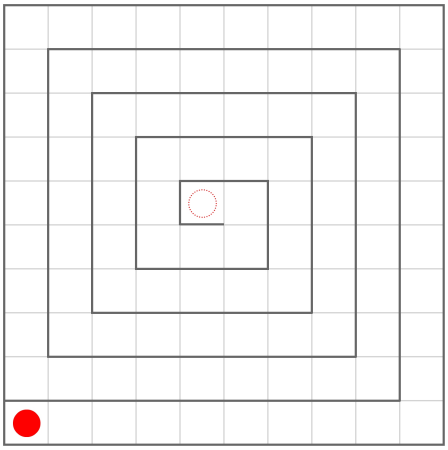
Poprawne rozwiązanie:

$\boxplus = \uparrow_9^? \rightarrow^? \downarrow_9^? \rightarrow$

$\bullet : \boxplus_6$

Ćwiczenie 1.20. Napisz jeden program rozwiązujący dwa poniższe zadania. Zoptymalizuj kod wykorzystując jak najmniej strzałek warunkowych oraz ich powtórzeń.





Poprawne rozwiązania:

$\boxplus = \rightarrow_9^? \uparrow_9^? \leftarrow_9^? \downarrow_9^?$

$\bullet : \boxplus_6$

lub:

$\boxplus = \rightarrow_9^? \uparrow_9^? \leftarrow_9^? \downarrow_9^?$

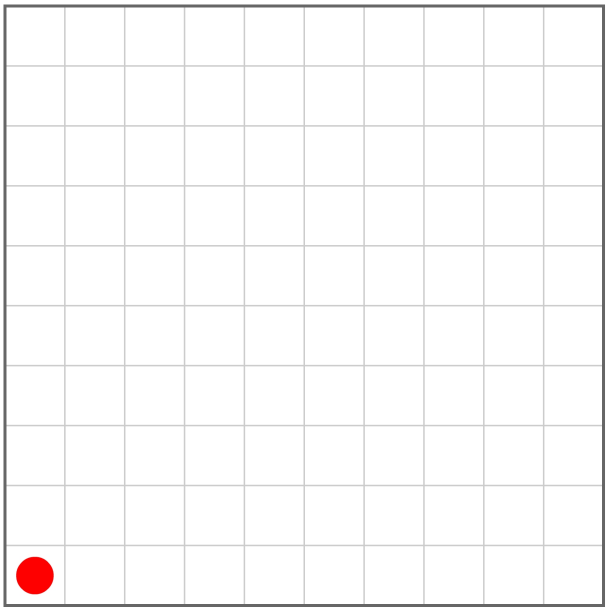
$\bullet : \boxplus_5 \rightarrow$

Warto porównać z uczniami obydwa rozwiązania. Pierwszy zapis jest bardziej zwięzły, ale drugi kod korzysta z mniejszej liczby powtórzeń paczki zawierającej strzałki warunkowe (zob. sekcja 1.11. Ekonomia ruchu).

1.10 Ukryty cel

Cel, do którego dąży kółko, może być na scenie ukryty. W takim wypadku należy napisać kod, który przeprowadzi kółko przez wszystkie dostępne pola, aby mieć pewność, że nie ominęliśmy celu.

Ćwiczenie 1.21. Napisz program, który w poszukiwaniu ukrytego celu przeprowadzi kółko przez wszystkie dostępne pola.

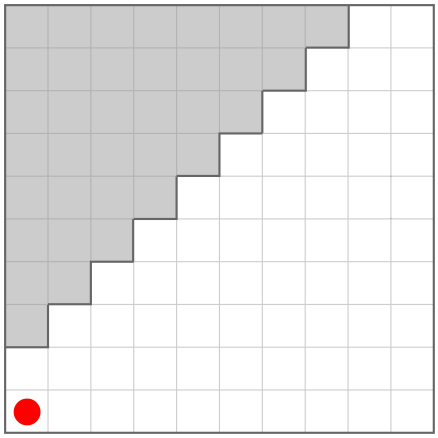


Rozwiązanie tego zadania nie wymaga użycia strzałek warunkowych:

$\boxplus = \uparrow_9 \rightarrow \downarrow_9 \rightarrow$

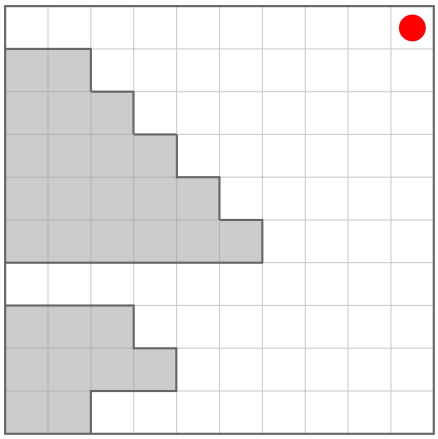
$\bullet : \boxplus_5$

Ćwiczenie 1.22. Napisz program, który w poszukiwaniu ukrytego celu przeprowadzi kółko przez wszystkie dostępne pola.



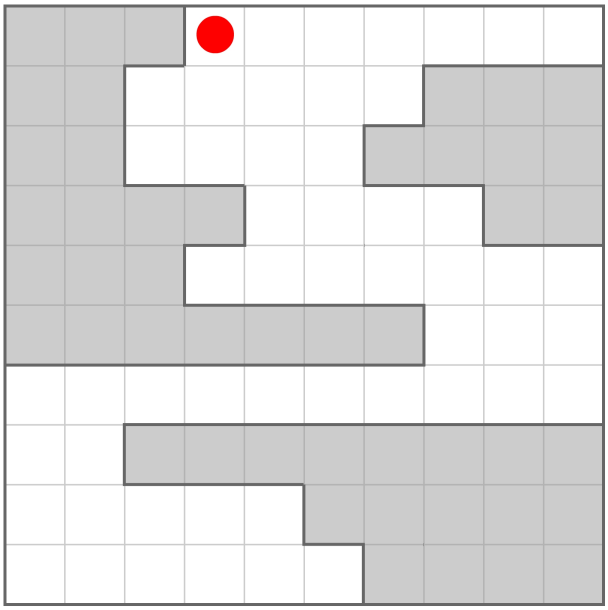
Zwięzłe rozwiązanie tego zadania wymaga już użycia strzałek warunkowych:
$\boxplus = \uparrow_9^? \downarrow_9^? \rightarrow$
$\bullet$: $\boxplus_{10}$

Ćwiczenie 1.23. Napisz program, który w poszukiwaniu ukrytego celu przeprowadzi kółko przez wszystkie dostępne pola.



Zwięzłe rozwiązanie tego zadania wymaga już użycia strzałek warunkowych:
$\boxplus = \leftarrow_9^? \rightarrow_9^? \downarrow$
$\bullet$: $\boxplus_{10}$

Ćwiczenie 1.24. Napisz program, który w poszukiwaniu ukrytego celu przeprowadzi kółko przez wszystkie dostępne pola.



Rozwiązanie:

$\boxplus = \rightarrow_6^? \leftarrow_9^? \downarrow^?$

$\bullet : \boxplus_3 \ \rightarrow_2 \downarrow \ \boxplus_2 \ \rightarrow_4 \downarrow \ \boxplus_5$

Warto zwrócić uwagę na to, że gdyby ostatnia instrukcja w paczce była zwykłą strzałką nie zaś strzałką warunkową, w pewnych sytuacjach (np. przy trzecim powtórzeniu paczki) kółko usiłowałoby przejść przez ścianę, co oznaczałoby błąd programu.

1.11 Ekonomia ruchu

Uczniowie szybko zorientują się, że zwykłe strzałki można zastąpić strzałkami warunkowymi. Istnieje kilka sposobów zapobiegania nadużywaniu strzałek warunkowych.

Po pierwsze, w treści zadania można z góry ustalić limit użycia poszczególnych strzałek warunkowych oraz ich powtórzeń.

W każdym zadaniu można też przyznać pewną pulę środków (pieniędzy), którymi trzeba płacić za użycie poszczególnych instrukcji. Strzałki warunkowe powinny być droższe od zwykłych strzałek. Pisząc kod uczeń będzie musiał kalkulować, czy opłaci mu się stosować takie a nie inne instrukcje. Wyższy koszt strzałek warunkowych ma swoje uzasadnienie w „prawdziwym" programowaniu. Koszt obliczeniowy wykonania instrukcji warunkowej jest wyższy od kosztu obliczeniowego wykonania zwykłej instrukcji.

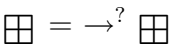
Z drugiej strony warto też zwracać uwagę na zwięzłość programu; na jego „elegancję". Można więc w treści zadania ustalić maksymalną długość kodu rozwiązującego to zadanie.

Ciężko arbitralnie stwierdzić co jest ważniejsze, koszt obliczeniowy, czy może zwięzłość programu. Wiele zależy tu od środowiska, w którym program jest wykonywany. Dlatego też niektóre ćwiczenia warto zadawać uczniom w dwóch wersjach: z jak najtańszym i jak najkrótszym kodem.

1.12 Dalszy rozwój

W przyszłości graficzny język programowania rozwinięty zostanie o możliwość zagnieżdżania paczek. Wewnątrz definicji paczki A można więc będzie umieścić inną, wcześniej zdefiniowaną paczkę B.

Ciekawym przypadkiem teoretycznym, wartym omówienia z uczniami, jest zagnieżdżenie paczki w samej sobie:



Powyższy kod definiuje potencjalnie nieskończony ruch kółka w prawą stronę. Stanowi on prototyp funkcji rekurencyjnej.

2 Mrówka

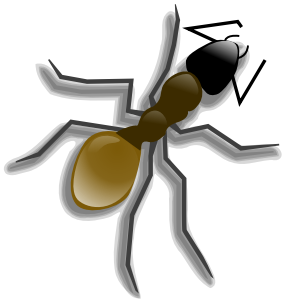
Punktem odniesienia w dotychczasowym modelu ruchu była scena. Obiekt był przesuwany w górę, w dół, w prawo, w lewo, niczym pionek na szachownicy. Kierunki wyznaczane były w odniesieniu do sceny.

Od teraz za punkt odniesienia uznawać będziemy sam obiekt. Wszelkie kierunki określane są przez obiekt, który – niczym osoba patrząca przed siebie – może iść do przodu, obracać się w prawo lub w lewo.

W celu przećwiczenia nowego modelu ruchu, warto powtórzyć na macie część ćwiczeń opisanych w poprzednim rozdziale. Różnica polegać będzie na tym, że uczniowie wydający polecenia, muszą przyjąć perspektywę poruszającego się po scenie ucznia.

Drugim nowym elementem będzie zmiana notacji. W miejsce dotychczasowych strzałek używać będziemy znaków literniczych znajdujących się na klawiaturze komputera i dostępnych w podstawowych edytorach tekstu. Zmiana ta jest jednym z etapów łagodnego przeprowadzania uczniów od zapisów intuicyjnych do notacji abstrakcyjnej, stosowanej w prawdziwym programowaniu.

2.1 Rozkazy



Przypuśćmy, że wyhodowaliśmy mrówkę o wysokim współczynniku inteligencji. Jej inteligencja jest tak duża, że potrafi wykonywać nasze polecenia. Mrówka może na nasze życzenie: poruszać się przed siebie, obracać, zostawiać ślad na swojej drodze. Polecenia przekazywane są mrówce w postaci ciągu znaków, a mrówka wykonuje je w takiej kolejności, z jaką je otrzymała. Na początek wypróbujmy następujące:

- `f` – idź do przodu jeden krok,
- `F` – idź do przodu jeden krok zostawiając ślad z feromonu,

- `r` – obróć się w prawo (zgodnie z ruchem wskazówek zegara),
- `l` – obróć się w lewo (przeciwnie do ruchu wskazówek zegara).

Kontaktujemy się z mrówką za pomocą Octave [1] i programu o nazwie `mrowkaGo`. Należy upewnić się, że w bieżącym katalogu linii komend Octave znajduje się program `mrowkaGo`. Należy pamiętać również, że mrówka na początku zawsze jest skierowana na wschód.

Spróbujmy wykorzystać mrówkę do narysowania kwadratu. Oczywiście mrówka sama z siebie, nie wie czym jest kwadrat. Umie ona jedynie wykonać każde z czterech powyżej opisanych poleceń. Z tego powodu trzeba się zastanowić, jaki ciąg poleceń wydać mrówce, aby otrzymać kwadrat. Mrówka wykona dokładnie to, co jej rozkazano, ale to my musimy zadbać o to, aby rezultat jej pracy był zgodny z oczekiwaniami.

Kwadrat można uzyskać na różne sposoby. Przykładowo, można kazać mrówce wykonać jeden krok, zostawiając ślad (`F`), następnie obrócić się w prawo (`r`) i wykonać jeden krok (`F`), znowu obrócić się w prawo (`r`) i wykonać jeden krok (`F`) i na koniec jeszcze raz obrócić się w prawo (`r`) i wykonać jeden krok (`F`). Ciąg poleceń będzie miał wówczas postać `FrFrFrF`.

Wyniki pracy mrówki możemy zobaczyć po uruchomieniu programu `mrowkaGo` w sposób taki, jak pokazano poniżej:

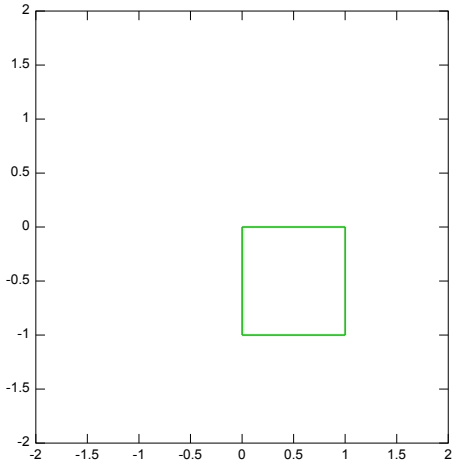
```
>> mrowkaGo('FrFrFrF')
```

Gdybyśmy chcieli aby mrówka wróciła dokładnie do stanu, w jakim znajdowała się na początku, należałoby kazać jej wykonać dodatkowy obrót: `'FrFrFrFr'`.

Kwadrat można uzyskać na inne sposoby. Przykładowo, zamiast obrotów w prawo możemy stosować obroty w lewo: `'FlFlFlF'`. Jedyna różnica jest taka, że teraz kwadrat znajduje się w innym miejscu płaszczyzny.

Ćwiczenie 2.1. Bardziej złożoną figurą od kwadratu jest prostokąt. Jako pierwsze ćwiczenie spróbuj ułożyć dla mrówki ciąg poleceń, po wykonaniu którego narysuje ona prostokąt o wymiarach `2` na `1` i sprawdź jego poprawność programem `mrowkaGo`. Przy okazji uprośćmy nieco język. Zamiast długiego sformułowania „ciąg poleceń, po wykonaniu którego mrówka narysuje ...", używać będziemy

[1] Omówienie środowiska programistycznego Octave znajduje się w dodatku A.

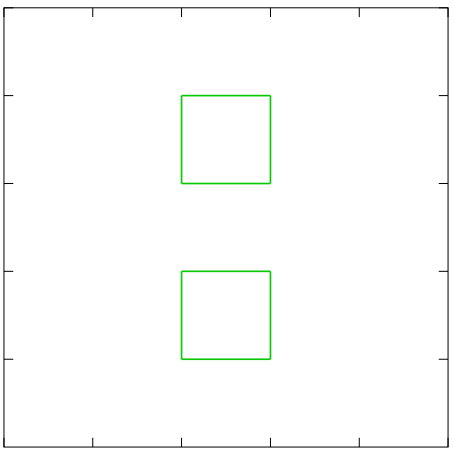


Rys. 1. Rezultat wykonania rozkazu `'FrFrFrF'`.

prostszego „rozkaz narysowania ...”. A zatem, wydaj mrówce rozkaz narysowania prostokąta.

Pora oswoić się z poleceniem `f`. Działa ono podobnie jak polecenie `F`. Różnica polega na tym, że przy `f` mrówka nie zostawia śladu.

Ćwiczenie 2.2. Wydaj mrówce rozkaz narysowania dwóch kwadratów przedstawionych na rysunku 2.



Rys. 2. Wynik ćwiczenia 2.

Zwróć uwagę, że na rysunku nie zaznaczono skali na osiach. Chodzi o to, że w rozwiązaniu nie interesuje nas położenie tylko kształt. Dzięki temu mamy więcej

możliwości rozwiązania tego ćwiczenia.

Oto jeden z dziwaczniejszych sposobów realizacji ćwiczenia 2.2:

```
'FlFfFlFlFfFllfrFlflF'.
```

Porównajmy go z innym rozkazem:

```
'FrFrFrFfFrFrFrF'.
```

Przyjrzyjmy się bardziej wnikliwie konstrukcji tych rozkazów. W przypadku pierwszego rozkazu czynności rysowania obu kwadratów przeplatają się, w drugim rozkazie te czynności zostały oddzielone. Drugi rozkaz sprowadza się do dwukrotnego wykonania rozkazu narysowania jednego kwadratu `'FrFrFrF'`, przy czym pomiędzy pierwszym a drugim kwadratem mrówka ma wykonać odstęp jednostkowy `f`. Najprawdopodobniej tak podszedłby do wykonania tego zadania niczym nieprzymuszony człowiek. Złożony rozkaz narysowania dwóch kwadratów został podzielony na dwa prostsze rozkazy: narysowania kwadratu i wykonania odstępu, przy czym rozkaz narysowania kwadratu został wydany dwukrotnie.

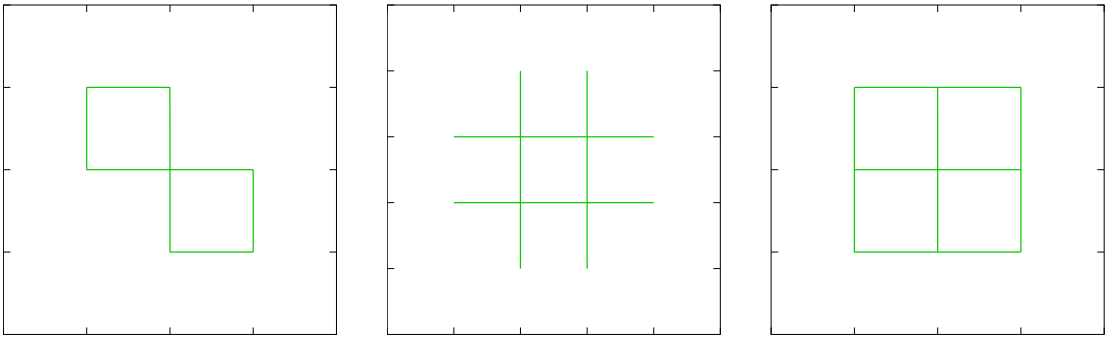
Ten drugi sposób uznawać będziemy za bardziej elegancki, przynajmniej z trzech powodów. Po pierwsze, łatwiej jest człowiekowi zrozumieć konstrukcję rozkazu a – co za tym idzie – modyfikować ją wedle uznania. Po drugie, możliwe jest zapisanie rozkazu w bardziej czytelnej postaci. Po trzecie wreszcie, łatwe jest konstruowanie rozkazów złożonych z wykorzystaniem rozkazów prostszych. Wszystkie te korzyści uzyskamy za pomocą ***paczek*** i ***programów***.

Ćwiczenie 2.3. Wydaj mrówce rozkazy narysowania rysunków przedstawionych na kolejnych obrazkach. Rysunek (c) spróbuj wykonać w dwóch wersjach: z użyciem polecenia `f` oraz bez niego.

2.2 Paczki

Jeżeli przewidujemy, że jakiś rozkaz jeszcze nam się przyda, warto go zapamiętać. Przypuśćmy, że tym rozkazem jest `'FrFrFrF'`, co oznacza, że chcemy mieć „pod ręką” kwadrat. Napiszmy w linii komend Octave:

```
>> kwadrat = 'FrFrFrF'
```



(a) (b) (c)

Rys. 3. Jakie rozkazy spowodują, że mrówka narysuje takie obrazki?

Spowoduje to, że ciąg znaków `'FrFrFrF'` zostanie zapisany w *paczce* o nazwie `kwadrat`. Wpiszmy w linii komend nazwę tej paczki:

```
>> kwadrat
```

a Octave wypisze nam jej zawartość. Możemy również umieszczać w tej paczce inne dane. Na przykład:

```
>> kwadrat = 'FlFlFlF'
```

wymaże zawartość pamięci, na którą wskazuje nazwa `kwadrat` i umieści tam nowy ciąg `'FlFlFlF'`.

Zwróćmy uwagę, że po wciśnięciu ENTER Octave wypisuje nam efekt wykonania komendy w linii komend, przypominając o ostatnio wykonanym działaniu. Jeżeli nie potrzebujemy tej informacji, wystarczy dopisać `;` na końcu komendy:

```
>> kwadrat = 'FlFlFlF';
```

Zauważmy na marginesie, że o samym środowisku Octave można myśleć jako o pewnego rodzaju mrówce: wydajemy mu komendy (polecenia), a on je wykonuje.

Użyty powyżej `kwadrat` jest przykładem ***paczki***. Możemy myśleć o paczce jako o miejscu do przechowywania różnych obiektów, na przykład ciągów znaków. Możemy w paczce coś przechować do późniejszego użycia, możemy tam zaglądać, możemy modyfikować jej zawartość.

W celu wyświetlenia wszystkich paczek, jakie utworzyliśmy w Octave, piszemy:

```
>> who
```

Komenda:

```
>> clear paczka
```

usuwa paczkę o nazwie `paczka` z pamięci, natomiast komenda:

```
>> clear all
```

usuwa z pamięci wszystkie paczki.

Kilka paczek możemy zebrać w jedną paczkę. Wystarczy w tym celu umieścić wybrane paczki wewnątrz nawiasów kwadratowych `[` i `]`. Oto przykład:

```
>> kwadrat = 'FrFrFrFr'
>> odstep = 'ff'
>> wiadro = 'rFlFlFr'
>> rysunek = [kwadrat odstep wiadro]
>> mrowkaGo(rysunek)
```

Można też łączyć paczki z rozkazami, które nie zostały umieszczone w żadnych paczkach:

```
>> rysunek = [kwadrat 'ffFf' kwadrat]
>> mrowkaGo(rysunek)
```

Choć na pierwszy rzut oka poniższa konstrukcja może wyglądać dziwnie, jest ona bardzo przydatna. Spójrzmy:

```
>> rysunek = [kwadrat 'ffFf' kwadrat]
>> rysunek = [rysunek 'fff' rysunek]
>> mrowkaGo(rysunek)
```

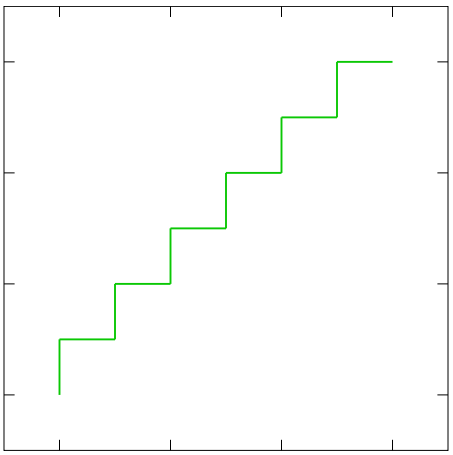
Dlaczego druga linijka działa? Otóż w pierwszej kolejności Octave używa bieżącej zawartości paczki `rysunek` do wykonania komendy `[rysunek 'fff' rysunek]`, a dopiero potem następuje zapisanie jej rezultatu do paczki `rysunek`. Dotychczasowa zawartość paczki zostaje zastąpiona nową zawartością. Będziemy z tego sposobu korzystać często.

Jeżeli z paczkami już się oswoiliśmy, pora opanować pierwszą poważną konstrukcję programistyczną.

2.3 Pętle czyli powtórzenia

Zdarzyć się może, że rozkaz jaki wydaliśmy mrówce, składa się z kilku powtórzeń pewnego ciągu poleceń.

Ćwiczenie 2.4. Wydaj mrówce rozkaz narysowania schodów przedstawionych na rysunku 4.



Rys. 4. Jaki rozkaz ma wykonać mrówka, żeby narysować takie schody?

Ćwiczenie można zrealizować następująco:

```
'lFrFlFrFlFrFlFrFlFrFlFrF'
```

Zauważmy, że w rozkazie powtarza się sześć razy sekwencja `'lFrF'`. Oznacza to, że wydanie jednego długiego rozkazu jest równoważne wydaniu sześciu krótkich identycznych rozkazów. Powód tego jest oczywisty: schody składają się z identycznych stopni, a krótki rozkaz odpowiada narysowaniu pojedynczego stopnia. Poskładamy teraz z tych krótkich rozkazów jeden długi rozkaz, podobnie jak robiliśmy to w części pierwszej, korzystając z paczki. W tej paczce zapamiętamy tylko krótki rozkaz narysowania pojedynczego stopnia:

```
>> stopien='lFrF'
```

Teraz należy połączyć ze sobą sześć stopni, otrzymując pożądany rozkaz narysowania schodów. Można zrobić to tak:

```
>> schody=[stopien stopien stopien stopien stopien stopien]
```

Ten sposób rozwiązania ma jednak poważną wadę. Jak bowiem utworzylibyśmy schody składające się z – dajmy na to – stu stopni?

W takich sytuacjach najlepiej posłużyć się tak zwaną pętlą. Pętla jest jedną z podstawowych konstrukcji programistycznych. Ma na celu ułatwienie wykonywania powtarzalnych czynności. Nazwa *pętla* jest fachowym określeniem *powtórzenia* (patrz, poprzedni rozdział). Rozpoczniemy od korzystania z „amatorskiej" wersji pętli. Wersja amatorska polega na użyciu przygotowanej komendy `repeat`:

```
>> schody = repeat(stopien, 6)
```

W wyniku działania powyższej pętli paczka `schody` zawierać będzie ciąg poleceń `'lFrFlFrFlFrFlFrFlFrF'`. Kod źródłowy komendy `repeat` znajduje się w opisie rozwiązania ćwiczenia 2.18.

Szablon komendy `repeat` ma postać:

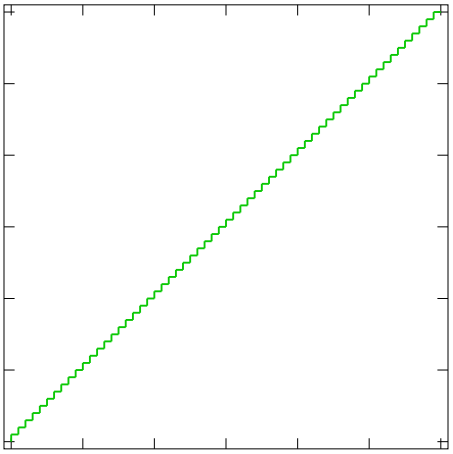
```
>> duzapaczka = repeat(malapaczka, ile_razy)
```

gdzie `ile_razy` jest liczbą, która wskazuje, ile paczek `malapaczka` ma zostać sklejonych w jedną paczkę `duzapaczka`.

Teraz łatwo jest narysować nawet bardzo wysokie schody. Wystarczy w miejsce `ile_razy` wstawić odpowiednią liczbę:

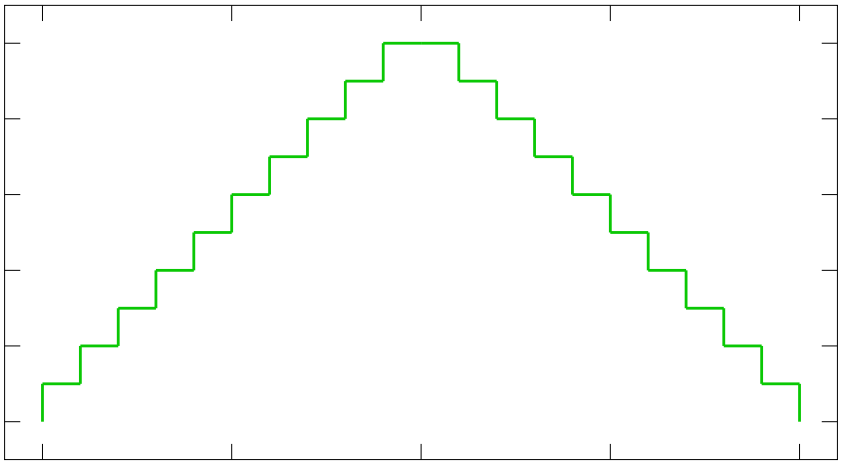
```
>> stopien = 'lFrF';
>> schody = repeat(stopien, 60);
>> mrowkaGo(schody);
```

Rezultat widać na rysunku 5.



Rys. 5. Wysokie schody narysowane za pomocą pętli.

Ćwiczenie 2.5. Rysowaliśmy schody, które pną się daleko w górę. Dorysujmy zatem schody, które prowadzą z powrotem na dół (patrz rysunek 6). Utwórz odpowiedni rozkaz, korzystając z pętli.



Rys. 6. Schody prowadzące do góry i z powrotem na dół.

Przykładowe rozwiązanie:

```
>> stopienwgore = 'lFrF';
>> schodywgore = repeat(stopienwgore, 10);
>> stopienwdol = 'FrFl';
>> schodywdol = repeat(stopienwdol, 10);
```

```
>> schody = [schodywgore schodywdol];
>> mrowkaGo(schody);
```

Zauważmy, że wprowadzenie nawet drobnej poprawki pociąga za sobą konieczność wywołania wszystkich komend w odpowiedniej kolejności. Warto uprościć sobie pracę, umieszczając zestaw poleceń w pliku tekstowym. Tworzymy w bieżącym katalogu plik tekstowy o dowolnej nazwie i rozszerzeniu `.m`, na przykład `mojekomendy.m`. Umieszczamy w nim wszystkie komendy, jakie mają zostać wykonane:

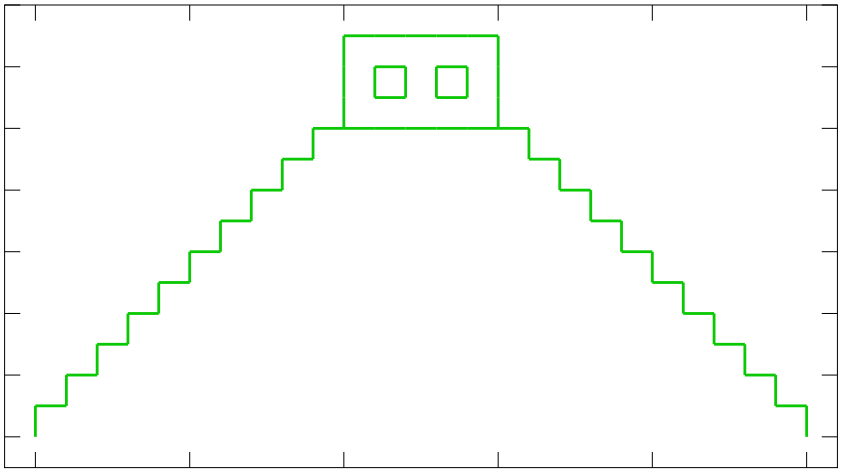
```
stopienwgore = 'lFrF';
schodywgore = repeat(stopienwgore, 10);
stopienwdol = 'FrFl';
schodywdol = repeat(stopienwdol, 10);
schody = [schodywgore schodywdol];
mrowkaGo(schody);
```

i zapisujemy. Teraz, w linii komend Octave, wpisujemy nazwę pliku:

```
>> mojekomendy
```

i po wpisaniu tej nazwy uruchomione zostaną kolejno wszystkie komendy, jakie umieściliśmy w pliku. Plik z komendami Octave nazywać będziemy *skryptem*. Plik `mojekomendy.m` jest przykładem skryptu. Jeżeli chcemy teraz wprowadzić zmiany, wystarczy zmodyfikować w odpowiednich miejscach zawartość skryptu. Nie trzeba już wywoływać kolejno wszystkich komend, gdyż zostanie to zrobione po wpisaniu nazwy skryptu w linii komend. W ten sposób nazwa skryptu staje się nową komendą dla Octave.

Ćwiczenie 2.6. Narysuj domek, do którego prowadzą wysokie schody z obu stron, jak pokazano na rysunku 7.



Rys. 7. Domek ze schodami.

W tym ćwiczeniu istotnym elementem rozwiązania jest podział problemu na trzy części:

- narysowanie schodów z jednej strony,
- narysowanie domku,
- narysowanie schodów z drugiej strony.

Narysowanie domku również można podzielić na trzy części:

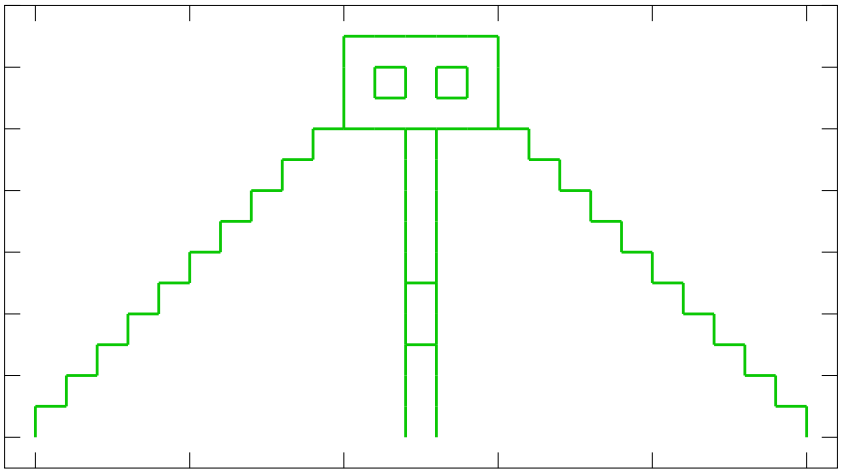
- narysowanie jednego okna,
- narysowanie drugiego okna,
- narysowanie ścian.

Przykładowe rozwiązanie, zapisane w skrypcie, jest następujące:

```
stopienwgore = 'lFrF';
schodywgore = repeat(stopienwgore, 10);
stopienwdol = 'FrFl';
schodywdol = repeat(stopienwdol, 10);
domek = 'FFFFFlFFFlFFFFFlFFFlflfrFlFlFlFlffFlFlFlFlffrfl';
schody = [schodywgore domek schodywdol];
mrowkaGo(schody);
```

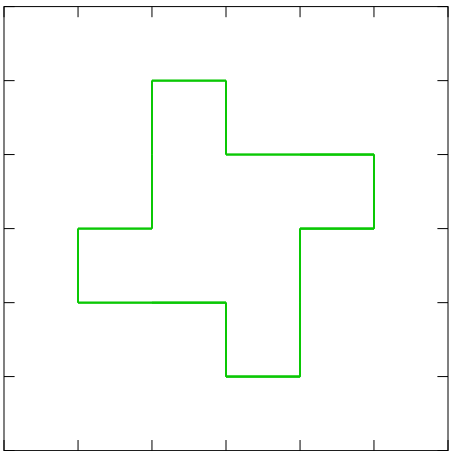
Warto zastanowić się, jak utworzyć paczkę `domek` z wykorzystaniem pętli.

Ćwiczenie 2.7. Jeżeli schody są wysokie, to komu chciałoby się po nich wchodzić? Dorysuj do poprzedniego obrazka windę według własnego pomysłu lub taką, jak na rysunku 8.



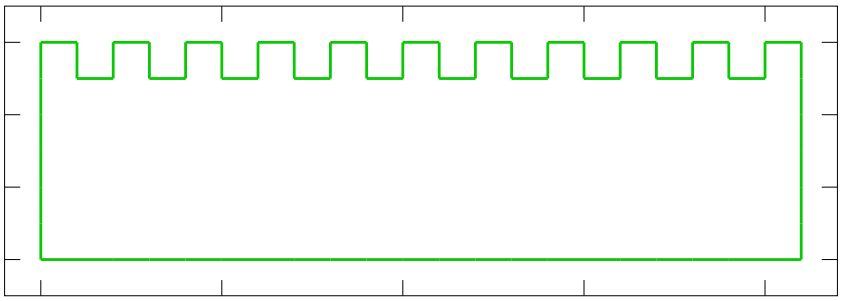
Rys. 8. Domek ze schodami i windą.

Ćwiczenie 2.8. Na rysunku 9 pokazany jest ślad, jaki powinna oznaczyć mrówka. Spróbuj ułożyć rozkaz narysowania takiego śladu, korzystając z pętli. Na jakie identyczne fragmenty można podzielić ten ślad?



Rys. 9. Zamknięta droga mrówki.

Ćwiczenie 9. Narysuj mur zamku, podobny do przedstawionego na rysunku 10. Staraj się, gdzie tylko jest to możliwe i celowe, stosować pętlę `repeat`.



Rys. 10. Mur zamku.

Przykładowe rozwiązanie jest następujące:

```
murek = 'FrFlFlFr';
murgora = repeat(murek,10);
bokprawy = repeat('F',6);
podstawa = repeat('F',21);
boklewy = repeat('F',6);
murdol = ['Fr' bokprawy 'r' podstawa 'r' boklewy];
mur = strcat(murgora,murdol);
mrowkaGo(mur);
```

Przy okazji rozwiązywania tego ćwiczenia, warto sprawdzić, co się stanie, jeżeli w linii zaczynającej się od `murdol=` ... wstawimy w miejsce paczek `bokprawy`, `podstawa`, `boklewy` od razu instrukcję `repeat...`, tzn.:

```
murek = 'FrFlFlFr';
murgora = repeat(murek,10);
murdol = ['Fr' repeat('F',6) 'r'
                  repeat('F',21) 'r' repeat('F',6)];
mur = [murgora murdol];
mrowkaGo(mur);
```

Linia zawierająca definicję paczki `murdol` jest tak długa, że trzeba ją było podzielić na dwie części, gdyż nie mieściła się na tej stronie. W skrypcie jednak jest to jedna długa linijka. Rezultat tego skryptu jest identyczny z rezultatem

wcześniejszego skryptu. Okazuje się, że wynik komendy `repeat` można od razu wstawić między nawiasy łączące paczki, bez konieczności definiowania paczki. Jest to sensowne, pod warunkiem, że ta paczka jest używana tylko w jednym miejscu i nie będzie już potrzebna w innym miejscu. Można zadać pytanie, że skoro tak to działa, to czy można komendę `repeat` umieścić wewnątrz komendy `repeat`.

Ćwiczenie 2.10. Sprawdź, jaki jest rezultat działania poniższej komendy, w której pętla `repeat` została umieszczona wewnątrz drugiej pętli `repeat`:

```
>> repeat(repeat('F',2),3)
mrowkaGo(mur);
```

Nietrudno jest odgadnąć, co się stanie. Sama komenda `repeat('F',2)` daje w wyniku paczkę `'FF'`. Z kolei komenda `repeat(repeat('F',2),3)` jest równoważna komendzie `repeat('FF',3)`, która zwraca paczkę `'FFFFFF'`. Tego typu konstrukcję nazywamy „podwójną pętlą". Kolejne ćwiczenie zademonstruje przydatność podwójnej pętli.

Ćwiczenie 2.11. Korzystając z podwójnej pętli, utwórz rozkaz narysowania siatki prostokątnej, składającej się z ośmiu kwadratów w poziomie i dziesięciu kwadratów w pionie (patrz rysunek 11).

Przyjrzyjmy się przykładowemu rozwiązaniu. Ważne jest, aby zdefiniować sobie paczkę `kwadrat` tak, aby mrówka po jego narysowaniu znajdowała się w dogodnej pozycji do rozpoczęcia rysowania kolejnego kwadratu. Poprzednio `kwadrat` był paczką o postaci:

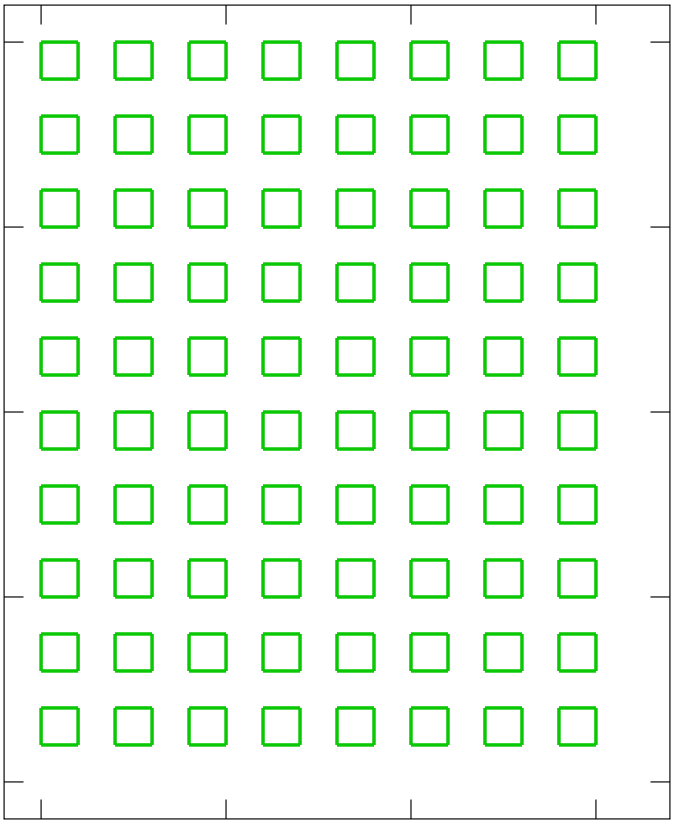
```
>> kwadrat = 'FrFrFrF'
```

Tym razem warto wstawić na koniec dodatkowe polecenia, które ustawią mrówkę po prawej stronie narysowanego kwadratu:

```
>> kwadrat = 'FrFrFrFrff'
```

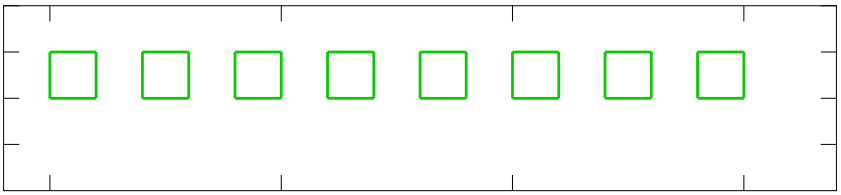
Mrówka będzie zwrócona na wschód i znajdować się będzie o jeden krok od prawego górnego rogu narysowanego kwadratu. Stąd może zacząć rysować kolejny kwadrat, określony identyczną paczką. Potem kolejny i jeszcze jeden i tak dalej:

```
>> mrowkaGo(repeat(kwadrat, 8))
```



Rys. 11. Siatka prostokątna.

Powyższa komenda spowoduje wyświetlenie rzędu ośmiu kwadratów (patrz rysunek 12). Po narysowaniu całego rzędu, mrówka znajdować się będzie o jeden krok



Rys. 12. Rząd kwadratów.

od prawego górnego rogu ostatniego kwadratu. Zanim przystąpimy do narysowania kolejnego rzędu, trzeba mrówkę przenieść w miejsce, z którego będzie zaczynała rysować kolejny rząd. Najpierw mrówka musi obrócić się za siebie (`'rr'` lub `'ll'`) i wrócić na początek bieżącego rzędu. Powrót polega na wykonaniu rozkazu `'f'` odpowiednią liczbę razy. Na każdy narysowany kwadrat przypadają dwa kroki

`'ff'`: jeden krok na bok kwadratu a drugi krok na odstęp między kwadratami. W sumie liczba kroków `'f'` do wykonania jest równa podwojonej liczbie kwadratów. Uzyskamy je komendą `repeat('f',16)` lub `repeat('ff',8)` . Mrówka znajduje się teraz w lewym górnym rogu pierwszego kwadratu narysowanego rzędu i jest zwrócona na zachód. Aby przejść do miejsca, które będzie początkiem drugiego rzędu, mrówka musi obrócić się w dół (`'l'),`wykonać dwa kroki w dół (`'ff'`) i obrócić się na wschód (`'l'`). Ostatni obrót ma na celu przygotowanie mrówki do rysowania kolejnego rzędu. Podsumowując, cała procedura przeniesienia mrówki na dół, to jest do pozycji początkowej do rysowania kolejnego rzędu, może być zapisana w paczce:

```
>> nadol = ['rr' repeat('ff', 8) 'lffl']
```

Rozkaz narysowania pojedynczego rzędu kwadratów z przejściem do kolejnego rzędu jest następujący:

```
>> mrowkaGo([repeat(kwadrat,8) nadol])
```

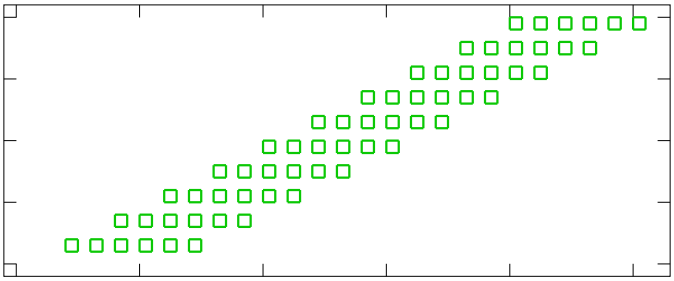
Natomiast rozkaz narysowania dziesięciu takich rzędów zrealizujemy poprzez „obudowanie" powyższego rozkazu pętlą `repeat`:

```
>> mrowkaGo(repeat([repeat(kwadrat,8), nadol], 10))
```

Powyższy rozkaz ma poważną wadę. Widać ją podczas próby zmiany szerokości siatki, na – dajmy na to – 11. Wydawałoby się, że jest to proste, wystarczy przecież zamienić `8` na `11`:

```
>> mrowkaGo(repeat([repeat(kwadrat,11) nadol], 10))
```

Rezultat widoczny na rysunku 13 jest inny niż oczekiwany. Problem polega na tym, że paczka `nadol` zakłada cofnięcie się mrówki o odległość ośmiu kwadratów z przerwami. Liczba kroków wstecz powinna być powiązana z liczbą kwadratów rysowanych w rzędzie.



Rys. 13.

Rozwiązanie tego problemu polega na tym, aby nie używać paczki `nadol` ale stosować bezpośrednio jej zapis:

```
>> mrowkaGo(repeat([repeat(kwadrat, 6)
        ['rr' repeat('ff', 6) 'lffl']], 10))
```

Powyższa komenda jest nieco przydługa, ale ma tę zaletę, że łatwiej jest za jej pomocą rysować siatki o różnych wymiarach. Dwie pierwsze liczby oznaczają liczbę kwadratów w rzędzie a ostatnia liczba oznacza liczbę rzędów. Szybko dochodzimy do wniosku, że jeszcze nie jest to najwygodniejsze rozwiązanie. Niewygoda polega na konieczności zmiany liczby, oznaczającej liczbę kwadratów w rzędzie, w dwóch miejscach. A przecież jest to jedna i ta sama liczba. Dlaczego zatem trzeba zmieniać jej wartość w dwóch miejscach? Rozwiązaniem tego problemu jest użycie paczki, w której przechowywać będziemy liczbę kwadratów w rzędzie. Nazwijmy tę paczkę `n`. Możemy używać ją w tych miejscach, w których potrzebujemy wstawić liczbę, jaka jest w niej zapisana:

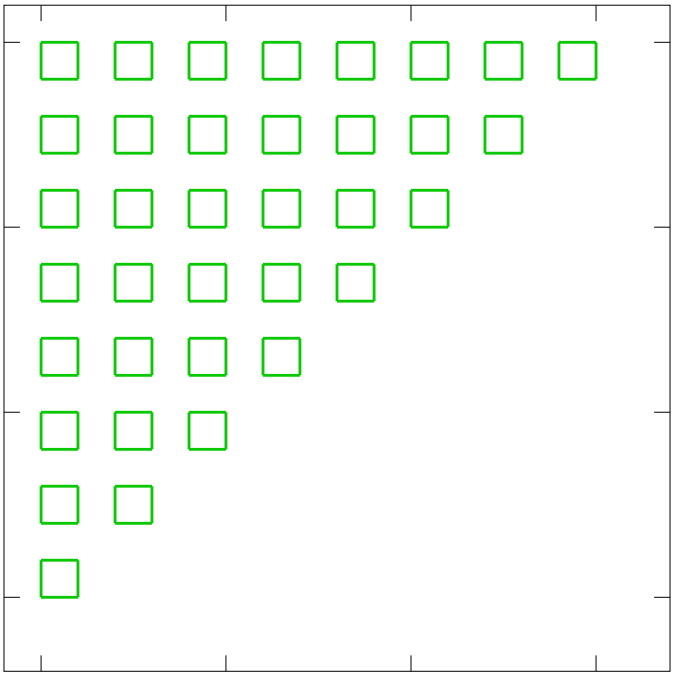
```
>> n = 8;
>> mrowkaGo(repeat([repeat(kwadrat, n)
              ['rr' repeat('ff', n) 'lffl']], 10))
```

Dodatkowo można utworzyć drugą paczkę `m`, do zapisania liczby rzędów i umieścić wszystko w skrypcie:

```
n = 8;
m = 10;
mrowkaGo(repeat([repeat(kwadrat, n)
         ['rr' repeat('ff', n) 'lffl']], m))
```

Teraz korzystanie z rozkazu rysowania siatki jest bardzo wygodne. Łatwo znaleźć miejsce, w którym wpisuje się pożądane wymiary siatki.

Ćwiczenie 2.12. Utwórz rozkaz narysowania siatki trójkątnej, składającej się z ośmiu kwadratów w pierwszym rzędzie, siedmiu kwadratów w drugim rzędzie, sześciu w trzecim i tak dalej (patrz rysunek 14).



Rys. 14. Siatka trójkątna.

Próbując rozwiązać to ćwiczenie z wykorzystaniem pętli `repeat`, zmuszeni jesteśmy ośmiokrotnie wykonać pętlę `repeat` z różną liczbą powtórzeń, na przykład tak:

```
kwadrat='FrFrFrFrff';
n = 8;
siatka = '';
siatka = [siatka repeat(kwadrat, 8) ];
siatka = [siatka 'rr' repeat('ff', 8) 'lffl'];
siatka = [siatka repeat(kwadrat, 7) ];
siatka = [siatka 'rr' repeat('ff', 7) 'lffl'];
```

```
siatka = [siatka repeat(kwadrat, 6) ];
siatka = [siatka 'rr' repeat('ff', 6) 'lffl'];
siatka = [siatka repeat(kwadrat, 5) ];
siatka = [siatka 'rr' repeat('ff', 5) 'lffl'];
siatka = [siatka repeat(kwadrat, 4) ];
siatka = [siatka 'rr' repeat('ff', 4) 'lffl'];
siatka = [siatka repeat(kwadrat, 3) ];
siatka = [siatka 'rr' repeat('ff', 3) 'lffl'];
siatka = [siatka repeat(kwadrat, 2) ];
siatka = [siatka 'rr' repeat('ff', 2) 'lffl'];
siatka = [siatka repeat(kwadrat, 1) ];
siatka = [siatka 'rr' repeat('ff', 1) 'lffl'];
mrowkaGo(siatka)
```

Każda zmiana liczby rzędów `n` pociąga za sobą konieczność doklejania lub usuwania odpowiedniej liczby linii skryptu. Nie tego oczekujemy od pętli, wolelibyśmy bardziej automatyczne rozwiązanie. Jest to możliwe tylko z wykorzystaniem profesjonalnej wersji pętli.

2.4 Pętla `for`

Komenda `repeat` wprowadzona w poprzednim podpunkcie jest prototypem pętli `for`. „Szablon" hipotetycznej, uproszczonej wersji pętli `for` mógłby wyglądać tak:

```
for ile_razy
    komendy;
end
```

Odstęp w środkowym wierszu ma jedynie znaczenie estetyczne, to znaczy ułatwia czytanie. W miejsce `ile_razy` podawalibyśmy liczbę wskazującą ile razy mają zostać wykonane `komendy`. W miejsce tekstu `komendy` wstawiane byłyby komendy do wykonania. W tej postaci pętla `for` nie różniłaby się niczym od komendy `repeat` poza tym, że zapis byłby trochę inny. Faktycznie pętla `for` może znacznie więcej od komendy `repeat` dzięki temu, że jej zapis zawiera coś jeszcze. Oto prawdziwy „szablon" pętli `for` :

```
for licznik = wartosc_poczatkowa:wartosc_koncowa
    komendy;
end
```

`licznik` traktujemy jak paczkę zawierającą pewną liczbę, podobnie jak paczka `n` z ćwiczenia 2.12. Przed każdym wykonaniem `komendy` liczba `licznik` przyjmuje określoną wartość liczbową. Przed pierwszym wykonaniem `komendy` zostaje mu przypisana `wartosc_poczatkowa`, przed kolejnym wykonaniem `komendy` wartość o `1` większa i tak dalej, aż do ostatniego wykonania `komendy` , kiedy to przypisana mu zostaje `wartosc_koncowa`. Aby zobaczyć, jakie to proste, sprawdźmy pętlę `for` w działaniu. Napiszmy następujący skrypt o nazwie `skryptfor.m` (lub jakiejkolwiek innej):

```
for i = 1:5
   i
end
```

Powyższy skrypt nie robi nic poza wypisaniem na ekranie wartości, jakie w kolejnych krokach pętli `for` przyjmuje licznik `i`:

```
>> skryptfor
i =  1
i =  2
i =  3
i =  4
i =  5
```

Sprawdźmy teraz efekt działania skryptu:

```
for i = 4:5
   i
end
```

Na ekranie zostało wypisane:

```
>> skryptfor
i =  4
i =  5
```

Ćwiczenie 2.13. Spróbuj podać wzór na liczbę powtórzeń w zależności od wartości liczb `wartosc_poczatkowa` i `wartosc_koncowa`.

Prawidłowym wzorem jest:
`liczba_powtorzen = wartosc_koncowa - wartosc_poczatkowa + 1`.

Ciekawym eksperymentem, potwierdzającym poprawność wzoru na liczbę powtórzeń, jest uruchomienie poniższego skryptu:

```
for i = 3:3
   i
end
```

W rezultacie otrzymujemy

```
>> skryptfor
i =  3
```

Jeszcze ciekawszym eksperymentem jest:

```
for i = 4:3
   i
end
```

W tym przypadku nie wykonało się żadne powtórzenie. Jest to naturalne, jeżeli przyjmujemy założenie, że wartości licznika `i` zwiększają się o `1`.

Ćwiczenie 2.14. Wykonaj ponownie ćwiczenie 2.4 rysowania schodów składających się z identycznych stopni. Tym razem do sformułowania rozkazu wykorzystaj pętlę `for`.

Rozwiązanie polega na utworzeniu pustych schodów a następnie dokładaniu kolejnych stopni w pętli `for`. Paczka `schody` początkowo będzie pusta:

```
>> schody = '';
```

Przypomnijmy rozkaz narysowania pojedynczego stopnia:

```
>> stopien='lFrF';
```

Liczbę stopni, z jakich składać się będą schody, umieścimy w paczce `n`:

```
>> n = 6;
```

Teraz można przystąpić do dodawania do paczki `schody` paczek `stopien`:

```
>> for i = 1:n
>    schody = [schody stopien]
> end
```

Po napisaniu `end` i wciśnięciu ENTER Octave wyświetli rezultat, jaki otrzymujemy po każdym powtórzeniu. Wynik końcowy sprawdzamy, wydając w linii poleceń komendę:

```
>> mrowkaGo(schody)
```

Jeżeli nie interesują nas rezultaty cząstkowe, dodajemy znak `;` na końcu komendy wewnątrz pętli `for`. Całość umieścić możemy w skrypcie:

```
n = 10;
stopien = 'lFrF';
schody = '';
for i = 1:n
   schody = [schody stopien];
end
mrowkaGo(schody)
```

Komenda:

```
schody = '';
```

jest potrzebna do tego, aby upewnić się, że przed rozpoczęciem dodawania stopni do paczki `schody`, paczka `schody` jest pusta. Zastanów się, co by się działo, gdyby tę komendę usunąć ze skryptu?

Tego rodzaju eksperymenty są częstą praktyką przy testowaniu działania skryptów. Ich przeprowadzanie ułatwia znak `%`. Tekst rozpoczynający się po tym znaku aż do końca linii jest ignorowany przez Octave. Na ogół korzysta się z tego do umieszczenia komentarzy w skrypcie:

```
n = 10;             % liczba stopni
stopien = 'lFrF';
schody = '';        % inicjalizacja
for i = 1:n
   schody = [schody stopien];   % dodanie stopnia do schodów
end
mrowkaGo(schody)    % wyświetlenie śladu zostawionego przez mrówkę
```

W ten sposób można również „wyłączyć" komendę, pozostawiając jej treść. Jest to wygodniejsze niż usuwanie, gdyż łatwo można wrócić do poprzedniej wersji skryptu:

```
n = 10;
stopien = 'lFrF';
% schody = '';
for i = 1:n
   schody = [schody stopien];
end
mrowkaGo(schody)
```

Warto wykonać jeszcze raz wszystkie ćwiczenia związane z wykorzystaniem komendy `repeat` za pomocą pętli `for`.

Szczególną uwagę poświęcimy problemowi narysowania siatki kwadratowej i trójkątnej.

Ćwiczenie 2.15. Korzystając z podwójnej pętli `for`, utwórz rozkaz narysowania siatki prostokątnej, składającej się z pięciu kwadratów w pionie i ośmiu kwadratów w poziomie.

Rozwiązanie umieścimy w skrypcie `siatkap.m`. Rozpoczniemy od zdefiniowania wymiarów siatki:

```
n = 5;
m = 8;
```

Definiujemy paczkę `kwadrat`:

```
kwadrat='FrFrFrFrff';
```

oraz przygotowujemy paczkę `siatka`:

```
siatka = '';
```

Pierwsza, zewnętrzna pętla dotyczyć będzie kolejnych rzędów:

```
for i = 1:n
    % rysowanie i-tego rzędu
end
```

Wewnątrz tej pętli rysować będziemy kolejne rzędy kwadratów. Skupić się zatem możemy teraz tylko na rozkazie rysowania pojedynczego rzędu. A rysowanie rzędu kwadratów wykonywać się będzie w pętli `for`. Licznik tej pętli działać powinien niezależnie od pętli zewnętrznej, dlatego oznaczymy go innym symbolem, dajmy na to `j`:

```
for j = 1:m
    siatka = [siatka kwadrat];
end
```

Ale to jeszcze nie wszystko. Pamiętamy, że po narysowaniu rzędu kwadratów, mrówka musi przyjąć pozycję dogodną do rysowania kolejnego rzędu. Jej ruch do tej pozycji rozpoczyna się od obrócenia za siebie:

```
siatka = [siatka 'rr'];
```

następnie wykonania tylu podwójnych kroków, bez zostawiania śladu, ile jest kwadratów w rzędzie:

```
for j = 1:m
    siatka = [siatka 'ff'];
end
```

i na koniec przejście do rzędu poniżej z ustawieniem się na wschód:

```
siatka = [siatka 'lffl'];
```

Podsumowując, zawartość skryptu jest następująca:

```
n = 5;
m = 8;
kwadrat='FrFrFrFrff';
siatka = '';
for i = 1:n
   for j = 1:m
       siatka = [siatka kwadrat];
   end
   siatka = [siatka 'rr'];
   for k = 1:m
       siatka = [siatka 'ff'];
   end
   siatka = [siatka 'lffl'];
end
mrowkaGo(siatka)
```

Teraz jasne jest, że wcięcia w liniach ułatwiają zrozumienie napisanego skryptu.

Do narysowania siatki kwadratowej wystarczy jedna liczba określająca wymiary, dzięki czemu skrypt się nieco upraszcza:

```
n = 8;
kwadrat='FrFrFrFrff';
siatka = '';
for i = 1:n
   for j = 1:n
       siatka = [siatka kwadrat];
   end
   siatka = [siatka 'rr'];
   for k = 1:n
       siatka = [siatka 'ff'];
   end
   siatka = [siatka 'lffl'];
end
mrowkaGo(siatka)
```

Liczniki obu pętli, zewnętrznej i wewnętrznej, przyjmują wartości od `1` do `n`.

Ćwiczenie 2.16. Wykonaj eksperymenty z podwójną pętlą `for`, polegające na wyświetlaniu wartości liczników w kolejnych powtórzeniach pętli:

```
for i = 1:3
   for j = 2:5
      i
      j
   end
end
```

lub

```
for i = 1:3
   i
   for j = 2:5
      j
   end
end
```

Ćwiczenie 2.17. Dopiero teraz jesteśmy w stanie „porządnie" zrealizować ćwiczenie 2.12. Przypomnijmy, chodziło o narysowanie siatki trójkątnej przedstawionej na rysunku 14).

W celu rozwiązania tego ćwiczenia można za punkt wyjścia przyjąć skrypt realizujący siatkę kwadratową. Zadajmy sobie pytanie, dlaczego rysowana tu siatka jest kwadratowa? Dlatego, że każdy rząd ma tę samą liczbę kwadratów, to oczywiste. A dlaczego każdy rząd ma tę samą liczbę kwadratów? Dlatego, że licznik `j` wewnętrznej pętli zmienia się od `1` do `n` dla każdego rzędu. Jak można spowodować, żeby w każdym kolejnym rzędzie liczba powtórzeń była o jeden mniejsza? A dokładniej: w pierwszym rzędzie mają być wszystkie kwadraty, w drugim o jeden mniej, w trzecim o dwa mniej, w czwartym o trzy mniej i tak dalej. Dla pierwszego rzędu licznik powinien zmieniać się od `1` do `n`, dla drugiego rzędu od `1` do `n-1`, dla trzeciego rzędu od `1` do `n-2`, dla czwartego rzędu od `1` do `n-3`. A jaki jest właściwie numer rzędu? Numer rzędu zliczany jest przez licznik `i`. A zatem możemy ogólnie napisać, że rząd numer `i` powinien mieć `n-i+1` kwadratów. Łatwo ten wynik sprawdzić, wstawiając kolejno za `i` numery rzędów. Oznacza to, że ćwiczenie można zrealizować skryptem:

```
n = 8;
kwadrat='FrFrFrFrff';
siatka = '';
for i = 1:n
   for j = 1:n-i+1
       siatka = [siatka kwadrat];
   end
   siatka = [siatka 'rr'];
   for k = 1:n-i+1
       siatka = [siatka 'ff'];
   end
   siatka = [siatka 'lffl'];
end
mrowkaGo(siatka)
```

Zauważmy, że licznik `k` musi „przebiegać" ten sam zakres, co licznik `j`. Identyczny efekt uzyskamy zmieniając liczniki `j` i `k` w taki sposób:

```
...
for j = i:n
...
for k = i:n
...
```

Używaliśmy wartości licznika `i` pochodzącego z pętli zewnętrznej do ustawienia wartości liczników `j` i `k` w pętli wewnętrznej. Trzeba pamiętać, że odwrotna czynność nie jest możliwa. Licznik pętli zewnętrznej „nie wie" nic o liczniku pętli wewnętrznej. Rozumowanie to można przenieść dalej, na potrójne, poczwórne i tak dalej, pętle. Natomiast w obrębie jednego poziomu pętli możliwe i bardzo wygodne jest stosowanie tego samego licznika w różnych pętlach `for`. W naszym przypadku, zamiast używać dwóch liczników: `j` i `k`, w zupełności wystarczy jeden licznik `j`.Jest to możliwe dlatego, że pętle `for` w obrębie tego samego *poziomu zagnieżdżenia* działają niezależnie od siebie, w innych przedziałach czasu. Liczniki pętli z różnego poziomu zagnieżdżenia „pracują" w tym samym momencie i nie można ich pomieszać.

2.5 Funkcje

Zaletą skryptu jest to, że łatwo jest wprowadzać pewne zmiany w komendach takie jak na przykład liczby powtórzeń. Istnieje jednak jeszcze lepsza metoda, która umożliwia to samo bez konieczności modyfikowania skryptu. Polega to na tym, że zamiast skryptu definiujemy funkcję, której można „podać" paczki, na których będzie ona operować. W celu wyjaśnienia tego, posłużymy się przykładem. Pamiętamy ćwiczenie narysowania siatki trójkątnej. Przepiszemy teraz skrypt realizujący to ćwiczenie, dodając do niego dwie nowe linie i usuwając jedną komendę:

```
function siatkatr(n)

    kwadrat='FrFrFrFrff';
    siatka = '';
    for i = 1:n
       for j = 1:n-i+1
           siatka = [siatka kwadrat];
       end
       siatka = [siatka 'rr'];
       for k = 1:n-i+1
           siatka = [siatka 'ff'];
       end
       siatka = [siatka 'lffl'];
    end
    mrowkaGo(siatka)

end
```

Najważniejsza jest pierwsza linia, zawierająca komendę `function`. Jest to tak zwany *nagłówek funkcji*. Tekst `siatkatr` jest nazwą definiowanej funkcji. Musi być ona identyczna z nazwą pliku, w którym jest zapisana. Dlatego powinniśmy od razu nadać plikowi nazwę `siatkatr.m`. Przed wyjaśnieniem znaczenia dodatkowych komend dobrze jest zobaczyć, jak korzysta się z funkcji. Otóż wystarczy wpisać w linii komend Octave nazwę funkcji, podając w nawiasie liczbę określającą pożądany wymiar siatki:

```
>> siatkatr(8);
```

Widzimy obrazek siatki trójkątnej z ośmioma rzędami. Wypróbujmy teraz komendę:

```
>> siatkatr(4);
```

Narysowana została siatka trójkątna z czterema rzędami.

Jak widać, uzyskaliśmy możliwość wpływania na liczbę powtórzeń bez wprowadzania zmian w pliku.

To, co znajduje się wewnątrz nawiasów `( )`, nazywamy *parametrem przekazywanym do funkcji*. Jest to profesjonalne określenie paczki, z której korzystać będzie funkcja, a której wartość nadawana jest z zewnątrz funkcji. W tym przypadku wartość przechowywaną w paczce `n` ustala się w momencie *wywołania funkcji* w linii komend Octave. Dlatego usunęliśmy z pliku linię:

```
n = 8;
```

Nie jest ona potrzebna, gdyż wartość `n` nadawana jest właśnie w linii komend Octave.

Ćwiczenie 2.18. Napisz kod komendy `repeat`.

Znajomość pętli `for` oraz techniki umieszczania kodu programu wewnątrz funkcji pozwala na samodzielne opracowanie komendy `repeat`. Pamiętamy, że komenda `repeat` wymaga podania dwóch rzeczy: paczki, którą chcemy zwielokrotnić oraz liczbę powtórzeń tej paczki. A zatem nagłówek funkcji realizującej komendę `repeat` ma następującą postać:

```
function repeat(paczka, ile_razy)
```

Pierwsza paczka o nazwie `paczka` zawiera ciąg poleceń dla mrówki. Druga paczka o nazwie `ile_razy` zawiera liczbę, która określa ile razy należy powtórzyć pierwszą paczkę. Wynik powtórzeń przechowywać będziemy w paczce o nazwie `duzapaczka`. Na początku paczka jest pusta:

```
duzapaczka = '';
```

Następnie dodajemy do niej zawartość paczki:

```
duzapaczka = [duzapaczka paczka];
```

Chcemy to zrobić dokładnie `ile_razy` razy, wobec tego stosujemy pętlę `for`:

```
for i=1:ile_razy
    duzapaczka = [duzapaczka paczka];
end
```

Zbierając wszystko w całość, otrzymujemy kod funkcji, który umieszczamy w pliku tekstowym o nazwie `repeat.m`:

```
function repeat(paczka, ile_razy)
    duzapaczka = '';
    for i=1:ile_razy
        duzapaczka = [duzapaczka paczka];
    end
end
```

Po napisaniu funkcji, należy ją przetestować, na przykład tak:

```
>> wynik = repeat('Ff', 3)
```

Niestety, otrzymujemy komunikat błędu. Octave informuje, że funkcja `repeat` nie zwróciła wyniku. Stało się tak dlatego, że pisząc kod funkcji nie sprecyzowaliśmy, co funkcja ma zwrócić „na zewnątrz" jako rezultat swojego działania. Sprowadza się to do dopisania w nagłówku funkcji nazwy paczki, która będzie traktowana jako rezultat działania funkcji:

```
function duzapaczka = repeat(paczka, ile_razy)
    duzapaczka = '';
    for i=1:ile_razy
        duzapaczka = [duzapaczka paczka];
    end
end
```

Powyższy kod jest już kompletny. Zawartość, jaka znajdzie się w paczce `duzapaczka` przed zakończeniem działania funkcji, zwracana jest jako rezultat działania funkcji. Można wynik ten zapamiętać w innej paczce, stosując przypisanie:

```
>> wynik = repeat('Ff', 3)
```

lub wyświetlając od razu wynik:

```
>> mrowkaGo(repeat('Ff', 3))
```

Ćwiczenie 2.X.* Uzupełnij zestaw poleceń dla mrówki tak, aby mogła ona poruszać się w przestrzeni trójwymiarowej. Zwróć uwagę na to, aby mrówka mogła osiągnąć dowolny punkt tej przestrzeni.

2.6 Instrukcja warunkowa `if`, czyli podejmowanie decyzji

Bardzo przydatną komendą w Octave jest `randi`. Jej działanie łatwo zrozumieć, wyobrażając sobie kapelusz, do którego wrzucamy karteczki oznaczone liczbami od `1` do dowolnej wybranej liczby naturalnej. Komenda `randi` zwraca rezultat przypadkowego wyboru jednej karteczki z kapelusza. Przykładowo:

```
>> randi(36)
```

odpowiada kapeluszowi z trzydziestoma sześcioma karteczkami oznaczonymi liczbami od `1` do `36`. Octave zwraca liczbę, którą oznaczona jest wylosowana karteczka. Można sprawdzić, że za każdym uruchomieniem komendy `randi(36)` Octave zwraca zazwyczaj inną liczbę, co zgadza się z naszym wyobrażeniem przypadkowego wyboru karteczki z kapelusza.

Zatrzymajmy się na chwilę przy sposobie, w jaki Octave zwraca wynik. Po wpisaniu:

```
>> randi(36)
```

Octave wypisuje na przykład:

```
ans =  29
```

Paczka `ans` jest specjalną paczką środowiska Octave, w której pamiętany jest wynik ostatniej komendy. Jej zawartość wypisywana jest tylko wtedy, gdy wynik ostatniej komendy nie został zapamiętany w żadnej innej paczce i do tego ostatnia komenda nie została zakończone znakiem `;`. Jeśli zapamiętamy wynik losowania karteczki z kapelusza w jakiejś paczce, Octave nie będzie pokazywać nam zawartości paczki `ans`:

```
>> pole = randi(36)
```

To tyle o `ans`, wróćmy do kapelusza. Ciekawym przypadkiem szczególnym losowania z kapelusza jest:

```
>> randi(2)
```

Jest to kapelusz, w którym umieszczono tylko dwie karteczki, jedną oznaczoną jedynką i drugą oznaczoną dwójką. W takim przypadku lepiej wyobrazić sobie monetę, zamiast kapelusza. Można się umówić, że jedynka odpowiada jednej stronie monety a dwójka drugiej stronie monety. A jeżeli ktoś lubi, aby `0` oznaczało orła a `1` reszkę, można od wyniku odjąć `1`:

```
>> wynik = randi(2) - 1
```

W ten sposób zamiast jedynki w paczce `wynik` znajdzie się `0` a zamiast dwójki w paczce `wynik` znajdzie się `1`.

Jeszcze wygodniejsza byłaby w takim przypadku funkcja `rzutmoneta` ,która wypisuje po prostu komunikat `orzel` lub `reszka`. Spróbujemy ją napisać.

Wypisanie komunikatu tekstowego w linii komend Octave realizuje komenda `disp('tekst')`:

```
>> disp('reszka');
```

Potrzebny jest jeszcze mechanizm, który pozwalałby na wykonanie komendy `disp('reszka')` lub `disp('orzel')` w zależności od tego, czy wynikiem rzutu wirtualną monetą będzie `1` lub `2`. Mechanizm taki zapewnia instrukcja warunkowa `if`, której szablon jest następujący:

```
if zdarzenie komenda1 else komenda2 end
```

Bardziej elegancko i przejrzyście jest zapisać to w kilku liniach:

```
if zdarzenie
    komenda1
else
    komenda2
end
```

Dla Octave nie ma żadnej różnicy, a człowiekowi łatwiej jest odczytać ten zapis. Przykładem użycia instrukcji warunkowej jest:

```
>> wynik = randi(2);
>> if wynik == 1 disp('reszka') else disp('orzel') end
```

lub zapisane w elegancki sposób:

```
if wynik == 1
    disp('reszka')
else
    disp('orzel')
end
```

Podwójny znak równości `==` oznacza porównanie. Gdybyśmy napisali `wynik = 1`, Octave w paczce `wynik` umieściłby jedynkę. Napisanie `wynik == 1` jest dla Octave poleceniem sprawdzenia, czy w paczce `wynik` znajduje się jedynka. Jeżeli w paczce `wynik` faktycznie znajduje się jedynka, to Octave wykonuje tylko pierwszą komendę, tj. `disp('reszka')` pomijając drugą komendę. Jeżeli natomiast w paczce `wynik` znajduje się cokolwiek innego niż jedynka, pierwsza komenda zostaje zignorowana a wykona się tylko druga komenda, tj. `disp('orzel')`. Całość umieścimy w jednym pliku tekstowym `rzutmoneta.m` , jako funkcję:

```
function rzutmoneta()
    wynik = randi(2);
    if wynik == 1
        disp('reszka')
    else
        disp('orzel')
    end
end
```

Funkcję uruchamiamy w linii komend Octave:

```
>> rzutmoneta
```

Wielokrotne uruchamianie funkcji upewnia nas, że faktycznie zachowuje się ona jak rzucana moneta.

Ćwiczenie 2.18. Napisz funkcję `spacer`, która utworzy rozkaz wykonania `n` kroków, przy czym `n` ma być parametrem przekazywanym do funkcji. Każdy krok ma zostawić ślad (polecenie `F`). Przed wykonaniem każdego kroku mrówka rzuca monetą. Jeżeli wypadnie jedna strona monety, to mrówka obraca się w lewo a

jeżeli wypadnie druga strona monety, to mrówka obraca się w prawo. Funkcja ma zwrócić na zewnątrz paczkę zawierającą gotowy rozkaz dla mrówki.

Rozwiązywanie zadania rozpoczniemy od napisania nagłówka funkcji w pliku `spacer.m`:

```
function rozkaz = spacer(n)
```

Paczka `n` to zadana liczba kroków, jakie ma wykonać mrówka. Funkcja zwraca na zewnątrz paczkę `rozkaz`, w której zostanie zapisany rozkaz dla mrówki. Na początku rozkaz nie zawiera żadnych poleceń:

```
rozkaz = '';
```

Następnie do paczki `rozkaz` dodawane będą kolejne pary poleceń < losowy obrót, ruch do przodu >. Par poleceń ma być `n`, a zatem potrzebne będzie użycie pętli `for`:

```
for licznik = 1:n
    tu zaraz będą komendy;
end
```

W miejscu pętli przeznaczonym na komendy obsłużymy dwie czynności mrówki: losowy obrót i następujący po nim krok do przodu. Najpierw trzeba rzucić monetą:

```
rzut = randi(2);
```

Umówmy się dla przykładu, że jeżeli została wylosowana jedynka, to mrówka obraca się w lewo, a jeżeli wypadła dwójka, to obraca się w prawo. A zatem, jeżeli w paczce `rzut` zostanie umieszczone `1`, to do paczki `rozkaz` trzeba dołączyć polecenie `'l'`. W przeciwnym przypadku do paczki `rozkaz` dołączamy polecenie `'r'`:

```
if rzut == 1
    rozkaz = [rozkaz 'l'];
else
    rozkaz = [rozkaz 'r'];
end
```

Po wykonaniu obrotu dokładamy jeszcze polecenie `'F'`:

```
rozkaz = [rozkaz 'F'];
```

Podsumowując, plik tekstowy o nazwie `spacer.m` zawiera poniższy kod:

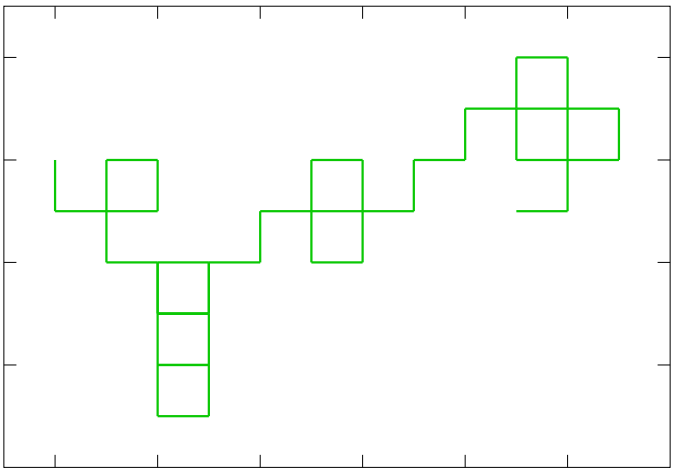
```
function rozkaz = spacer(n)
    rozkaz = '';
    for licznik = 1:n
        rzut = randi(2);
        if rzut == 1
            rozkaz = [rozkaz 'l'];
        else
            rozkaz = [rozkaz 'r'];
        end
        rozkaz = [rozkaz 'F'];
    end
end
```

Przykładowe wykorzystanie funkcji `spacer.m` z linii komend Octave (wynik na rysunku 15:

```
>> s = spacer(50);
>> mrowkaGo(s);
```

lub w jednej linii:

```
>> mrowkaGo(spacer(50));
```



Rys. 15. Przykładowy rezultat wykonania komendy `mrowkaGo(spacer(50));`.

Ćwiczenie 2.19. Zmodyfikuj funkcję `spacer` w taki sposób, aby przy losowaniu obrotu możliwe były trzy wyniki: obrót w lewo i prawo (jak dotychczas) oraz dodatkowo brak obrotu.

Modyfikacje trzeba będzie wprowadzić w dwóch miejscach. Po pierwsze, zamiast rzutu monetą (`rzut = randi(2)`) potrzebować będziemy kapelusza z trzema karteczkami:

```
wynik = randi(3);
```

Po drugie, instrukcja warunkowa `if` powinna uwzględnić dodatkową, trzecią możliwość. Można to zrobić poprzez zagnieżdżenie instrukcji `if` w uprzednio napisanej instrukcji `if`:

```
if wynik == 1
    rozkaz = [rozkaz 'l'];
else
    % tutaj będzie zagnieżdżona instrukcja if
    if wynik == 2
        rozkaz = [rozkaz 'r'];
    else
        % no właśnie, co powinno być tutaj?
    end
end
```

Octave „zrozumie” powyższy kod następująco. Jeżeli w rezultacie losowania w paczce `wynik` znalazła się jedynka, mrówka ma obrócić się w lewo. W przeciwnym przypadku wykona kolejną instrukcję warunkową `if`. W tej kolejnej instrukcji ponownie sprawdza paczkę `wynik`, ale tym razem bada czy znalazła się w niej dwójka. Jeżeli tak, to mrówka ma obrócić się w prawo. Jeżeli nie, to mrówka nie wykonuje żadnego obrotu. Jak zapisać komendę nie wykonywania obrotu? Po prostu, nie pisać żadnej komendy, zostawiając między `else` a `end` puste miejsce. W takim wypadku Octave nie potrzebuje nawet komendy `else`. W uproszczonym zapisie kod wygląda tak:

```
if wynik == 1
```

```
    rozkaz = [rozkaz 'l'];
else
    if wynik == 2
        rozkaz = [rozkaz, 'r'];
    end
end
```

Można połączyć słowo `else` z zagnieżdżonym `if`, uzyskując równoważny kod:

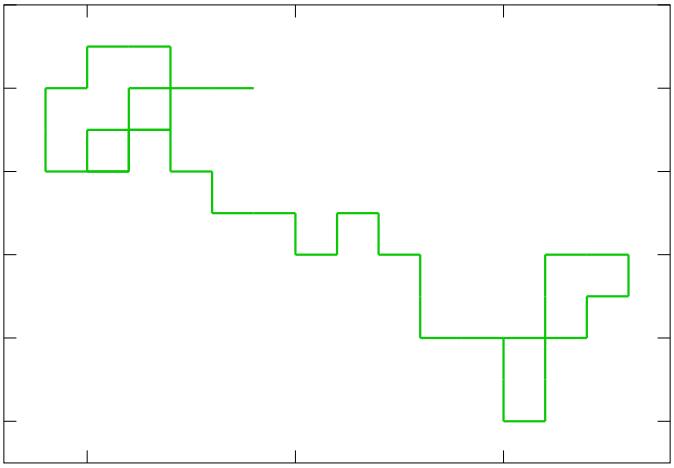
```
if wynik == 1
    rozkaz = [rozkaz 'l'];
elseif wynik == 2
    rozkaz = [rozkaz 'r'];
end
```

Ostatnia konstrukcja jest bardzo wygodna, gdyż pozwala na elegancki zapis wielokrotnego wyboru. Funkcja `spacer` po tych zmianach wygląda tak:

```
function rozkaz = spacer(n)
    rozkaz = '';
    for licznik = 1:n
        wynik = randi(3);
        if wynik == 1
            rozkaz = [rozkaz 'l'];
        elseif wynik == 2
            rozkaz = [rozkaz 'r'];
        end
        rozkaz = [rozkaz 'F'];
    end
end
```

Porównaj efekt pracy mrówki przedstawiony na rysunkach 15 i 16.

Ćwiczenie 2.20. Napisz funkcję spacer w taki sposób, aby mrówka wykonała dokładnie `n` poleceń, przy czym każde polecenie ma być losowo wybrane spośród czterech dostępnych: `l`, `r`, `f`, `F`.



Rys. 16. Przykładowy rezultat wykonania komendy `mrowkaGo(spacer(50));`.

Nagłówek funkcji będzie taki sam jak poprzedni. Zmieni się nieco jedynie wnętrze pętli `for`. Przede wszystkim w paczce `wynik` umieszczamy wynik losowania spośród czterech możliwości:

```
wynik = randi(4);
```

Teraz instrukcja warunkowa `if` składa się z jednego `if` i następujących po nim kilku „stopni” `elseif` :

```
if wynik == 1
    rozkaz = [rozkaz 'l'];
elseif wynik == 2
    rozkaz = [rozkaz 'r'];
elseif wynik == 3
    rozkaz = [rozkaz 'f'];
elseif wynik == 4
    rozkaz = [rozkaz 'F'];
end
```

Przy każdym `if` lub `elseif` Octave sprawdza, czy w paczce `wynik` znajduje się określona liczba. Co się dzieje, jeżeli w paczce `wynik` znalazła się jedynka? Wówczas już sprawdzenie `wynik == 1` da rezultat pozytywny, co spowoduje, że do rozkazu

mrówki doda się polecenie obrotu w lewo. Dalej zawartość paczki `wynik` nie będzie już sprawdzana i żadna dalsza komenda się nie wykona. A co, jeżeli w paczce `wynik` jest dwójka? Wtedy sprawdzenie przy pierwszym `if` da rezultat negatywny a zatem pierwsza komenda nie będzie wykonywana. Natomiast kolejne sprawdzenie `wynik == 2` da rezultat pozytywny, co spowoduje wykonanie drugiej komendy, czyli dodanie polecenia obrotu w prawo do rozkazu mrówki. Dalej zawartość paczki `wynik` nie będzie sprawdzana i kolejne komendy nie zostaną wykonane. Gdy w paczce `wynik` znajdzie się trójka, dopiero sprawdzenie `wynik == 3` przy `elseif` da rezultat pozytywny. Analogicznie byłoby z czwórką. Jeżeli jakimś sposobem w paczce `wynik` znalazłoby się cokolwiek innego niż `1`, `2`, `3` lub `4` (na przykład przez pomyłkę mógłby ktoś napisać `wynik = randi(5);`), to wtedy żadne ze sprawdzeń nie dałoby rezultatu pozytywnego i żadna komenda nie wykonałaby się.

Opisany wyżej sposób działania instrukcji warunkowej `if` powoduje, że kolejność sprawdzeń nie ma wpływu na rezultat jej działania. Przykładowo, poniższa instrukcja `if` jest równoważna poprzedniej:

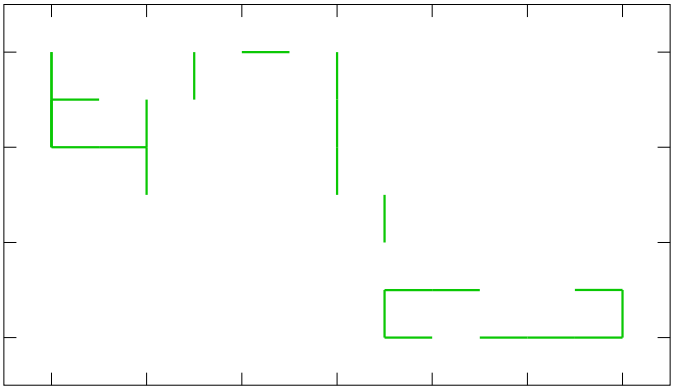
```
if wynik == 2
    rozkaz = [rozkaz 'r'];
elseif wynik == 4
    rozkaz = [rozkaz 'F'];
elseif wynik == 3
    rozkaz = [rozkaz 'f'];
elseif wynik == 1
    rozkaz = [rozkaz 'l'];
end
```

Podsumowując, rozwiązaniem zadania jest plik tekstowy o nazwie `spacer.m`, zawierający kod:

```
function rozkaz = spacer(n)
    rozkaz = '';
    for licznik = 1:n
        wynik = randi(4);
        if wynik == 1
            rozkaz = [rozkaz 'l'];
        elseif wynik == 2
```

```
            rozkaz = [rozkaz 'r'];
        elseif wynik == 3
            rozkaz = [rozkaz 'f'];
        elseif wynik == 4
            rozkaz = [rozkaz 'F'];
        end
    end
end
```

Przykładowy wynik przedstawiony jest na rysunku 17.



Rys. 17. Przykładowy rezultat wykonania komendy `mrowkaGo(spacer(100));`.

Zastanówmy się jeszcze nad tym, czy ostatnie sprawdzenie `wynik == 4` ma sens. Po wykonaniu komendy

```
wynik = randi(4);
```

w paczce `wynik` może znaleźć się tylko `1, 2, 3,` albo `4`. Nie ma innych możliwości. Skoro Octave sprawdził, że w paczce `wynik` nie ma jedynki, dwójki ani trójki, to z pewnością jest tam czwórka. Nie ma już potrzeby sprawdzać tego faktu. Funkcja `spacer` może więc równie dobrze być napisana tak:

```
function rozkaz = spacer(n)
    rozkaz = '';
    for licznik = 1:n
        wynik = randi(4);
        if wynik == 1
```

```
            rozkaz = [rozkaz 'l'];
        elseif wynik == 2
            rozkaz = [rozkaz 'r'];
        elseif wynik == 3
            rozkaz = [rozkaz 'f'];
        else wynik == 4
            rozkaz = [rozkaz 'F'];
        end
    end
end
```

Jedyna różnica polega na tym, że ostatnie sprawdzenie związane jest ze słowem `else` zamiast `elseif`zdarzenie .

Po sprawdzeniu można wykonać kilka komend. Sprawdź działanie takiej wersji funkcji `spacer`:

```
function rozkaz = spacer(n)
    rozkaz = '';
    for licznik = 1:n
        wynik = randi(4);
        if wynik == 1
            rozkaz = [rozkaz 'l'];
            disp('obrot w lewo');
        elseif wynik == 2
            rozkaz = [rozkaz 'r'];
            disp('obrot w prawo');
        elseif wynik == 3
            rozkaz = [rozkaz 'f'];
            disp('do przodu');
        else
            rozkaz = [rozkaz 'F'];
            disp('do przodu slad');
        end
    end
end
```

2.7 Co to jest program?

Wszystkie użyteczne fragmenty kodu, czyli takie, które „coś robią", nazywamy **programami**. W szczególności, skrypty i funkcje Octave są programami. A zatem programy są po prostu specjalnym rodzajem tekstu, który można napisać w edytorze tekstu lub nawet na kartce papieru. Wartość tego tekstu polega na tym, że jest on tak napisany, aby mogła go odczytać maszyna taka jak komputer, robot, tablet, smartfon. A co to znaczy, że maszyna odczytuje program? To znaczy, że maszyna znajduje w podanym jej tekście słowa-klucze powodujące, że wykona ona pewne działanie, które ktoś wcześniej przypisał tym słowom. Przykładem maszyny jest mrówka, która zna cztery słowa klucze: `F`, `f`, `l`, `r`. Programem dla mrówki jest na przykład `'FflFlFFrFfF'`. Pisaliśmy też bardziej skomplikowane programy.

Programy składają się z

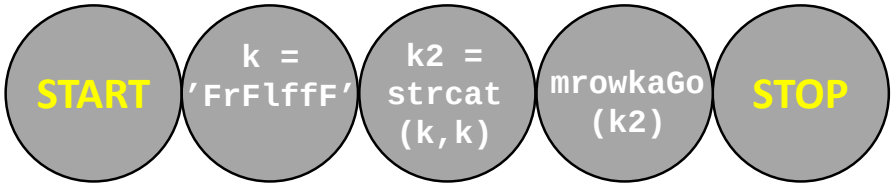
- komend,
- pętli,
- instrukcji warunkowych.

Jak widać, reguł tworzenia programu nie ma zbyt dużo. Pomimo tego programowanie jest sztuką, na opanowanie której trzeba poświęcić trochę czasu. Tę sztukę może opanować każdy, wystarczy tylko chcieć uruchomić nieco wyobraźni.

Istnieją bardzo wygodne sposoby graficznego przedstawienia działania programu, wspomagające wyobraźnię i ułatwiające napisanie i zrozumienie programu. Na początek zaczniemy od pokazania jak można zilustrować program składający się z samych komend, bez pętli i instrukcji warunkowych. Niech będzie to poniższy skrypt:

```
k = 'FrFlffF';
k2 = [k k];
mrowkaGo(k2);
```

Rysunek 18 jest ilustracją tego skryptu. Przypomina to gry planszowe polegające na przebyciu pionkiem wszystkich pól począwszy od pola START do pola STOP. Zazwyczaj o liczbie pól przebytych przez pionek w jednej turze decyduje rzut kostką, tutaj jednak pionek porusza się w każdej turze tylko o jedno pole. Niektóre



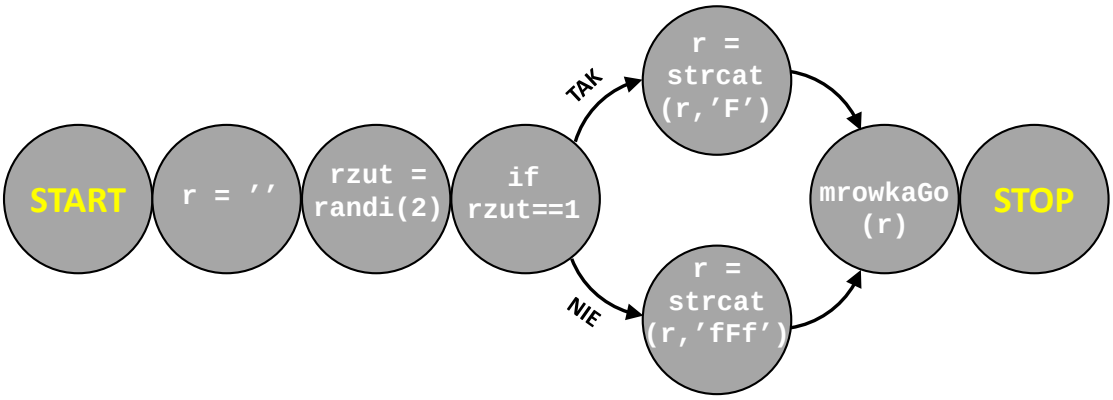
Rys. 18. Ilustracja graficzna prostego programu.

pola w grach planszowych mają znaczenie specjalne. Po zatrzymaniu się na takim polu pionek wykonuje pewną dodatkową akcję, na przykład wraca się o trzy pola.

Każdy program możemy wyobrażać sobie jako grę planszową. Na jej polach znajdują się komendy dla maszyny. Pola specjalne odpowiadają instrukcjom warunkowym. Pokażemy teraz, jak można zilustrować program z instrukcją warunkową `if`:

```
r = '';
rzut = randi(2);
if rzut == 1
    r = [r 'F'];
else
    r = [r 'fFf'];
end
mrowkaGo(r);
```

Od pola z instrukcją `if` rozpoczyna się rozgałęzienie. O tym, którą z dróg wybierze pionek, decyduje wynik sprawdzenia.

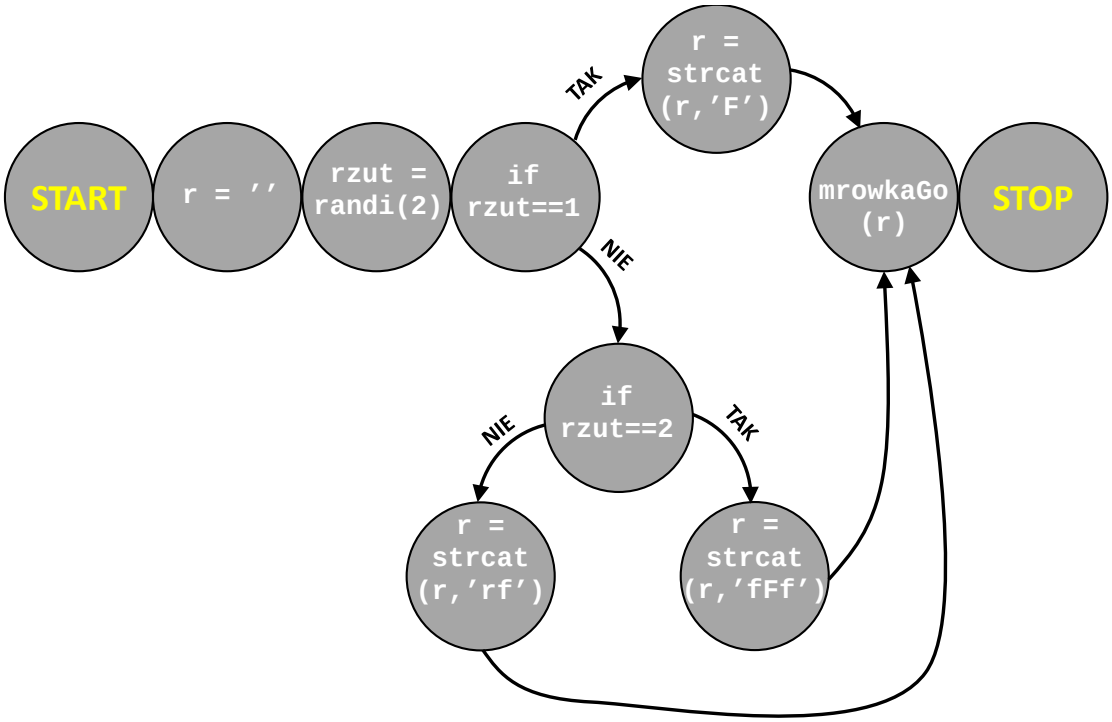


Rys. 19. Ilustracja graficzna programu z instrukcją warunkową.

Rozbudujmy instrukcję warunkową:

```
r = '';
rzut = randi(3);
if rzut == 1
    r = [r 'F'];
elseif rzut == 2
    r = [r 'fFf'];
else
    r = [r 'rf'];
end
mrowkaGo(r);
```

Ilustracja graficzna przedstawiona jest na rysunku 20.



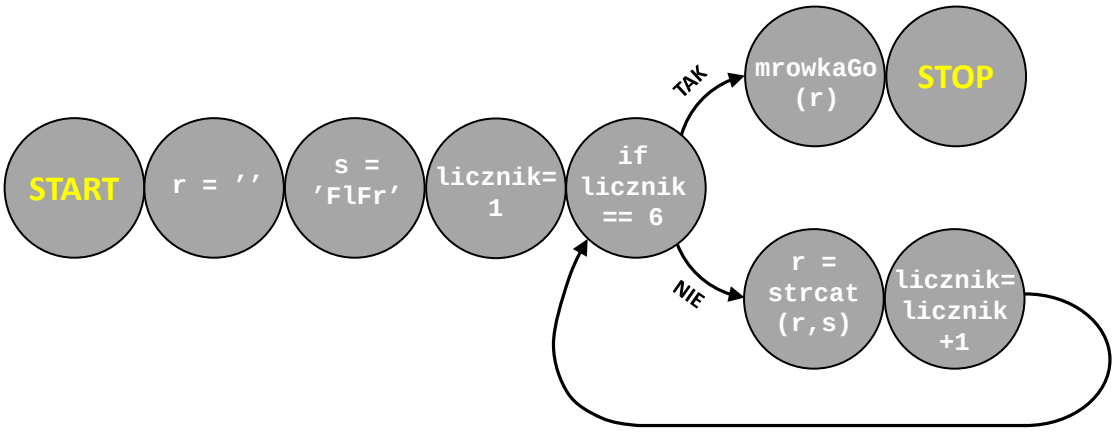
Rys. 20. Ilustracja graficzna programu z rozbudowaną instrukcją warunkową.

Teraz zobaczymy, jak zilustrować program z pętlą, na przykład taki:

```
r = '';
```

```
s = 'FlFr';
for licznik = 1:5
    r = [r s];
end
mrowkaGo(r);
```

Okazuje się, że przy odrobinie sprytu można wykonać „planszę do gry w pętlę” `for` korzystając tylko z poznanej metody ilustrowania instrukcji warunkowej `if`. Rozwiązanie podane na rysunku 21 jest jednym z wielu możliwych. Zachęcamy do wymyślania innych rozwiązań.



Rys. 21. Ilustracja graficzna programu z pętlą `for`.

2.8 Pętla `while`

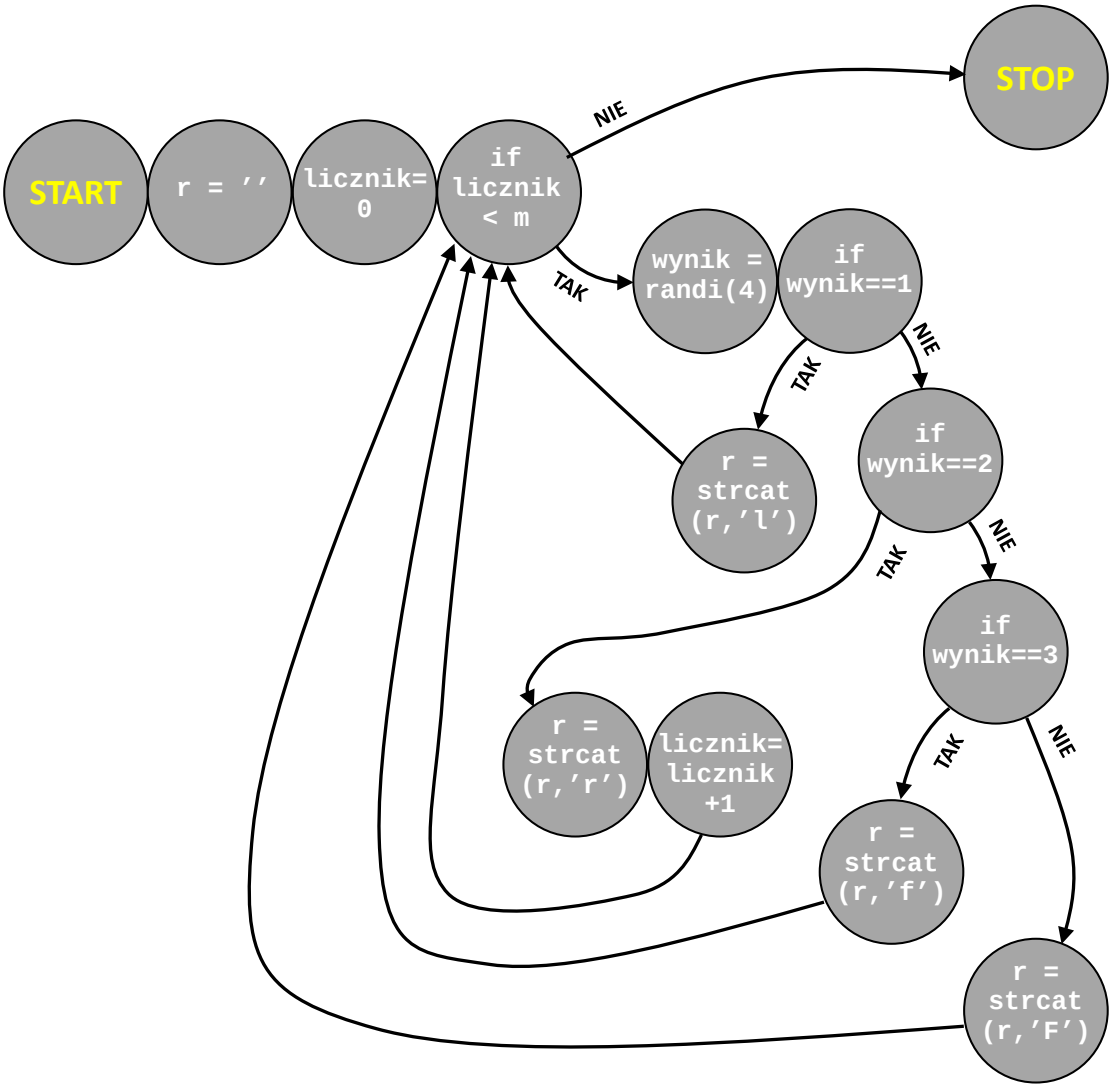
Charakterystyczną cechą pętli `for` jest to, że z góry wiadomo ile powtórzeń zostanie wykonanych. Przykładowo, pętla

```
for licznik = 1:10
```

ma dokładnie dziesięć powtórzeń. Są jednak sytuacje, gdy liczby powtórzeń w pętli nie da się z góry określić.

Ćwiczenie 2.21. Rozważmy funkcję `spacer` z ćwiczenia 2.20. Zmodyfikuj jej działanie w taki sposób, aby mrówka spacerowała tak długo, aż wykona dokładnie `m` obrotów w prawo.

Stosunkowo łatwo jest wykonać interpretację graficzną programu, który ma zostać napisany (patrz rysunek 22).



Rys. 22. Ilustracja graficzna programu do ćwiczenia 2.21.

Zwróćmy uwagę, że `licznik` nie jest tym razem licznikiem powtórzeń w pętli, lecz licznikiem obrotów w prawo. Licznik powtórzeń pętli nie jest tu w zasadzie potrzebny. Pętla „kręci się” tak długo, jak długo sprawdzenie `licznik < m` daje wynik pozytywny. Na początku paczka `licznik` przechowuje zero, gdyż wtedy mrówka nie wykonała jeszcze żadnego obrotu. Licznik zwiększa się tylko po doda-

niu polecenia obrotu w prawo do rozkazu dla mrówki. Gdy liczba obrotów w prawo będzie równa `m`, wówczas w paczce `licznik` znajdzie się liczba `m` i nastąpi wyjście z pętli.

Kluczowym sprawdzeniem w pętli jest `licznik < m`, czyli sprawdzenie, czy liczba zapisana w paczce `licznik` jest mniejsza od liczby zapisanej w paczce `m`. Przy okazji omówmy krótko zapisy innych porównań. Zapis `<=` oznacza „mniejsze lub równe", zapis `>=` to „większe lub równe" a zapis `!=` interpretowany jest jako „różne od".

Do wykonania ilustracji programu (nieistniejącego jeszcze) do ćwiczenia 2.21 nie potrzebowaliśmy niczego nowego ponad to, co poznaliśmy do tej pory. Jak jednak zaprogramować tę konstrukcję? Pętla `for` do tego celu się nie nadaje, gdyż polecenia dla mrówki generowane są losowo i nigdy nie wiadomo, w którym powtórzeni w pętli pojawi się `m`-te polecenie obrotu w prawo. Do realizacji zadania potrzebny będzie inny rodzaj pętli. Liczba powtórzeń w takiej pętli nie może być sztywno określona, lecz powinna zależeć od wyniku sprawdzenia, czy jakieś zdarzenia zachodzi. Wspomnianą cechę posiada pętla `while`. Jej szablon jest następujący:

```
while zdarzenie
    komendy;
end
```

Wszystko, co można zrobić za pomocy pętli `for`, da się też zrobić pętlą `while`(odwrotne zdanie nie jest prawdziwe). Przykładowo, dla pętli `for`:

```
for licznik = 1:10
    disp(licznik);
end
```

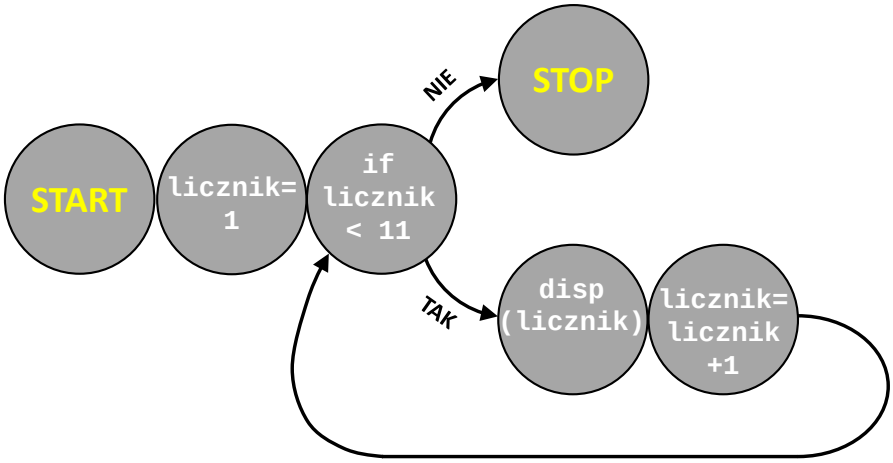
równoważny kod z pętlą `while` ma postać:

```
licznik = 1;
while licznik <= 10
    disp(licznik);
    licznik = licznik + 1;
end
```

Można to zrealizować na więcej sposobów, na przykład tak:

```
licznik = 1;
while licznik < 11
    disp(licznik);
    licznik = licznik + 1;
end
```

Ilustracja graficzna znajduje się na rysunku 23.



Rys. 23.

Zasadnicza różnica między pętlami `for` i `while` polega na tym, że pętla `for` sama zajmuje się zwiększaniem wartości licznika w każdym powtórzeniu w pętli, natomiast w pętli `while` to my kontrolujemy wartość licznika. Jeżeli zapomnielibyśmy o tym i napisali:

```
licznik = 1;
while licznik <= 10
    disp(licznik);
end
```

to program nigdy się nie zatrzyma, bo sprawdzenie `licznik == 10` nie da nigdy wyniku negatywnego. Pętla `while` działa tak długo jak długo sprawdzenie, czy zdarzenie zdefiniowane po słówku `while` zachodzi, daje wynik pozytywny.

Trzeba też pamiętać, żeby zainicjować licznik, to znaczy wpisać do niego jakąś wartość przed rozpoczęciem pętli. Jeżeli o tym zapomnimy, na przykład:

```
while licznik < 5
    disp(licznik);
end
```

to efekt może być trudny do przewidzenia i zależny od rodzaju maszyny, która taki program miała nieszczęście dostać. Niektóre maszyny mogą nawet odmówić wykonania takiego programu, z korzyścią dla siebie, programisty i niewinnych użytkowników.

Mamy już wystarczający zasób wiedzy, aby zrealizować ćwiczenie 2.21:

```
function r = spacer(m)
    r = '';
    licznik = 0;
    while licznik < m
        wynik = randi(4);
        if wynik == 1
            r = [r 'l'];
        elseif wynik == 2
            r = [r 'r'];
            licznik = licznik + 1;
        elseif wynik == 3
            r = [r 'f'];
        else
            r = [r 'F'];
        end
    end
end
```

Ćwiczenie 2.22. Rozważmy ponownie funkcję `spacer` z ćwiczenia 2.20. Zmodyfikuj jej działanie w taki sposób, aby mrówka spacerowała tak długo, aż wykona jeden pełny obrót w prawo.

Tym razem rozwiązanie przedstawimy bez komentarza. Jedną z ważnych umiejętności programisty jest zrozumienie istniejącego kodu programu.

```
function r = spacer()
    r = '';
```

```
    licznik = 0;
    while licznik < 4
        wynik = randi(4);
        if wynik == 1
            r = [r 'l'];
            licznik = licznik - 1;
        elseif wynik == 2
            r = [r 'r'];
            licznik = licznik + 1;
        elseif wynik == 3
            r = [r 'f'];
        else
            r = [r 'F'];
        end
    end
end
```

Pamiętaj, że powyższe rozwiązanie wynika z przyjęcia określonej interpretacji zdarzenia „jeden pełny obrót w prawo". Oznacza to, że samych rozwiązań jest co najmniej tyle, ile sposobów interpretacji zadania.

2.9 Zaawansowane polecenia dla mrówki

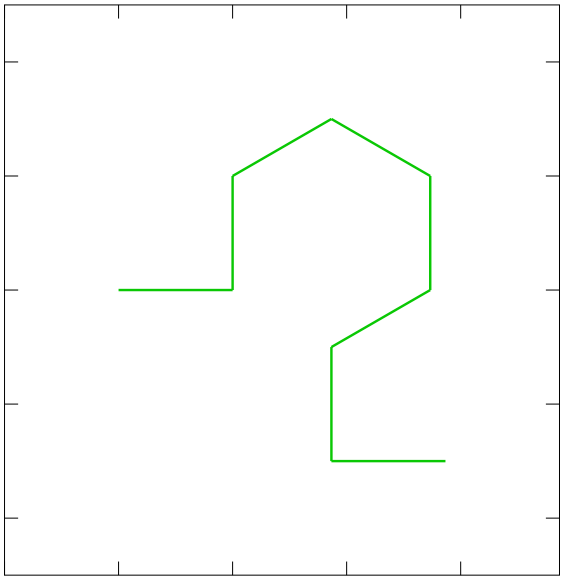
2.9.1 Obroty o dowolny kąt

Swoboda naszej mrówki jest mocno ograniczona. Może ona obracać się wyłącznie o wielokrotności dziewięćdziesięciu stopni. Jest jednak komenda `^x`, która spowoduje, że kąt obrotu mrówki ustawiany jest na wartość `x` stopni. Ustawienie to ma charakter trwały. Oznacza to, że po zmianie kąta każde kolejne polecenie obrotu wykonane zostanie z wprowadzoną wartością kąta. Oto przykład:

```
>> mrowkaGo('FlF^60rFrFrFrFlF^90lF')
```

wraz z obrazkiem wykonanym przez mrówkę (rys. 24). Liczba określająca kąt może być ułamkiem, na przykład:

```
>> mrowkaGo('FlF^60.8rFrFrFrFlF^90.95lF')
```

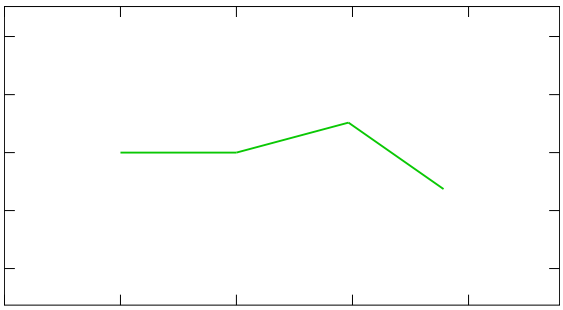


Rys. 24.

Polecenie `^x` ustawia identyczny kąt obrotu dla obrotu w lewo i prawo. Gdyby z jakiś powodów zależało nam na tym aby kąt obrotu w lewo był inny niż w prawo, można zastosować polecenia `<x` i `>x`. Polecenie `<x` ustawia tylko kąt dla obrotu w lewo, natomiast polecenie `>x` tylko kąt dla obrotu w prawo. Przykład:

```
>> mrowkaGo('<15>50FlFrF')
```

Efekty widać na rysunku 25.



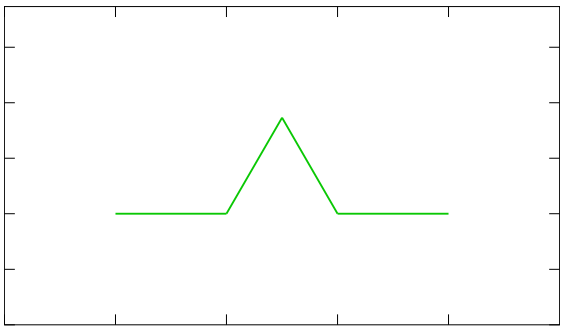
Rys. 25.

Wprowadzenie poleceń modyfikacji kątów obrotu sprawia, że – w pewnym sensie – jeden rozkaz może być wykonany na różne sposoby, w zależności od ustawień kątów obrotu.

Ćwiczenie 2.23. Porównaj efekty następujących poleceń, różniących się jedynie wartością kąta obrotu:

```
>> mrowkaGo('^45FrFrFrFrFrF')
>> mrowkaGo('^60FrFrFrFrFrF')
>> mrowkaGo('^60FrFrFrFrFrF')
>> mrowkaGo('^90FrFrFrFrFrF')
>> mrowkaGo('^105FrFrFrFrFrF')
>> mrowkaGo('^120FrFrFrFrFrF')
>> mrowkaGo('^180FrFrFrFrFrF')
```

Ćwiczenie 2.24. Ułóż rozkaz, którego efektem będzie obrazek na rysunku 26.



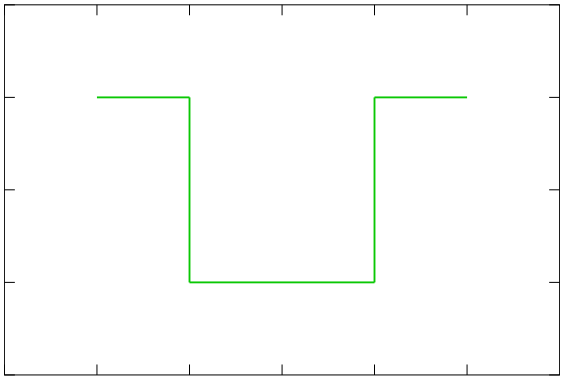
Rys. 26.

2.9.2 Modyfikacja długości kroku

Dla wygody warto mieć możliwość ustawienia nowej długości kroku. Służy do tego polecenie `|x`. Ustawia ono nową długość kroku poprzez wymnożenie dotychczasowej długości przez współczynnik `x`. Poniżej przykład ilustrujący działanie polecenia:

```
>> mrowkaGo('Fr|2FlFlFr|0.5F')
```

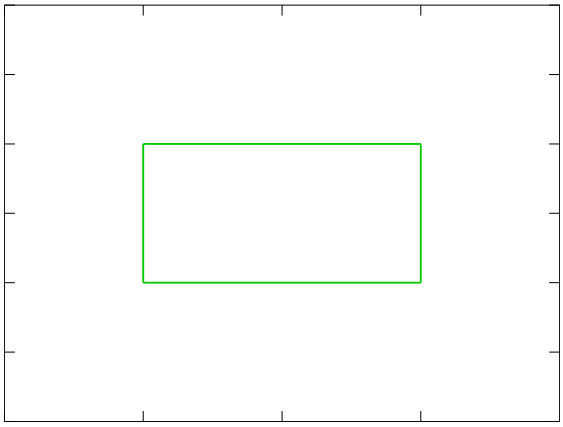
Rezultat działania przedstawia rysunek 27. Przy pierwszym użyciu polecenia `|2` pierwotna długość kroku zostaje podwojona. Mrówka wykonuje „podwójne” kroki aż do napotkania polecenia `|0.5`, które nakazuje jej zmniejszyć bieżącą długość kroku o połowę. Powoduje to, że długość kroku wykonywanego przez mrówkę wraca



Rys. 27.

do pierwotnej. W tym momencie mógłby ktoś zapytać: a jaka jest ta pierwotna długość kroku? Otóż pytanie to nie ma większego sensu. Obrazek narysowany na ekranie jest wyskalowany tak, aby zmieścić cały ślad mrówki. Jeżeli podwoimy długość kroku, skala też się odpowiednio zmieni i nie zauważymy różnicy poza jednym technicznym szczegółem: marginesy mogą się lekko przesunąć. Proponujemy samodzielne eksperymentowanie.

Ćwiczenie 2.25. Ułóż rozkaz, którego efektem będzie obrazek na rysunku 28.

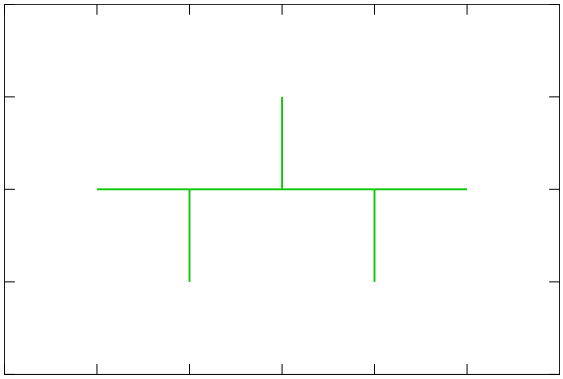


Rys. 28.

2.9.3 Klonowanie

Mrówka ma specjalną zdolność do klonowania się. Służą do tego dwa polecenia: `{` i `}`. Polecenie `{` każe mrówce wykonać swoją kopię, nazwijmy ją mrówką-córką. Mrówka-matka w tym momencie zastyga w bezruchu. Kolejne polecenia dotyczyć będą tylko mrówki-córki. Poleceniem `}` możemy zakończyć pracę mrówki-córki, która natychmiast wyparowuje. Wtedy mrówka-matka kontynuuje swoją pracę jakby nic się nie stało. Wszystkie polecenia pomiędzy `{` a `}` wykonuje mrówka-córka, a mrówka-matka nie jest ich świadoma. Oto przykład (rys. 29):

```
>> mrowkaGo('F{rF}F{lF}F{rF}F')
```



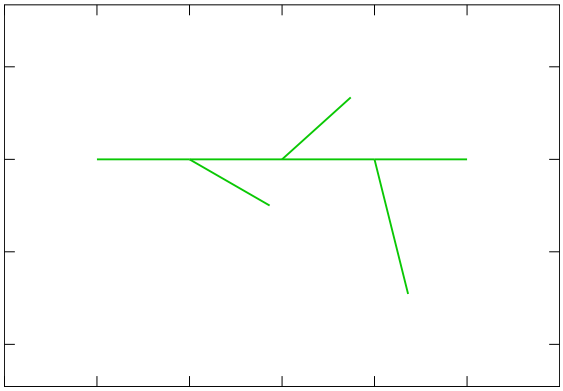
Rys. 29.

Mrówka-matka rysowała poziomą linię, natomiast kolejne córki zajmowały się dorysowywaniem pionowych kresek. Spróbuj narysować identyczny obrazek bez używania polecenia klonowania mrówki. Oto kolejny przykład (rys. 30):

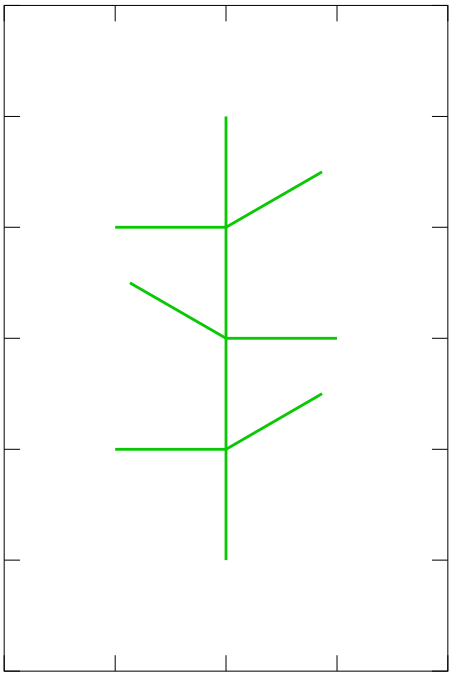
```
>> mrowkaGo('F{^30rF}F{^42lF}F{|1.5^76rF}F')
```

Zwróć uwagę, że mrówki-córki miały zmieniane kąty obrotu i długości kroku ale nie wpłynęło to na mrówkę-matkę.

Ćwiczenie 2.26. Wykonaj obrazek podobny do przedstawionego na rysunku 31 wiedząc, że mrówka może mieć dwie i więcej córek pod rząd.

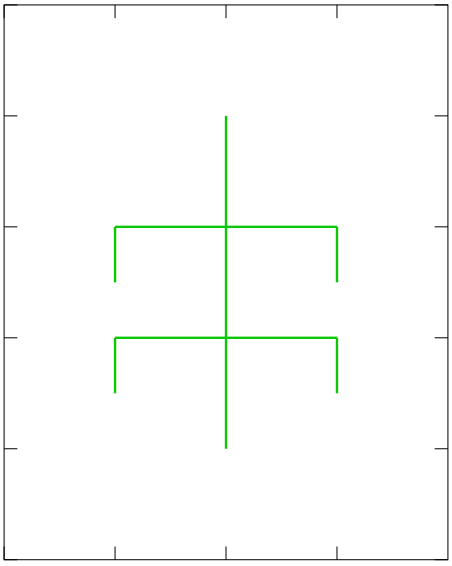


Rys. 30.



Rys. 31.

Ćwiczenie 2.27. Wykonaj obrazek podobny do przedstawionego na rysunku 32 wiedząc, że mrówka może zostać babcią.



Rys. 32.

2.10 Podmieniak

Na naszej wirtualnej mrówce czasami zamieszkuje pewien drobnoustrój, ledwie widoczny gołym mrówczym okiem. Przez fachowców został nazwany podmieniakiem.

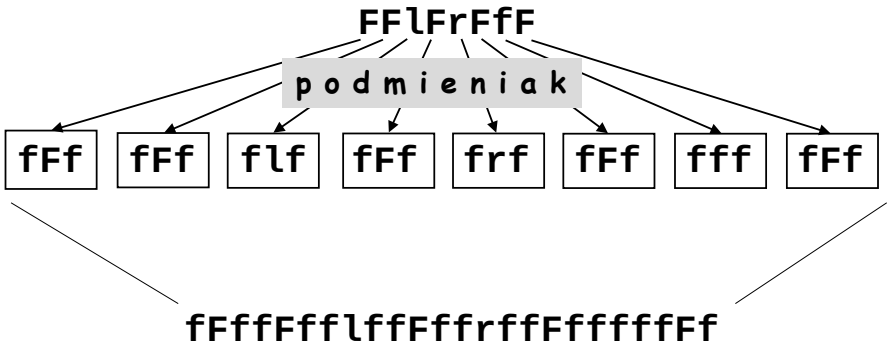
Jego ulubionym miejscem na ciele mrówki jest głowa, na której może oddawać się swojej ulubionej rozrywce. Jest nią zabawa w przekręcanie rozkazów, jakie otrzymuje mrówka. Podmieniak przechwytuje rozkaz zanim ten trafi do mózgu mrówki, błyskawicznie przekręca polecenia i przesyła mrówce podmieniony rozkaz jak gdyby nigdy nic.

Podmieniak ma naturę biurokraty. Lubi działać według jasno określonych reguł i niechętnie ucieka się do improwizacji. Sposób postępowania podmieniaka jest prosty. Bierze on rozkaz, który miał trafić do mrówki i odczytuje go znak po znaku. Każdy znak podmienia na jakiś ciąg znaków, przestrzegając przy tym ścisłych reguł. Przykładowo, jeżeli reguła podmieniania każe w miejsce `F` wstawiać `fFf`, w miejsce `f` wstawiać `fff`, zamiast `l` podstawiać `flf` a `r` zastąpić przez `frf`, to podmieniak zastosuje się do tych reguł w stu procentach. Każdy osobnik ma w móżdżku zapisaną specjalną tablicę reguł, na przykład:

```
F -> fFf
f -> fff
```

```
l -> flf
r -> frf
```

Rozważmy konkretny przykład. Co się stanie, gdy podmieniak przechwyci rozkaz `FFlFrFfF`? Poczynania podmieniaka w takim przypadku ilustruje rysunek 33.



Rys. 33.

A co zrobi podmieniak po odczytaniu znaku, którego nie ma w tablicy reguł? Po prostu zostawi ten znak w spokoju, nie usuwając go z rozkazu i nic za niego nie wstawiając. Przykład:

```
F -> lFFr
```

oraz przechwycony rozkaz: `FfF`. Podmieniak zrobi z niego `lFFrflFFr`.

Gdy podmieniak ma więcej czasu na działanie, może zrobić z przekręconym rozkazem jeszcze raz to samo, co robił z oryginałem. A więc bierze rozkaz `lFFrflFFr` i robi z niego rozkaz `llFFrlFFrrfllFFrlFFrr`. Jeśli mrówka jeszcze w niczym się nie zorientowała i w związku z tym podmieniak nadal ma trochę czasu, może jeszcze raz ... i tak dalej.

Ćwiczenie 2.28. Napisz funkcję `podmieniak`, która dla podanego rozkazu zwraca jego podmienioną postać. Zaproponuj, jakie parametry powinna mieć funkcja. Skorzystaj, w razie potrzeby, z komendy:

```
length(paczka)
```

która zwraca liczbę znaków, jakie są zapisane w paczce `paczka` (o ile oczywiście paczka zawiera znaki, a nie na przykład liczbę).

Rozwiązanie, jakie tu zaprezentujemy, należy traktować jako wstępną propozycję do własnych pomysłów. Proponujemy funkcję o postaci:

```
function podmrozkaz = podmieniak(rozkaz,ilerazy,regF,regf,regl,regr)
```

Kolejnym parametrom nadajemy następujące znaczenie: `rozkaz` to rozkaz jaki miała dostać mrówka , `ilerazy` decyduje o tym, ilekrotnie podmieniak podmieni rozkaz, `regF` to ciąg poleceń, jakim podmieniak zastąpi `F` , analogiczne znaczenie mają paczki `regf`, `regl`, `regr`. Funkcja zwraca paczkę `podmrozkaz`, w której znajdzie się końcowy rezultat pracy podmieniaka.

Na początku rozkaz nie jest jeszcze podmieniony, więc do paczki `podmrozkaz` wkładamy `rozkaz`:

```
podmrozkaz = rozkaz;
```

Podmieniak wykonuje swoją pracę dokładnie `ilerazy` razy, a zatem użyjemy pętli `for` z licznikiem `i` do obsługi powtórzeń:

```
for i = 1:ilerazy
    tutaj zaraz coś będzie;
end
```

Cząstkowe rezultaty podmieniania będziemy umieszczać w paczce `nowapaczka`, do której wstępnie wpiszemy pusty rozkaz:

```
nowyrozkaz = '';
```

Podmiana rozkazu składa się z podmian pojedynczych znaków. Ile jest tych znaków? Można policzyć je za pomocą komendy `length(podmrozkaz)`. Wobec tego pętla, która przetworzy wszystkie znaki rozkazu, musi wykonać się dokładnie tyle razy:

```
for k = 1:length(podmrozkaz)
    tutaj za chwilę coś będzie;
end
```

Zwróć uwagę, że mamy już pętlę w pętli. Licznik w pętli zewnętrznej to `i` a w pętli wewnętrznej to `j`. Każda pętla ma swój własny licznik. W pętli wewnętrznej paczka `k` jest licznikiem znaków w rozkazie. W `k`-tym kroku w pętli chcemy podmienić znak o numerze `k` w rozkazie. Jak dostać się do tego i tylko tego znaku? Można to zrobić, pisząc:

```
podmrozkaz(k)
```

Powyższego zapisu nie należy mylić z wywołaniem funkcji, choć tak właśnie korzystamy z funkcji. Octave „wie", że `podmrozkaz` jest paczką a nie funkcją i zachowa się stosownie do sytuacji. Przetestujmy ten nowy sposób korzystania z paczek:

```
>> byleco = 'FrllfFF';
>> byleco(1)
>> byleco(4)
>> length(byleco)
```

Jest to trochę tak, jakbyśmy z paczki nie wyciągali całej zawartości, a tylko pojedyncze elementy. Korzystając z nowej umiejętności, łatwo zapiszemy dalszy fragment kodu programu symulującego działanie podmieniaka. Musimy sprawdzać, jakie są kolejne znaki rozkazu i zamieniać je zgodnie z regułami podanymi w parametrach funkcji:

```
if podmrozkaz(k) == 'F'
    nowyrozkaz = [nowyrozkaz regF];
elseif podmrozkaz(k) == 'f'
    nowyrozkaz = [nowyrozkaz regf];
elseif podmrozkaz(k) == 'l'
    nowyrozkaz = [nowyrozkaz regl];
elseif podmrozkaz(k) == 'r'
    nowyrozkaz = [nowyrozkaz regr];
end
```

Gdy podmieniak skończył podmieniać rozkaz, należy wynik zapamiętać w paczce `podmrozkaz`:

```
podmrozkaz = nowyrozkaz;
```

Paczka zostanie albo zwrócona na zewnątrz jako wynik działania funkcji albo podmieniak przystąpi do ponownej podmiany (kolejny obieg pętli). Cały kod jest następujący:

```
function podmrozkaz = podmieniak(rozkaz,ilerazy,regF,regf,regl,regr)

podmrozkaz = rozkaz;

for i = 1:ilerazy
    nowyrozkaz = '';
    for k = 1:length(podmrozkaz)
        if podmrozkaz(k) == 'F'
            nowyrozkaz = [nowyrozkaz regF];
        elseif podmrozkaz(k) == 'f'
            nowyrozkaz = [nowyrozkaz regf];
        elseif podmrozkaz(k) == 'l'
            nowyrozkaz = [nowyrozkaz regl];
        elseif podmrozkaz(k) == 'r'
            nowyrozkaz = [nowyrozkaz regr];
        end
    end
    podmrozkaz = nowyrozkaz;

end
```

Ćwiczenie 2.29. Wypróbuj podmieniaka, starając się utworzyć przy jego pomocy rozkazy dające jak najciekawsze obrazki.

Dodatki

Dodatek A – Octave

Przykłady programów zostały opracowane w środowisku programistycznym Octave 3.6.2 z interfejsem graficznym GNU Octave 1.5.4. Octave jest darmowym środowiskiem programistycznym wzorowanym na środowisku MATLAB, które używane jest na całym świecie przez jednostki naukowe i duże firmy do wykonywania zaawansowanych obliczeń inżynierskich.

Najnowsze wersje Octave dla systemu Windows można pobrać ze strony:
http://sourceforge.net/projects/octave/files/Octave%20Windows%20binaries/,
natomiast dla systemu Linux ze strony:
http://www.gnu.org/software/octave/download.html.
Interfejs graficzny nie jest niezbędny do uruchomienia programów. Stanowi jedynie ułatwienie w korzystaniu z podstawowego środowiska uruchomieniowego. Dostępne są różne interfejsy graficzne dla Octave. Program instalacyjny dla interfejsu używanego przez autorów do opracowania przykładów programów można pobrać ze strony:
http://www.softpedia.com/get/Science-CAD/GUI-Octave.shtml.

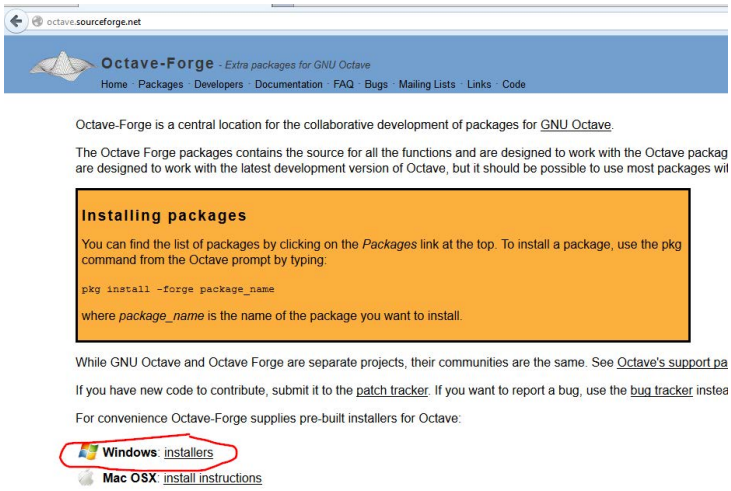
Korzystanie z GNU Octave sprowadza się do wyboru w oknie *Current Directory* bieżącego katalogu oraz wywoływania programów w oknie poleceń. W przypadku braku interfejsu graficznego, dostępne jest jedynie okno poleceń. Wówczas, w celu ustawienia bieżącego katalogu na pożądany należy posłużyć się następującym zestawem poleceń:

- `pwd` – wyświetla pełną ścieżkę bieżącego katalogu,
- `ls` lub `dir` – wyświetla nazwy katalogów i plików w bieżącym katalogu,
- `cd`katalog – zmienia bieżący katalog (nowy katalog można podać względem bieżącego katalogu lub jako pełną ścieżkę),
- `cd..` – zmienia katalog bieżący na katalog nadrzędny,
- `clc` – czyści ekran.

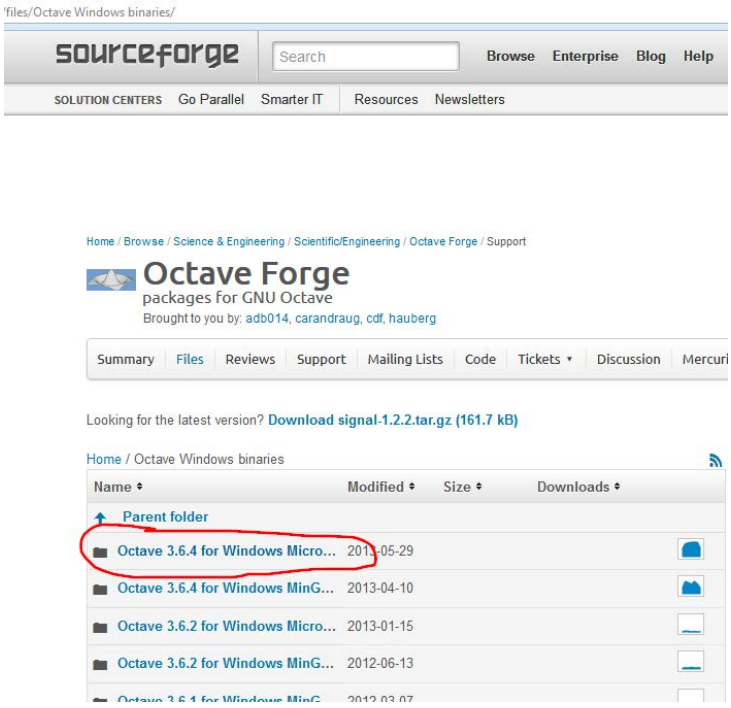
W przypadku wpisywania długich nazw przydaje się autouzupełnianie (klawisz TAB). Przydatny są również klawisze ↑, ↓, które pozwalają na poruszanie się po historii wpisywanych poleceń.

Dodatek B – Octave - instrukcja instalacji i konfiguracji

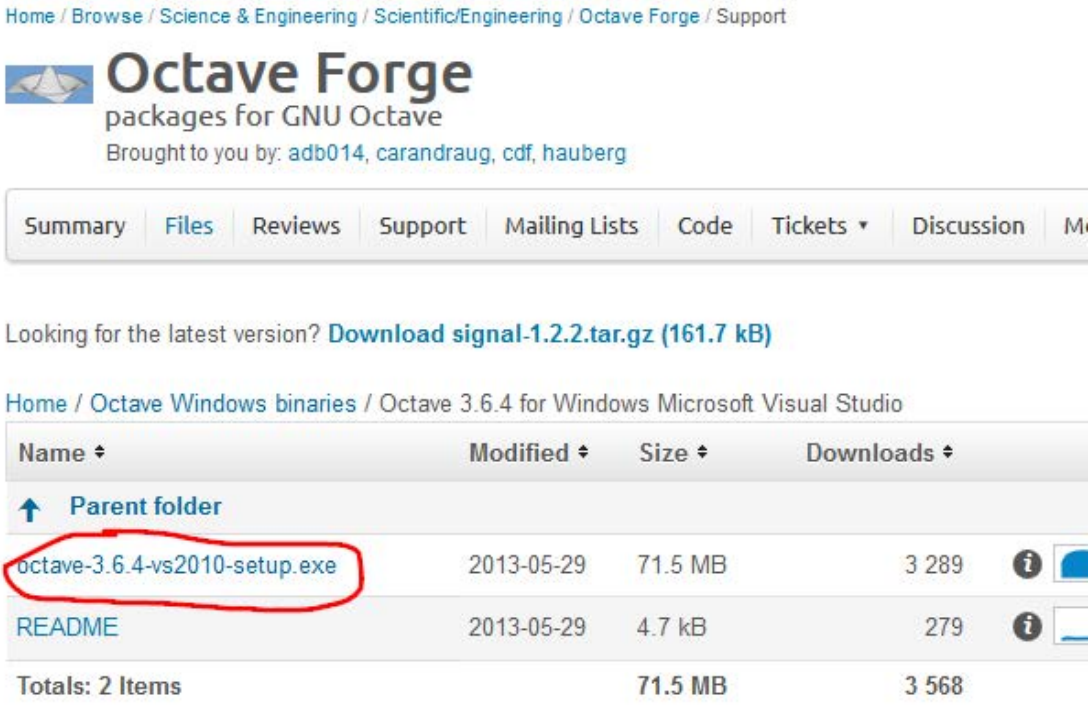
Jeżeli nie mamy jeszcze pobranego programu instalacyjnego, to w przeglądarce wpisujemy lokalizację http://octave.sourceforge.net/:



Wybieramy najnowszą wersję instalatora (w momencie pisania tej instrukcji była to wersja 3.6.4):

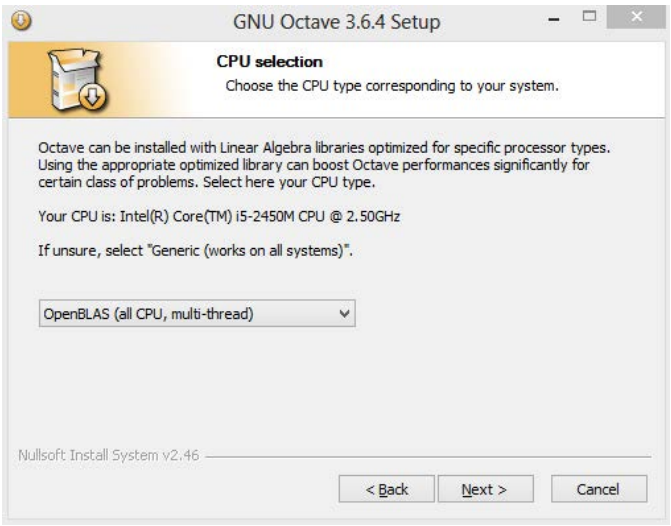


Klikamy na bezpośredni odnośnik do programu instalacyjnego:

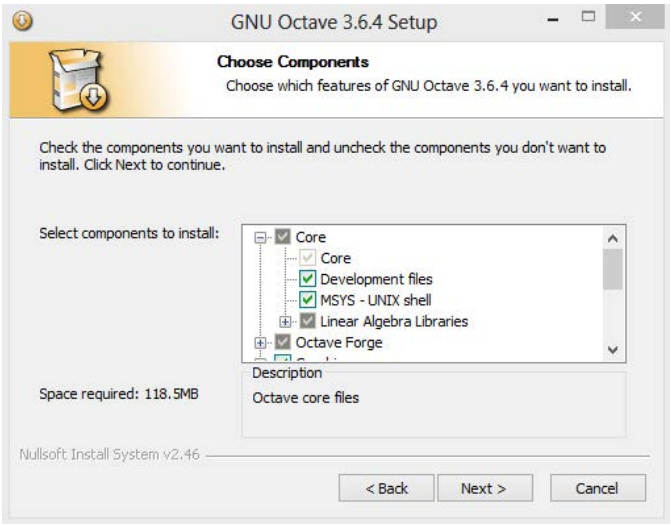


Uruchamiamy plik instalacyjny Octave. Następnie rozpoczynamy instalację klikając Next.

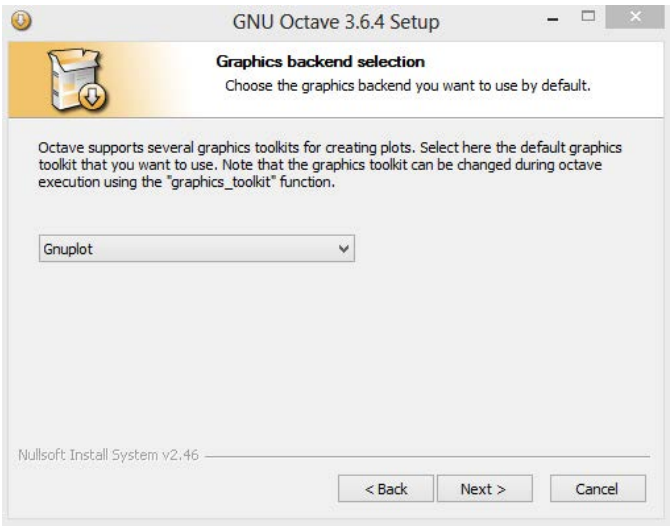
Wybór procesora pozostawiamy maszynie. Wystarczy kliknąć Next.



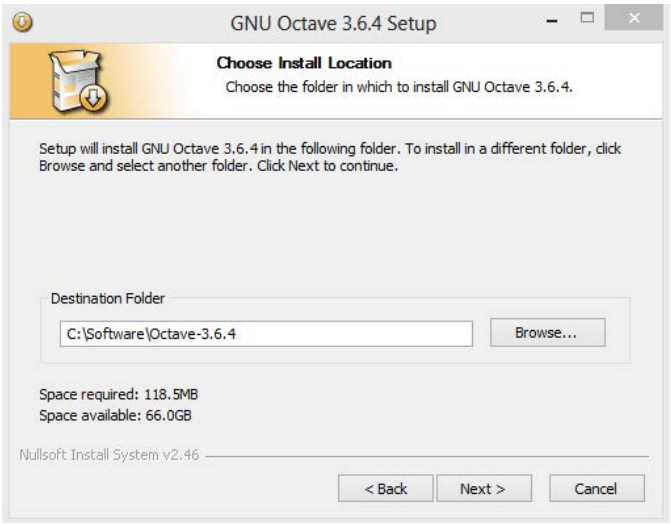
Wybór komponentów można pozostawić domyślny. Klikamy Next.



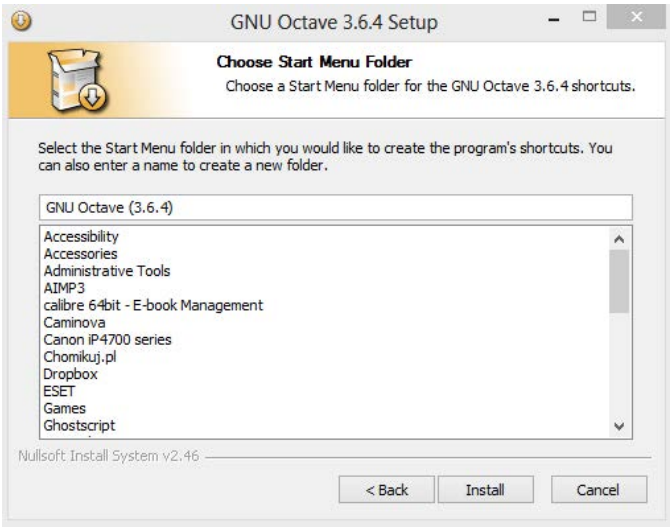
Wybór maszyny graficznej zostawiamy domyślny. Klikamy Next.



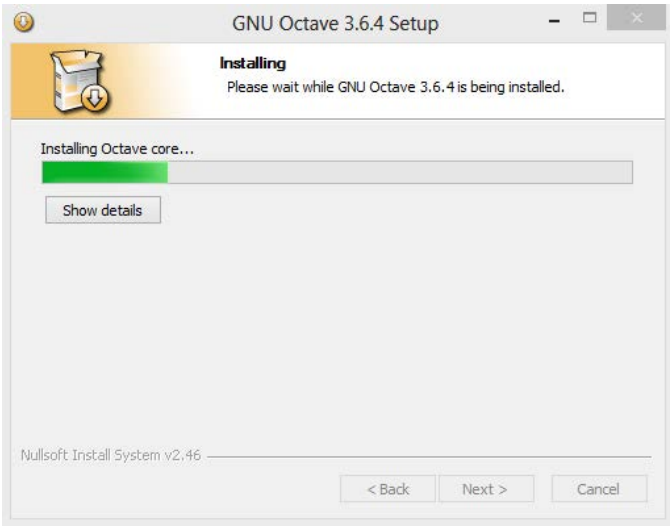
Wybieramy lokalizację plików Octave. Można zostawić domyślną. Klikamy Next.



Wybór lokalizacji folderu w menu Start zostawiamy domyślny. Klikamy Next.

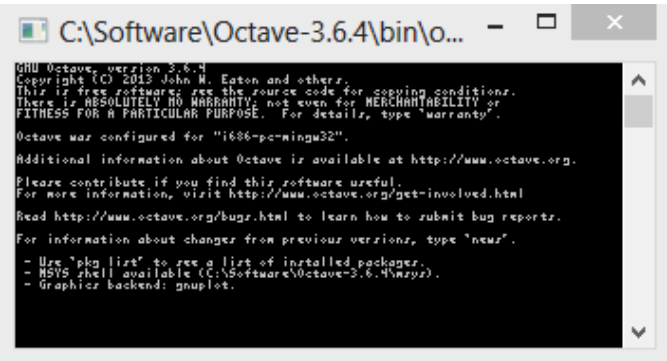


Czekamy na zakończenie kopiowania plików.

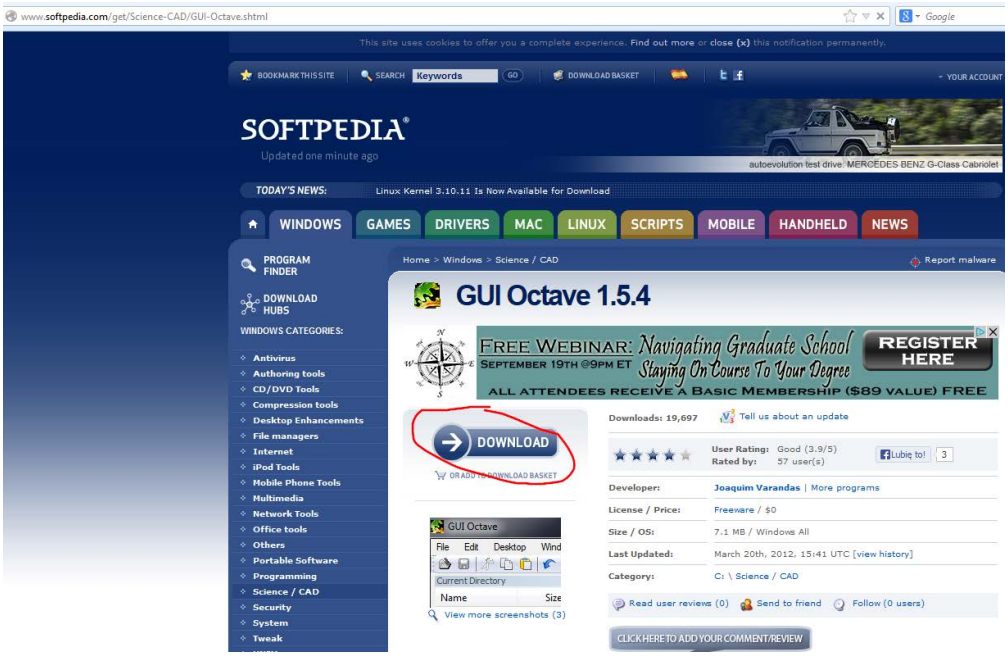


Octave został zainstalowany. Klikamy Finish.

Jeżeli otworzy się okno z trybem poleceń, zamykamy klikając w krzyżyk w prawym górnym rogu.



Przechodzimy do instalacji interfejsu graficznego Octave, tj. GUI Octave. Jeżeli nie mamy jeszcze pobranego programu instalacyjnego, to w przeglądarce wpisujemy lokalizację http://www.softpedia.com/get/Science-CAD/GUI-Octave.shtml lub znajdujemy ją, podając w wyszukiwarce hasła „GUI Octave” i „softpedia”. Wybieramy właściwy przycisk pobierania instalatora:

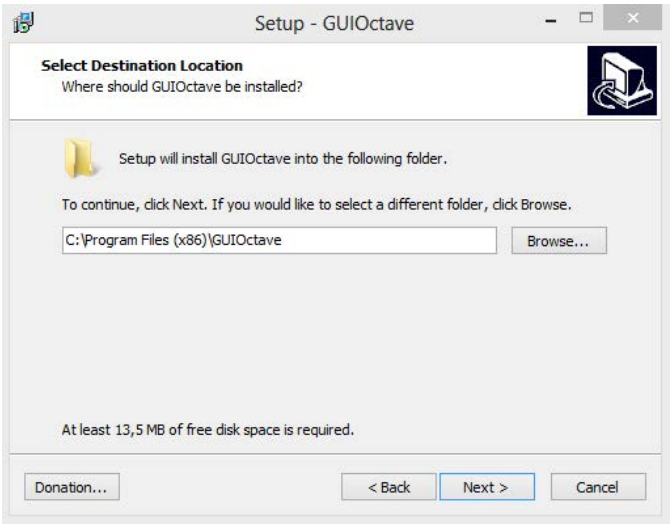


Wybieramy dowolny serwer plików na pobranie instalatora:

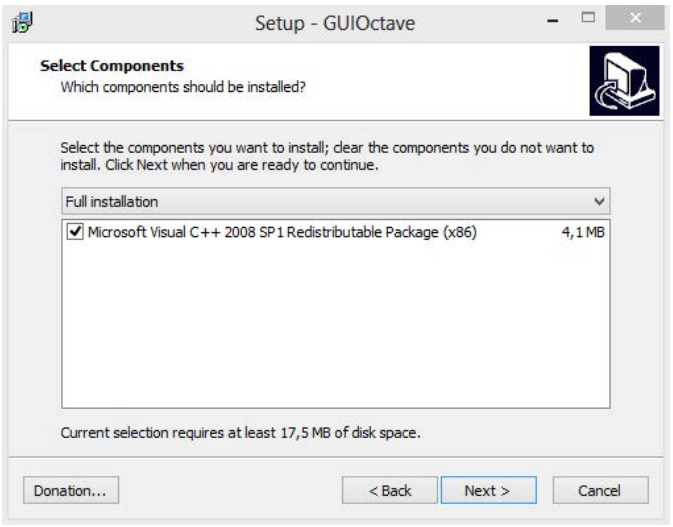
Uruchamiamy plik instalacyjny GUI Octave. Następnie rozpoczynamy instalację klikając Next.

Akceptujemy umowę licencyjną. Klikamy Next.

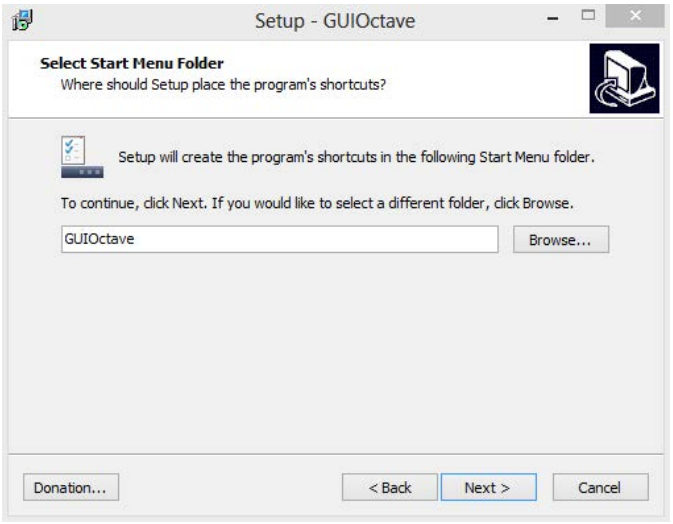
Potwierdzamy lokalizację dla plików GUI Octave. Klikamy Next.



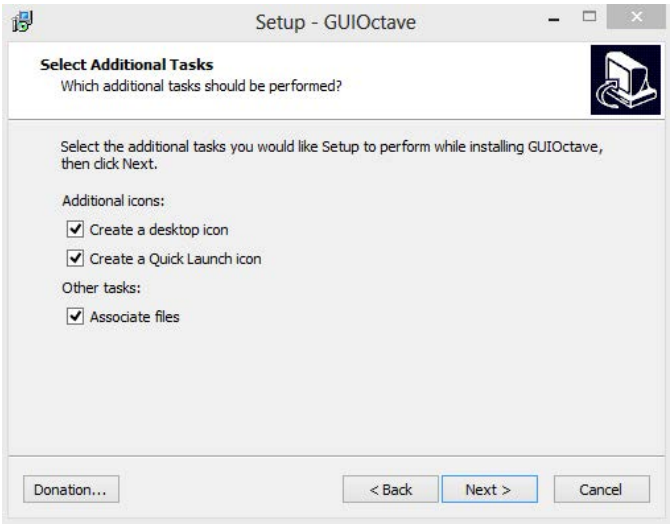
Potwierdzamy wybór kompilatora C++ do instalacji. Jeżeli wiemy, że kompilator jest już zainstalowany w systemie, możemy odznaczyć opcję instalacji. Klikamy Next.



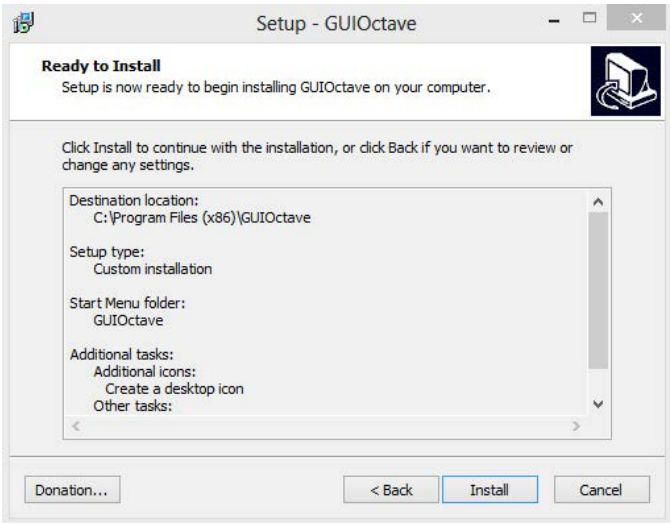
Wybór lokalizacji folderu w menu Start zostawiamy domyślny. Klikamy Next.



Dodatkowe czynności. Można dowolnie wybrać, zaakceptować i kliknąć Next.

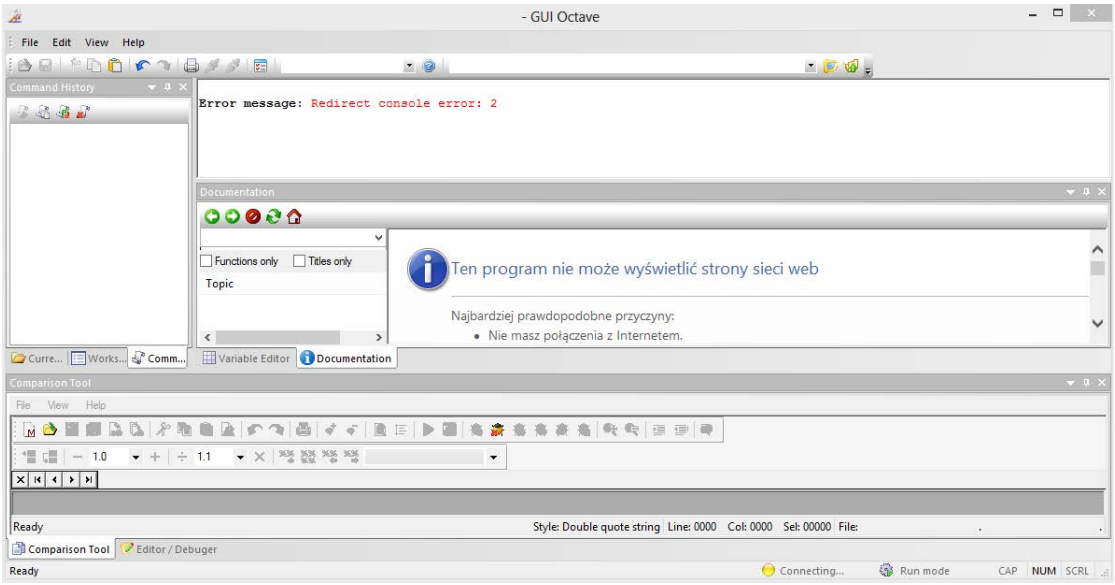


Potwierdzamy gotowość do instalacji. klikamy Install.

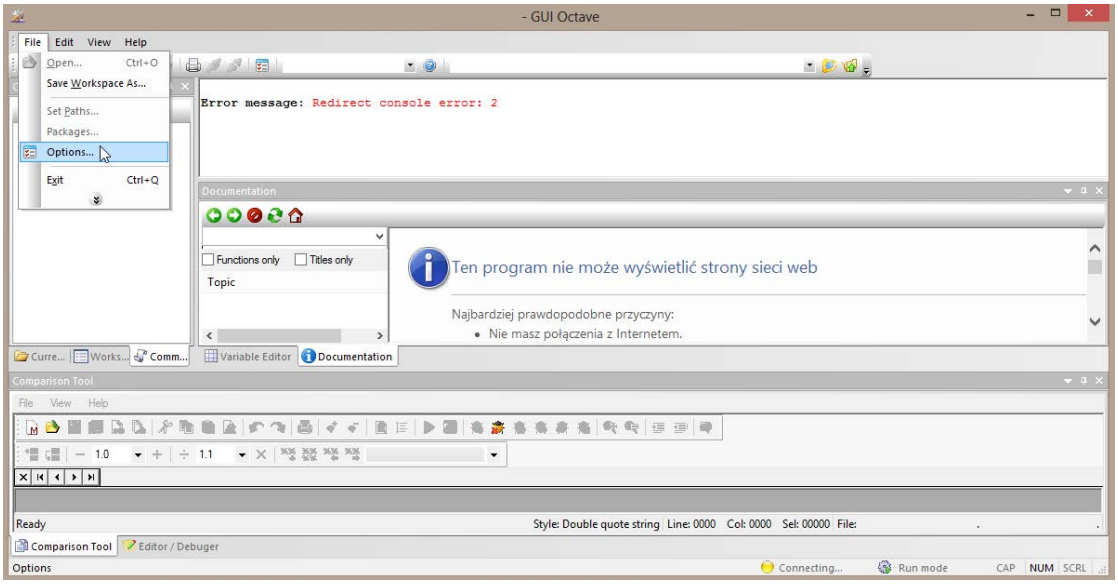


Octave został zainstalowany. Klikamy Finish.

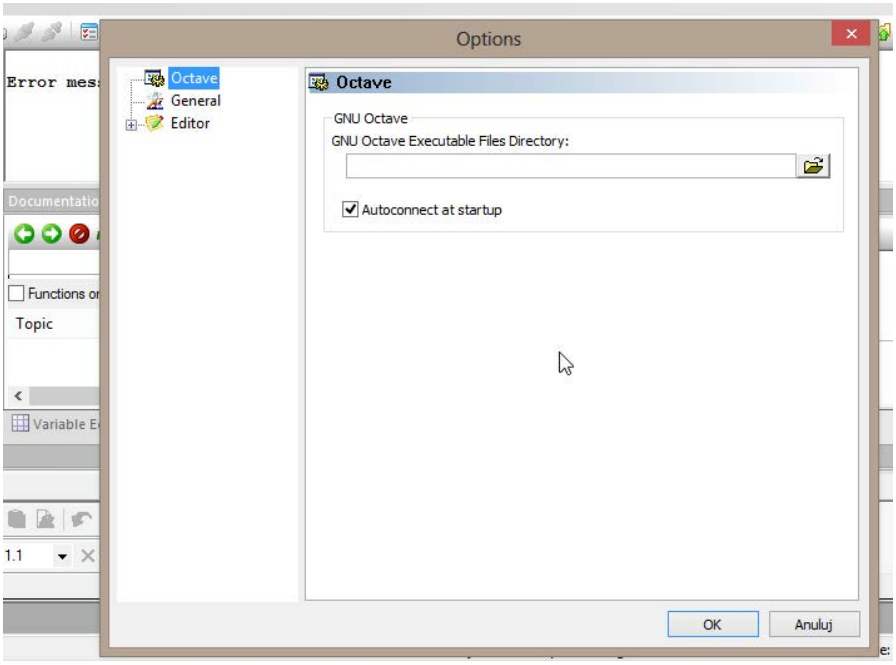
Po uruchomieniu GUI Octave widzimy komunikat o błędzie. GUI Octave nie zna lokalizacji pliku uruchomieniowego Octave. Teraz zajmiemy się podłączeniem GUI Octave do pliku uruchomieniowego Octave.



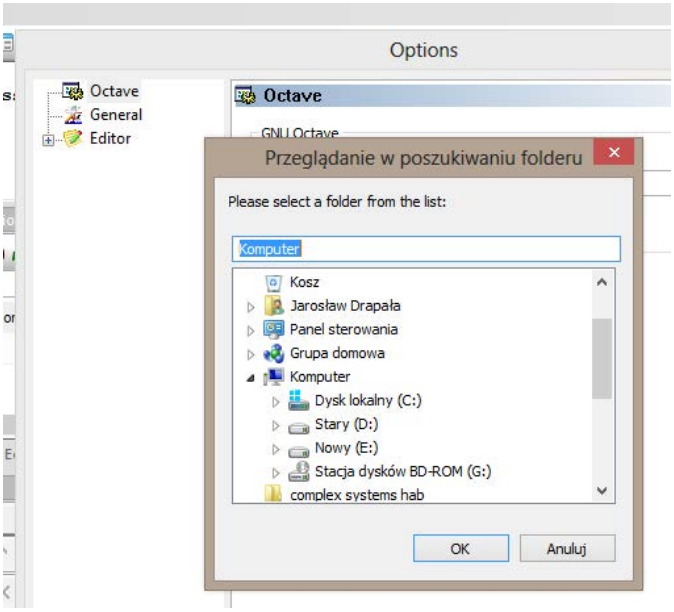
Wybieramy File − > Options ...



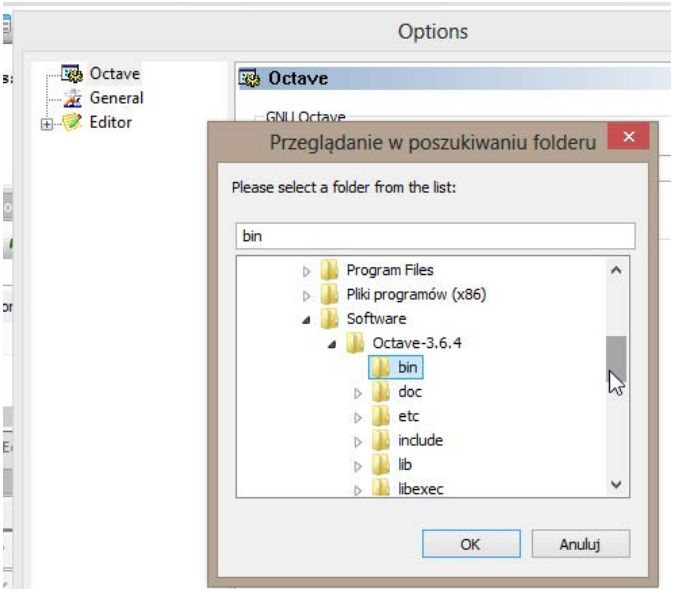
W oknie Options klikamy w ikonę wyszukiwania katalogu (GNU Octave Executable Files Directory:).



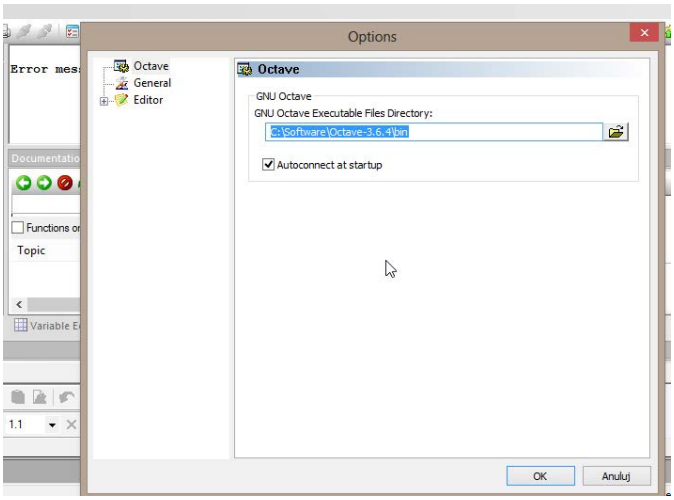
Znajdujemy folder Octave. Jeżeli zapomnieliśmy, gdzie to było, najprawdopodobniej znajduje się on na dysku C w katalogu Software.



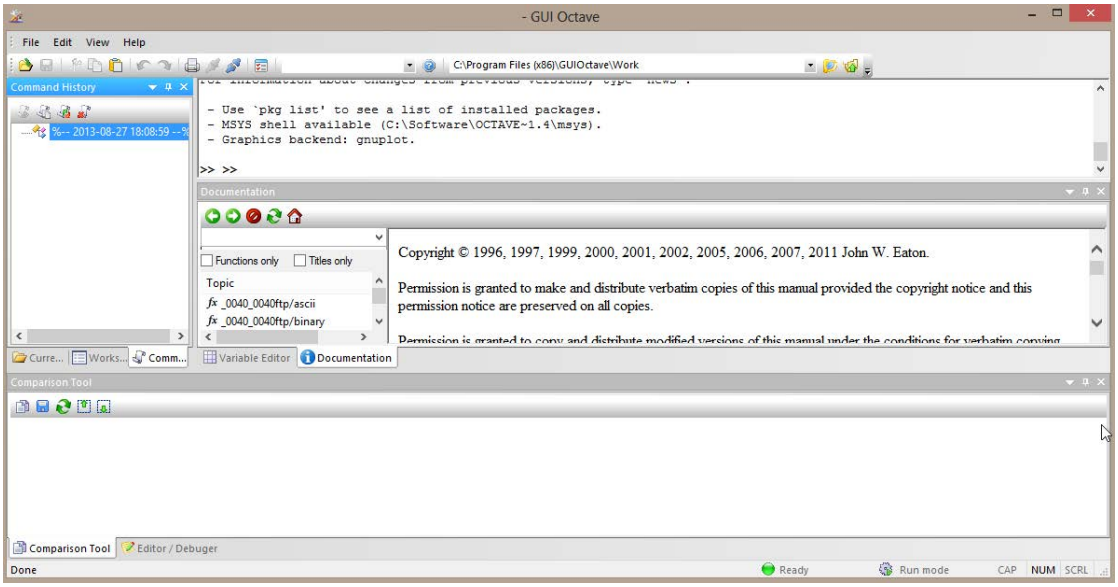
Z folderu Octave wybieramy katalog bin. Potwierdzamy kliknięciem OK.



Ponownie potwierdzamy wybór katalogu kliknięciem OK.

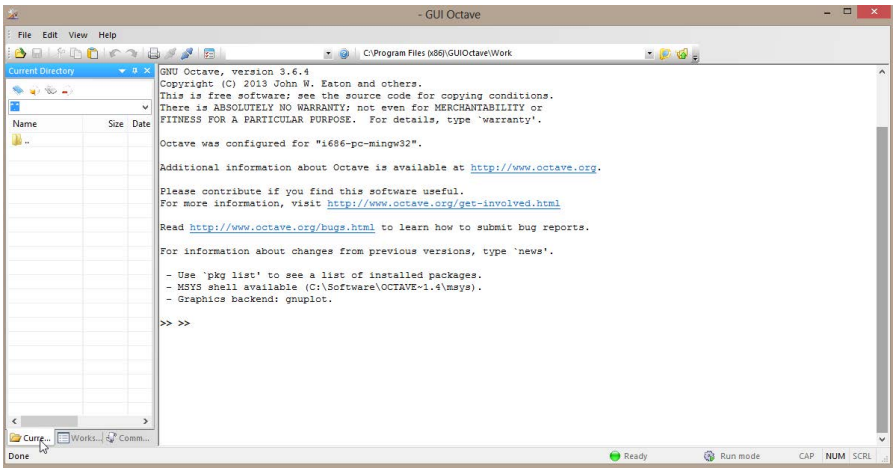


Interfejs zawiera okna, które nie będą nam potrzebne do pracy.

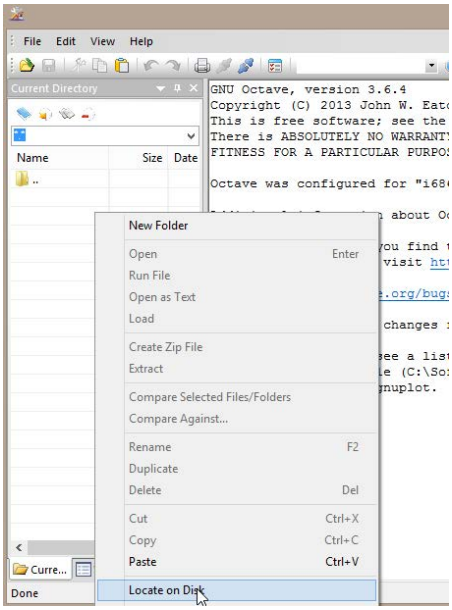


Zamykamy niepotrzebne okna, zostawiając tylko te na poniższym rysunku. Na koniec zamykamy GUI Octave.

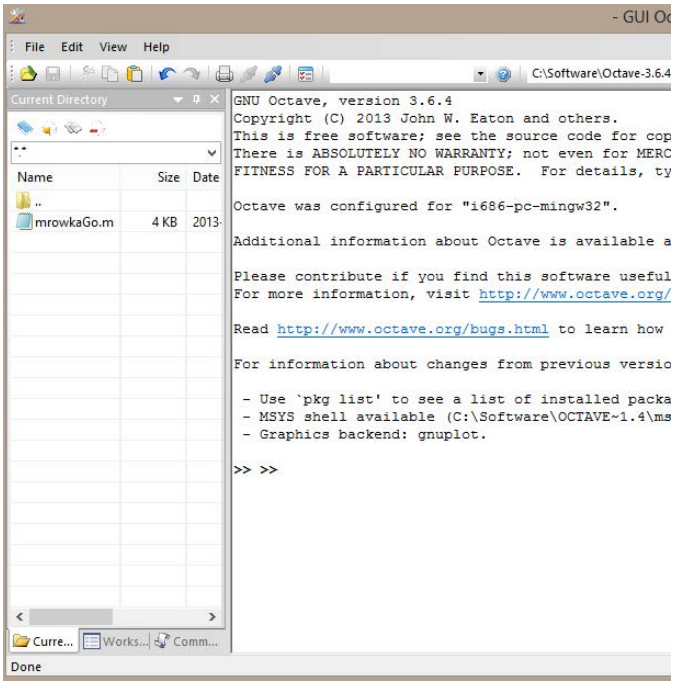
Uruchamiamy ponownie GUI Octave. Tym razem nie ma komunikatu o błędzie. Największe okno to okno z linią komend Octave. W oknie Octave „Current Directory" widzimy zawartość bieżącego katalogu.



Najłatwiej można się dostać do bieżącego katalogu, klikając w oknie Current Directory prawym przyciskiem myszy i wybierając „Locate on Disk".



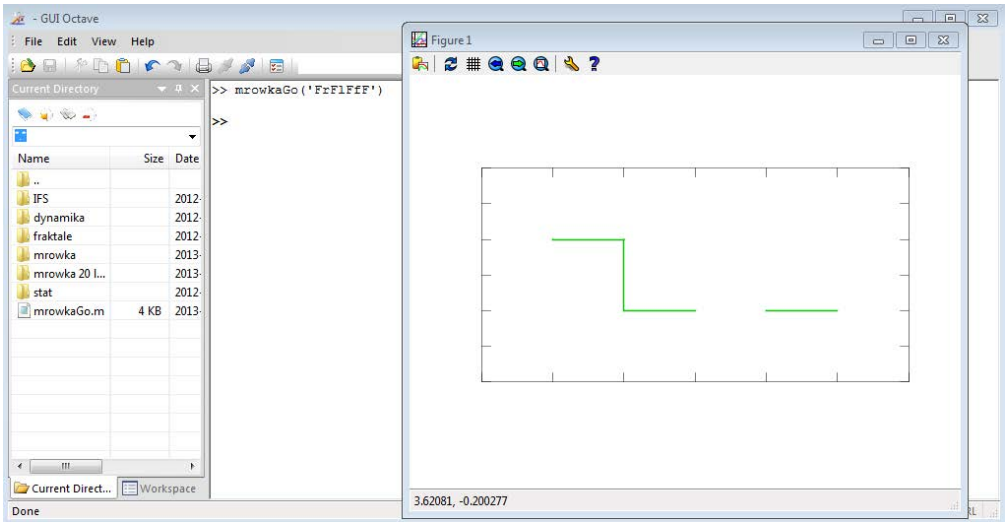
Otworzy się okno eksploratora. Można teraz plik z programem `mrowkaGo.m` skopiować do bieżącego katalogu i rozpocząć przygodę z mrówką.



Wpiszmy w linii komend jakieś polecenie, na przykład:

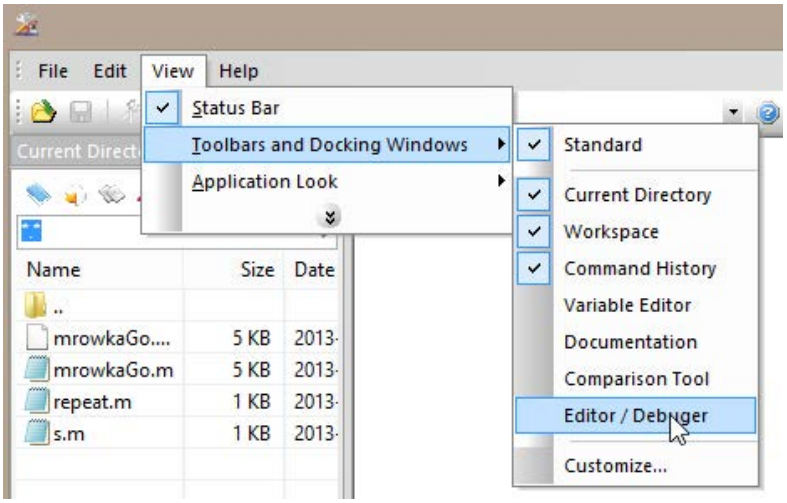
```
>> mrowkaGo('FrFlFfF')
```

Po wciśnięciu Enter zobaczymy obrazek przedstawiający ślad mrówki.

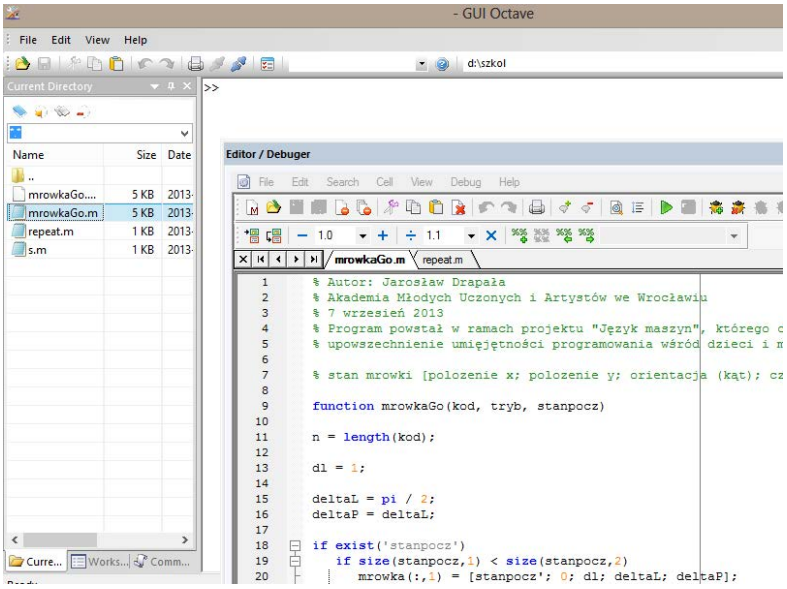


Dodatek C – Octave - edytor tekstu

Aby skorzystać z wbudowanego edytora plików tekstowych, musi być aktywny widok okna edytora. Jeżeli nie jest, uaktywniamy go wybierając z menu View „Toolbars and Docking Windows” − > „Editor/Debugger”:



Okno edytora najlepiej „odczepić” od okna Octave, dwukrotnie klikając w belkę edytora lewym przyciskiem myszy:



Dodatek D – `mrowkaGo` - jak zobaczyć mrówkę

Jeżeli z jakichś względów chcemy zobaczyć, gdzie i w jakim stanie znajduje się mrówka po wykonaniu rozkazu, możemy skorzystać z trybu `'on'` mrówki. Poniższe dwa obrazki ilustrują sposób użycia tego trybu.