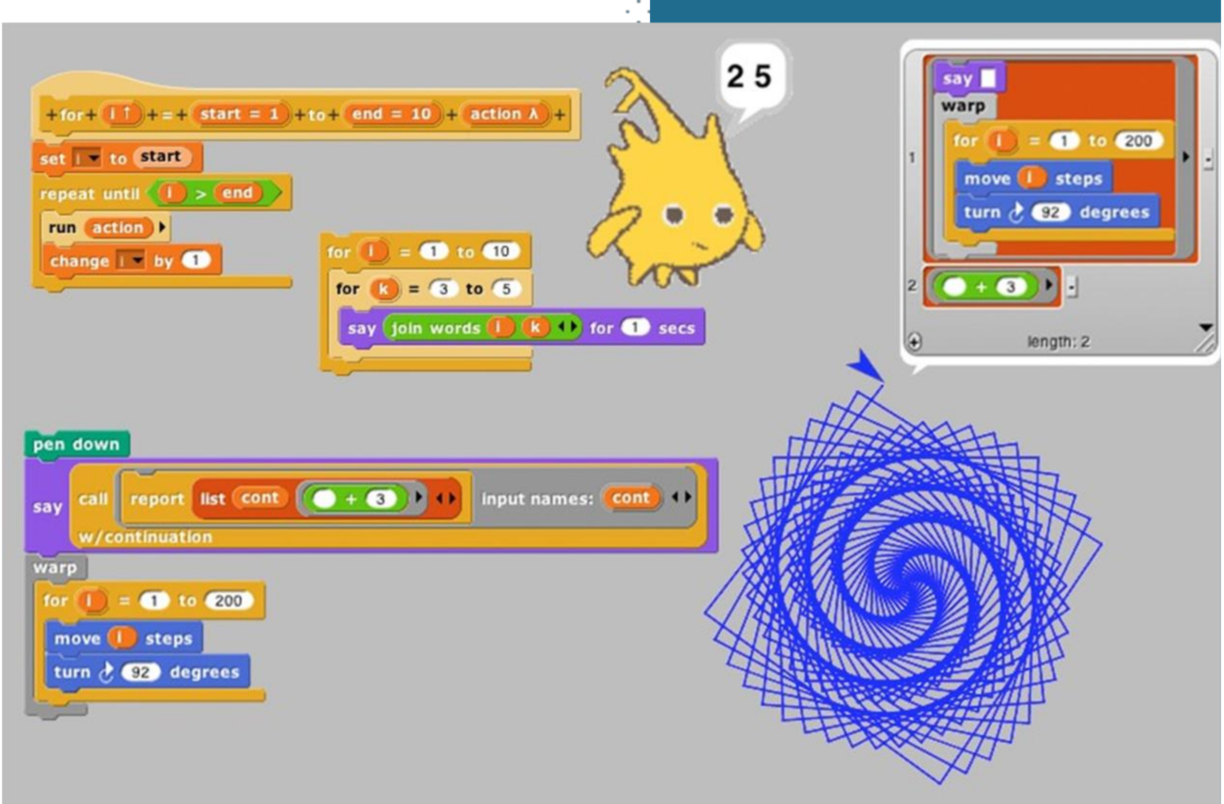
Build Your Own Blocks

Buduj własne bloki

4.1

# SNAP*!* Podręcznik użytkownika





Brian Harvey

Jens Mönig


*Tłumaczenie WtK*

# Spis treści


Podziękowania ... 3
Od tłumacza ... 3
I Bloki, skrypty i duszki ... 4
Bloki kapeluszowe i bloki poleceń (komend) ... 5
B Duszki i procesy równoległe ... 7
Kostiumy i dźwięki ... 7
Komunikacja między duszkami przez nadawanie komunikatów 8
C Zagnieżdżenie duszków: kotwice i części ... 9
D Bloki operacji i wyrażenia ... 9
E Predykaty i obliczenia warunkowe ... 11
F Zmienne ... 12
Zmienne globalne ... 13
Zmienne skryptu ... 14
Zmiana nazw zmiennych ... 14
Zmienne chwilowe ... 15
G Debugowanie – poprawianie błędów ... 16
Przycisk pauzy ... 16
Pułapki: blok pauzuj wszystko ... 16
Wizualizacja pracy krokowej ... 17
H Etcetera ... 18
Biblioteka narzędzi (bloki utworzone w Snap!): ... 19
II Zapisywanie i ladowanie projektów i mediów ... 20
A Zapis lokalny ... 20
Magazyn przeglądarki ... 20
Eksport XML ... 21
B Zapis w chmurze ... 21
C Ładowanie zapisanych projektów ... 22
III Budowanie nowego bloku ... 23
A Proste bloki ... 23
Nowe bloki z polem wejścia (parametrem) ... 25
B Rekurencja ... 26
C Biblioteki bloków ... 27
IV Listy pierwszej klasy ... 28
A Blok lista ... 28
B Listy list ... 29
D Operacje wyższego rzędu na listach i obwiednie ... 31
E Widok tabeli a widok listy ... 33
CSV wartości oddzielone przecinkami ... 36
V Typy pól wejść (parametrów) ... 37
A Notacja typów w Scratchu ... 37
B Okno dialogowe nazwy parametru w Snap*!* ... 37
Typy procedur ... 38
Rozwijane pola wejścia ... 39
Warianty wejścia ... 41
Oznaczenia w prototypach bloków ... 41
Tekst tytułowy i symbole ... 42
VI Procedury jako dane ... 43
A Wywołaj i uruchom ... 43
Wywołaj/uruchom z polami wejścia ... 43
Zmienne w obwiedniach ... 44
B Pisanie procedur wyższego rzędu ... 44
Rekurencyjne wywołania bloków z wieloma wejściami ... 46
C Parametry formalne ... 47
D Procedury jako dane ... 48
E Formy specjalne ... 49
Specjalne formy w Scratchu ... 50
VII Programowanie obiektowe i duszki ... 51
A Duszki pierwszej kategorii ... 51
B Klony stałe i tymczasowe ... 52
C Wysyłanie komunikatów do duszków ... 52
Polimorfizm ... 53
D Stan lokalny duszka: zmienne i atrybuty ... 54
E Prototypowanie: rodzice i dzieci ... 54
F Dziedziczenie przez delegowanie ... 55
G Lista atrybutów ... 56
VIII Programowanie obiektowe i procedury ... 57
A Stan lokalny ze zmiennymi skryptu ... 57
B Wiadomości i procedury wysyłania ... 58
C Dziedziczenie przez delegowanie ... 59
D Implementacja prototypowania OOP ... 60
IX Świat zewnętrzny ... 63
A World Wide Web ... 63
B Urządzenia sprzętowe ... 64
C Data i czas ... 64
X Kontynuacje ... 65
A Styl przekazywania kontynuacji ... 66
B Wywołaj/Uruchom z kontynuacją ... 69
Wyjście nielokalne ... 71
Tworzenie systemu wątków ... 72
XI Elementy interfejsu użytkownika ... 73
A Elementy paska narzędzi ... 73
Menu Logo Snap*!* ... 73
Menu plik ... 74
Menu Chmura ... 83
Menu ustawień ... 84
Kontrola pracy krokowej ... 86
Przyciski zmiany rozmiarów sceny ... 86
Przyciski kontroli projektu ... 87
B Obszar palety bloków ... 87
Przyciski w palecie ... 87
Menu kontekstowe palety bloków ... 88
Menu kontekstowe tła palety ... 88
C Obszar skryptów ... 89
Wygląd duszka i kontrolki zachowania ... 89
Zakładki pola skryptów ... 89
Skrypty i bloki w skryptach ... 89
Menu kontekstowe tła w obszarze skryptów ... 91
Kontrolki w zakładce Kostiumy ... 92
Edytor obrazów ... 95
Kontrolki w zakładce Dźwięki ... 96
D Edytowanie z klawiatury ... 96
Rozpoczynanie i kończenie edytowania z klawiatury ... 96
Nawigacja podczas edytowania z klawiatury ... 97
Edytowanie skryptu ... 97
Uruchamianie wybranego skryptu ... 98
E Kontrolki sceny ... 98
Zagroda duszków i przyciski tworzenia duszka ... 100
F Załadowanie projektu gdy zaczynamy Snap! ... 100
G Strony lustrzane ... 101

## Podziękowania

Mieliśmy ogromne szczęście do naszych mentorów. Jens ostrzył zęby w towarzystwie pionierów Smalltalka: Alana Kay, Dana Ingallsa i całej reszty bandy, która wymyśliła komputery osobiste i programowanie obiektowe w czasach Xerox PARC. Pracował z Johnem Maloneyem z Zespołu Scratch MIT, który opracował środowisko graficzne Morphic, będące wciąż sercem Snap*!*.

*Wspaniały projekt Scratch, z Lifelong Kindergarten Group w MIT Media Lab, ma kluczowe znaczenie dla Snap!. Nasza wcześniejsza wersja, BYOB, była bezpośrednią modyfikacją kodu źródłowego Scratcha. Snap! został napisany od nowa, ale struktura jego kodu i interfejs użytkownika są w pełni zapożyczone ze Scratcha.*
*W dodatku zespół Scratcha, który mógł postrzegać nas jako rywali, był dla nas w pełni wspierający i przyjazny.*

Brian dorastał w MIT i Stanford Artificial Intelligence Labs, ucząc się od twórcy Lispa Johna McCarthy'ego, twórców Scheme Geralda J. Sussmana i Guy'a Steele'a oraz autorów najlepszej na świecie książki informatycznej Structure and Interpretation of Computer Programs[1], Hal'a Abelson'a i Gerald'a J. Sussman'a z Julie Sussman i wielu innych herosów informatyki.

*W czasach chwały Logo Lab w MIT mówiliśmy: „Logo to Lisp przebrany za BASIC". Teraz, dzięki procedurom pierwszej klasy, zakresowi leksykalnemu i kontynuacjom pierwszej klasy - Snap! to Scheme przebrany za Scratcha.*

Mieliśmy szczęście poznać niesamowitą grupę świetnych gimnazjalistów (!) i licealistów poprzez forum Scratch Advanced Topics. Wielu z nich dodało kod do Snap*!*: Kartik Chandra, Nathan Dinsmore, Connor Hudson, Ian Reynolds i Dylan Servilla. Znacznie więcej osób zgłosiło pomysły i raportowało błędy w trakcie testów alfa. Studenci UC Berkeley, którzy wnieśli swój wkład do kodu, to Michael Ball, Achal Dave. Kyle Hotchkiss, Ivan Motyashov i Yuan Yuan. Współtwórcy tłumaczeń są zbyt liczni, aby ich tutaj wymienić, ale są w polu "O Snap!..." w programie Snap*!*.

Materiał ten jest oparty na pracach wspieranych w części przez National Science Foundation w ramach Grantów Nr 1138596, 1143566 i 1441075; a częściowo przez MioSoft, Arduino.org, SAP i YC Research. Wszelkie opinie, wnioski i konkluzje lub zalecenia wyrażone w niniejszym materiale są opiniami autora (autorów) i niekoniecznie odzwierciedlają poglądy National Science Foundation lub innych fundatorów.

## Od tłumacza

Tłumaczenie podręcznika powstawało pod koniec roku 2017, bodźcem było prośba Andrzeja Batorskiego z serwisu `https://www.edukator.pl` o pomoc przy poprawianiu spolszczenia programu Snap*!*. Moje pierwotne tłumaczenie jeszcze wersji BYOB z roku 2011 rzeczywiście tego wymagało. Tak więc wróciłem do Snap*!* i wielokrotnie byłem zadziwiony jego możliwościami i tym jak znacznie odbiega on od swojego pierwowzoru – Scratcha. To już nie jest zabawka dla dzieci, to rzeczywiście wizualny język programowania o niskiej podłodze i bardzo wysokim suficie.
Brian Harvey używa go w Berkeley do zaprezentowania programowania studentom w swoim kursie „Beauty and Joy of Computing" (`http://bjc.berkeley.edu`). Tłumaczenie powstało z myślą o naszych studentach, nauczycielach informatyki, którzy przychodzą szkolić się w Ośrodku Edukacji Informatycznej i Zastosowań Komputerów ( `http://www.oeiizk.waw.pl`) i mam nadzieję, że będzie przez nich wykorzystywane.

Podręcznik nie jest łatwy i nie jest tylko instrukcją dla użytkownika. Wśród poruszanych zagadnień są: procesy równoległe, rekurencja, procedury jako dane, programowanie obiektowe, kontynuacje. Wielokrotnie omawia się rozbudowane przykłady różnych konstrukcji, jak choćby implementacje prototypowania OOP czy systemu wątków. Jednym z przykładów siły Snap*!* jest napisana w nim i przedstawiona w przykładzie *Codification* aplikacja umożliwiająca (ograniczone, ale jednak) tłumaczenie skryptów Snap*!* na kod kilku języków programowania (w tym Python i JavaScript).

Nie bez zawstydzenia przyznaję, że tłumaczenie zostało wykonane z pomocą `https://translate.google.pl`. Jednak mimo wielkiego postępu w automatycznym tłumaczeniu, „Beauty and Joy of Computing" dało w wyniku „Piękno i radość z komputerów".

Piątek, 29 grudnia 2017 *Witold Kranas*

[1] Polskie tłumaczenie w serii Klasyka informatyki: Abelson H., Sussman G. J., Sussman J.: Struktura i interpretacja programów komputerowych. (przełożył: Kubica M.), WNT, Warszawa, 2002 (ISBN 83-204-2712-6).[*WtK*]

# Snap! Podręcznik użytkownika

**Wersja 4.1**

Snap! (poprzednio BYOB) jest rozszerzoną reimplementacją Scratcha (**http://scratch.mit.edu**), która pozwala użytkownikowi budować własne bloki (Build Your Own Blocks). Ma ona również listy, procedury, duszki, kostiumy, dźwięki i kontynuacje pierwszej klasy. Te dodatkowe możliwości czynią ją odpowiednim narzędziem, by poważnie uczyć licealistów i studentów wprowadzenia do informatyki.

W podręczniku czasami odwołujemy się do Scratcha, np., aby wyjaśnić, jak dana funkcja w Snap! rozszerza coś znanego ze Scratcha. Bardzo pomocne, ale nie niezbędne jest posiadanie doświadczenia ze Scratchem przed przeczytaniem tego podręcznika.

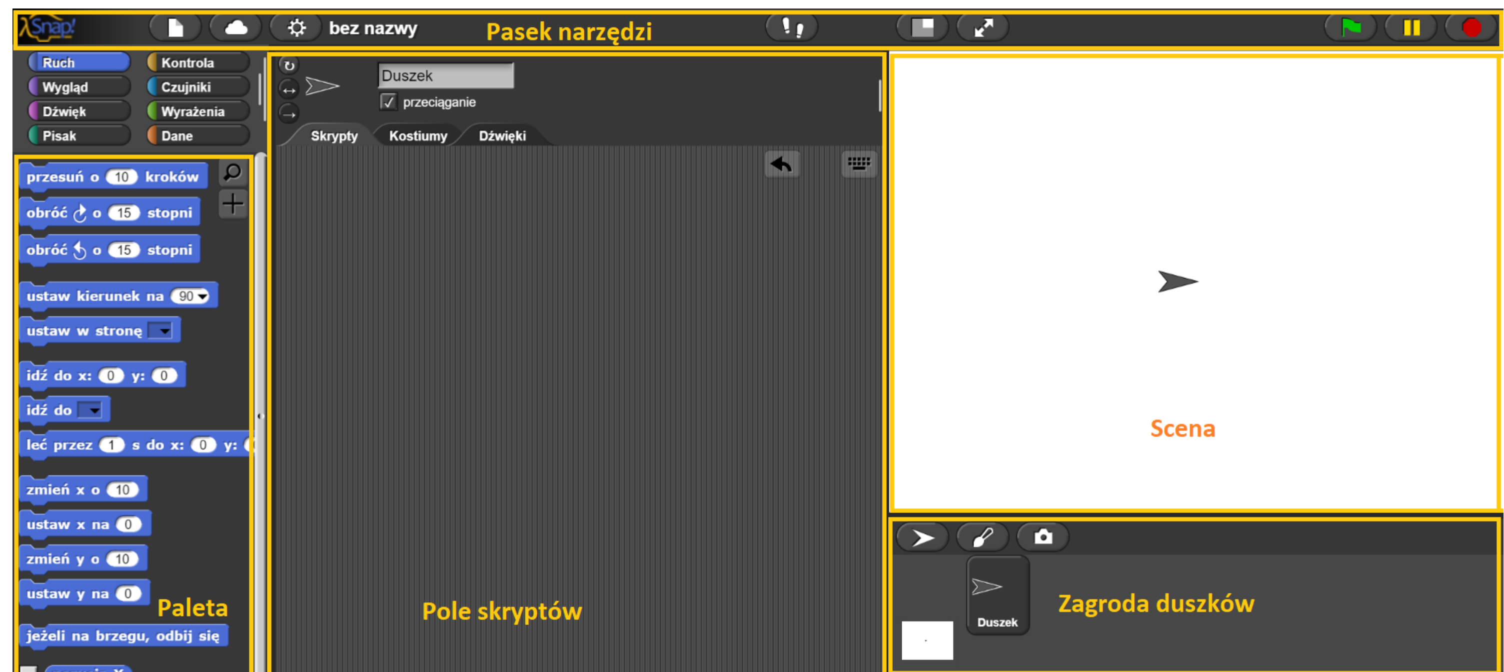
Żeby uruchomić Snap!, otwórz okno przeglądarki i połącz się z **http://snap.berkeley.edu/run**, aby rozpocząć od minimalnego zestawu bloków lub **http://snap.berkeley.edu/init**[2], aby załadować mały zestaw dodatkowych bloków (trochę wolniejsze uruchamianie, ale zalecane dla wygody i wykorzystywane w tym podręczniku).

## I Bloki, skrypty i duszki

Ta część opisuje odziedziczone po Scratchu funkcje Snap!; doświadczeni użytkownicy Scratcha mogą przejść do części B.

Snap! to język programowania – sposób zapisu, dzięki któremu możesz powiedzieć komputerowi, co chcesz zrobić. W przeciwieństwie do większości języków programowania Snap! jest językiem ***wizualnym***, zamiast pisać program za pomocą klawiatury, programista używa w Snap! techniki „przeciągnij i upuść”, dobrze znanej każdemu użytkownikowi komputera.

Uruchom Snap!. Zobaczysz okno z następującymi elementami:

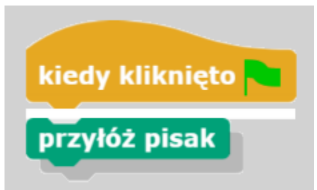


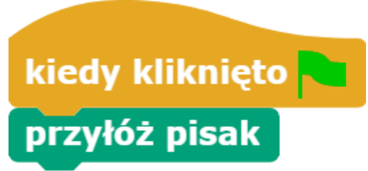

(Proporcje poszczególnych pól mogą być nieco różne w zależności od rozmiaru i kształtu okna przeglądarki.)

[2] Dodatkowe bloki będą w wersji angielskiej. Żeby mieć polskie wersje, trzeba pobrać bibliotekę z: https://www.dropbox.com/s/dxqukr5engll2c/BiblioNarzPLv5.xml?dl=0, zapisać ją lokalnie, a następnie otworzyć (Importuj…). [przypis WtK]

Program w Snap*!* składa się z jednego lub więcej skryptów, z których każdy składa się z bloków. Oto typowy skrypt:

Pięć bloków tworzących ten skrypt ma trzy różne kolory, odpowiadające trzem z ośmiu palet, w których można znaleźć bloki. Obszar palety przy lewej krawędzi okna pokazuje jedną paletę na raz, wybieraną za pomocą ośmiu przycisków tuż nad obszarem palety. W tym skrypcie złote bloki pochodzą z palety Kontrola; zielony blok pochodzi z palety Pisak; a niebieskie bloki pochodzą z palety Ruch. Skrypt jest układany poprzez przeciągnięcie bloków z palety do pola skryptów w środkowej części okna. Bloki łączą się ze sobą - zatrzaskują się (stąd nazwa Snap*!*) po przeciągnięciu bloku, tak aby jego wcięcie było w pobliżu zakładki nad nim:




Biała pozioma linia jest sygnałem, że jeśli puścisz zielony blok, zatrzaśnie się w zakładce złotego.

## Bloki kapeluszowe i bloki poleceń (komend)

U góry skryptu znajduje się blok kapeluszowy, który wskazuje, kiedy skrypt powinien zostać wykonany. Nazwy bloków kapeluszy zazwyczaj zaczynają się od słowa „**kiedy**"; w przykładzie rysującym kwadrat na górze strony, skrypt powinien być uruchamiany, gdy kliknięto zieloną flagę znajdującą się w prawym rogu paska narzędzi Snap*!*. (Pasek narzędzi Snap*!* jest częścią okna Snap*!*, nie jest tym samym, co pasek menu przeglądarki lub systemu operacyjnego). Skrypt nie musi mieć bloku kapeluszowego, ale jeśli go nie ma, to będzie uruchamiany tylko wtedy gdy użytkownik kliknie na sam skrypt. Skrypt nie może mieć więcej bloków kapeluszowych niż jeden, a blok ten może być użyty tylko u góry skryptu; jego charakterystyczny kształt ma to przypominać.

Jeden z bloków kapeluszowych, ogólny blok „kiedy cokolwiek", jest nieco inny od pozostałych. Po kliknięciu znaku stopu ten blok nie sprawdza już, czy warunek umieszczony w jego sześciokątnym polu wejściowym jest prawdziwy, więc skrypt pod nim nie będzie działał, dopóki nie zostanie uruchomiony inny skrypt w projekcie (ponieważ na przykład klikniesz zieloną flagę).

Pozostałe bloki w naszym przykładowym skrypcie to bloki poleceń. Każdy blok poleceń odpowiada akcji, którą Snap*!* rozpoznaje i wie jak wykonać. Na przykład blok mówi duszkowi (o kształcie grota strzałki, na scenie po prawej stronie okna), aby przesunąć się o dziesięć kroków (krok to bardzo mała jednostka odległości) w kierunku, w którym wskazuje grot strzałki. Niedługo zobaczymy, że może być więcej niż jeden duszek i że każdy duszek ma swoje własne skrypty. Poza tym duszek nie musi wyglądać jak grot strzałki, ale może mieć dowolny obrazek jako kostium. Kształt bloku **przesuń** ma przypominać klocek Lego ™; skrypt to stos bloków. (Słowo „blok" oznacza zarówno kształt graficzny na ekranie, jak i procedurę, działanie, które blok wykonuje).

Liczba 10 w bloku **przesuń** powyżej nazywana jest *wejściem*[3] (*input*) do bloku. Klikając na biały owal, możesz wpisać dowolną liczbę zamiast 10. Przykładowy skrypt na górze strony używa 100 jako wartości wejściowej. Zobaczymy później, że pola wejściowe mogą mieć nieowalne kształty, które akceptują wartości inne niż liczby. Zobaczymy także, że można obliczać wartości wejściowe, zamiast wpisywać określoną wartość do owalu. Blok może mieć więcej niż jedno pole wejściowe. Na przykład blok **leć,** znajdujący się mniej więcej w połowie palety Ruch, ma trzy pola wejścia.

[3] Wolałbym tu słowo parametr, ale chcę zachować opisową narrację w tej części podręcznika [*przypis WtK*].

Bloki poleceń mają głównie kształt cegiełki, ale niektóre, podobnie jak blok **powtórz** w przykładowym skrypcie, mają kształt litery C. Większość bloków C-kształtnych znajduje się w palecie bloków Kontrola. Puste miejsce w środku litery C jest specjalnym rodzajem pola wejściowego, które akceptuje *skrypt* jako dane wejściowe. W przykładowym skrypcie blok **powtórz** ma dwa wejścia: liczbę 4 i skrypt:

przesuń o 100 kroków
obróć ↻ o 90 stopni

Bloki w kształcie litery C można umieścić w skrypcie na dwa sposoby. Jeśli zobaczysz białą linię i puścisz, blok zostanie wstawiony do skryptu jak każdy blok poleceń:

Ale jeśli zobaczysz pomarańczową obwiednię i puścisz blok, to otoczy on obwiedzione bloki:

Obwiednia będzie zawsze rozciągać się od pozycji kursora do dolnej części skryptu:

Jeśli chcesz umieścić wewnątrz tylko część bloków, po wstawieniu możesz chwycić pierwszy blok, którego nie chcesz mieć wewnątrz, pociągnąć w dół i przytwierdzić pod blokiem w kształcie litery „C".

W przypadku bloków w kształcie litery „E" z więcej niż jednym pustym miejscem, jakie było w bloku C-kształtnym, tylko pierwszy otwór będzie obejmował bloki istniejące w skrypcie i tylko wtedy, gdy będzie on pusty. (Możesz wypełnić pozostałe miejsca, przeciągając bloki do wybranego otworu).

## B Duszki i procesy równoległe

Tuż pod sceną znajduje się przycisk „nowy duszek”. Kliknij ten przycisk, aby dodać duszka. Nowy duszek pojawi się na scenie w losowej pozycji, zwrócony w przypadkowym kierunku, z losowym kolorem.

Każdy duszek ma swoje własne skrypty. Aby zobaczyć w obszarze skryptów skrypty dla danego duszka, kliknij na obrazek tego duszka w zagrodzie duszków w prawym dolnym rogu okna. Spróbuj wprowadzić jeden z następujących skryptów w obszarze skryptów każdego z duszków:

Kiedy klikniesz zieloną flagę, zobaczysz, jak jeden duszek obraca się, a drugi porusza się tam i z powrotem. Ten eksperyment pokazuje, w jaki sposób różne skrypty mogą działać równolegle. Obracanie i poruszanie się odbywają się razem. Równoległość można zaobserwować również w wielu skryptach jednego duszka. Spróbuj tego przykładu:

Kiedy naciśniesz klawisz spacji, duszek powinien poruszać się zawsze w kółko, ponieważ bloki **przesuń** i **obróć** działają równolegle. (Aby zatrzymać program, kliknij czerwony znak stopu na prawym końcu paska narzędzi).

### Kostiumy i dźwięki

Aby zmienić wygląd duszka, zaimportuj dla niego nowy *kostium*. Można to zrobić na trzy sposoby. Najpierw wybierz pożądanego duszka w zagrodzie. Następnie, jednym ze sposobów jest kliknięcie ikony pliku na pasku narzędzi i wybranie z menu pozycji „**Kostiumy ...**”. Zobaczysz listę kostiumów z biblioteki mediów publicznych i możesz wybrać jeden z nich. Drugi sposób, w przypadku kostiumu przechowywanego na własnym komputerze, polega na kliknięciu ikony pliku i wybraniu pozycji menu „**Importuj ...**”. Następnie można wybrać plik w dowolnym formacie obrazu (PNG, JPEG itp.) obsługiwanym przez przeglądarkę. Trzeci sposób jest szybszy. Jeśli żądany plik jest widoczny na pulpicie, po prostu przeciągnij plik na okno Snap*!*. W każdym z tych przypadków obszar skryptów zostanie zastąpiony przez coś takiego:

Powyżej tej części okna znajduje się zestaw trzech zakładek: Skrypty, Kostiumy i Dźwięki. Zobaczysz, że karta Kostiumy jest teraz zaznaczona. W tym widoku garderoby duszka, możesz wybrać, czy duszek powinien nosić swój kostium Żółwia czy też strój Alonzo. (Alonzo, maskotka Snap*!*, ma imię po Alonzo Church’u, matematyku, który

wymyślił koncepcję procedur jako danych, najważniejszy sposób, w jaki Snap*!* różni się od Scratcha). Możesz dodać duszkowi tyle kostiumów, ile chcesz, a następnie wybrać, który z nich będzie nosić, klikając w jego szafie lub używając bloku **zmień kostium na Żółw** lub **następny kostium** w skrypcie. (Każdy kostium ma numer i nazwę, a blok **następny kostium** wybiera kolejny kostium według numeru, po kostiumie o najwyższym numerze przestawia się na kostium 1. Żółw, kostium 0, nigdy nie jest wybierany przez blok **następny kostium**). Kostium Żółw jest jedynym, który zmienia kolor, aby go dopasować do koloru pisaka.

Oprócz kostiumów, duszek może mieć dźwięki; odpowiednik garderoby duszka można nazwać szafą grającą. Pliki dźwiękowe można importować w dowolnym formacie (WAV, OGG, MP3 itp.) obsługiwanym przez przeglądarkę. Zadanie odgrywania dźwięków realizują dwa bloki. Jeśli chcesz, aby skrypt działał dalej, podczas gdy dźwięk jest odtwarzany, użyj bloku **zagraj dźwięk hej**. Natomiast żeby czekać na zakończenie odtwarzania dźwięku przed kontynuowaniem reszty skryptu możesz użyć bloku **zagraj dźwięk hej i czekaj**.

## Komunikacja między duszkami przez nadawanie komunikatów

Wcześniej widzieliśmy przykład dwóch duszków poruszających się w tym samym czasie. W bardziej interesującym programie, duszki na scenie będą *współdziałać*, aby opowiedzieć historię, zagrać w grę itp. Często jeden duszek będzie musiał powiedzieć innemu duszkowi, aby uruchomił skrypt. Oto prosty przykład:

kiedy kliknięto
powiedz Hej! Jak masz na imię? przez 2 s
nadaj szczek do wszystkich i czekaj
powiedz Cześć Hauu. Co chcesz robić? przez 2 s
nadaj szczek do wszystkich i czekaj
powiedz Co za niespodzianka! przez 2 s

W bloku **nadaj szczek do wszystkich i czekaj** słowo „szczek" jest po prostu dowolną nazwą, którą wymyśliłem. Po kliknięciu strzałki w dół w tym polu wejściowym, jednym z wyborów (jedynym wyborem, za pierwszym razem) jest „nowy", który następnie prosi o podanie nowej nazwy komunikatu. Po uruchomieniu tego bloku wybrana wiadomość jest wysyłana do każdego duszka, dlatego blok nazywa się „nadaj". W przykładowym programie jednak tylko jeden duszek ma skrypt odbierający ten komunikat, jest to pies. Ponieważ skrypt chłopca wykorzystuje **blok nadaj … do wszystkich i czekaj**, a nie **blok nadaj … do wszystkich**, chłopiec nie przechodzi do następnego bloku, dopóki skrypt psa się nie skończy. Właśnie dlatego te dwa duszki mówią na zmianę, a nie jednocześnie.

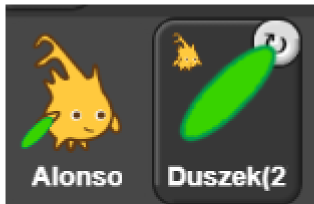
Zauważ, przy okazji, że pierwsze pole wejściowe bloku **powiedz** jest prostokątne, a nie owalne. Oznacza to, że dane wejściowe mogą być dowolnym ciągiem tekstowym, a nie tylko liczbą. W wejściach do wprowadzania tekstu znak spacji jest wyświetlany jako brązowa kropka, dzięki czemu można policzyć liczbę spacji między wyrazami, a w szczególności można odróżnić pole puste i zawierające spacje. Brązowe kropki nie są pokazywane na scenie podczas wykonywania bloku.

Scena ma własny obszar skryptów. Można go wybrać, klikając ikonę Sceny po lewej stronie zagrody dla duszków. Jednak w przeciwieństwie do duszka scena nie może się poruszyć. Zamiast kostiumów ma *tła*: obrazy, które wypełniają cały obszar sceny. Duszki pojawiają się na bieżącym tle. W skomplikowanym projekcie często wygodnie jest używać skryptu w obszarze sceny jako ogólnego reżysera akcji.

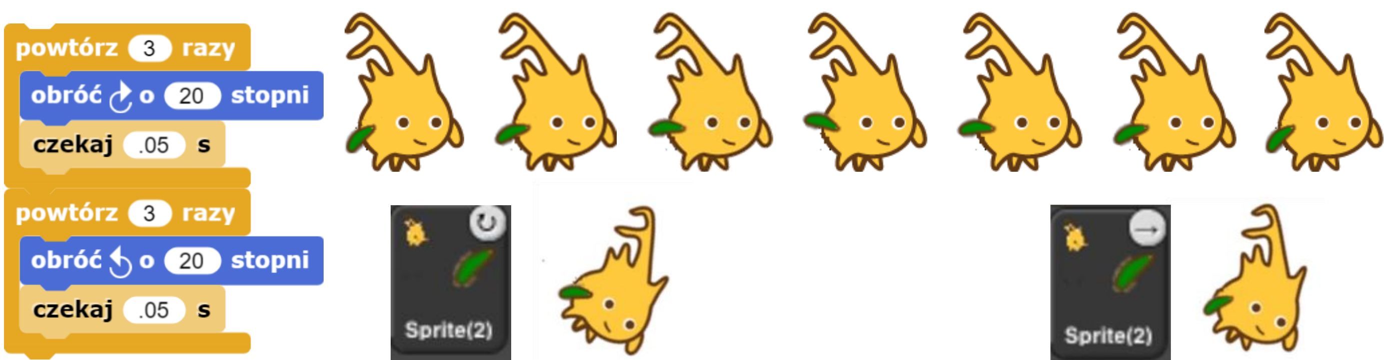
## C Zagnieżdżenie duszków: kotwice i części

Czasami potrzebne jest stworzenie czegoś w rodzaju „super-duszka" złożonego z elementów, które mogą poruszać się razem, ale można je również oddzielnie określać. Klasycznym przykładem jest ciało składające się z tułowia, kończyn i głowy. Snap*!* pozwala wyznaczyć jednego duszka jako *kotwicę* połączonego kształtu, z innymi duszkami jako jego *częściami*. Aby ustawić zagnieżdżanie duszków, przeciągnij ikonkę duszka-części z zagrody na wyświetlaną na scenie postać wybranego duszka-kotwicy (nie na ikonę duszka w zagrodzie!).

Zagnieżdżenie duszków widoczne jest w ikonkach duszka-kotwicy (Alonso) i duszka-części (Duszek(2)) znajdujących się w zagrodzie.




Na tej ilustracji chcemy animować ramię Alonzo. (Ramię ma kolor zielony na tym obrazku, aby powiązanie dwóch duszków był wyraźniejsze, ale w prawdziwym projekcie prawdopodobnie będą one tego samego koloru). Duszek reprezentujący ciało Alonzo, jest kotwicą; Duszek(2) to ramię. Ikona kotwicy pokazuje na dole małe obrazy co najwyżej trzech dołączonych części. Ikona każdej części pokazuje mały obraz kotwicy w lewym górnym rogu oraz *oznaczenie synchronizacji/podwieszenia obrotu* w prawym górnym rogu. W początkowym ustawieniu, jak pokazano powyżej, kółko ze strzałką oznacza, że gdy duszek-kotwica obraca się, duszek-część również obraca się, a także obraca się wokół kotwicy. Po kliknięciu kółka strzałka zmienia się z okrągłej w prostą i wskazuje, że gdy duszek-kotwica obraca się, duszek-część rotuje wokół niego, ale nie obraca się z nim, zachowując swoją oryginalną orientację. (Część można również obracać osobno, używając jej bloków **obróć**). Każda zmiana położenia lub rozmiaru kotwicy jest zawsze przenoszona na jej części.




*Góra: obracana część: zielona ręka. Dół: obracanie kotwicy, z ramieniem synchronicznym (po lewej) i zwisającym (po prawej).*

## D Bloki operacji i wyrażenia

Do tej pory używaliśmy dwóch rodzajów bloków: bloków kapeluszowych i bloków poleceń (komend, ang. *command*). Innym rodzajem jest *blok operacji (funkcji)*, który ma owalny kształt: pozycja X. Nazywa się go „operacją" (ang. *reporter*), ponieważ kiedy blok jest uruchamiany, zamiast wykonywać akcję, daje w wyniku wartość, która może być użyta jako dana wejściowa do innego bloku. Jeśli przeciągniesz pojedynczy blok operacji do obszaru skryptów i klikniesz go, wartość, którą przekazuje, pojawi się w dymku obok bloku:

pozycja X -120

Podczas przeciągania bloku operacji nad polem wejściowym innego bloku, wokół miejsca wejścia pojawia się białe „halo”, analogiczne do białej linii, która pojawia się, przy przyczepianiu bloku komendy w skrypcie:

Oto prosty skrypt, który będzie wykorzystywać blok operacji:

Tutaj operacja (funkcja) **pozycja X** wypełnia pierwsze pole wejścia do bloku **powiedz**.

(Pozycja X duszka to jego poziome położenie, odległość w lewo (wartości ujemne) lub w prawo (wartości dodatnie) od środka sceny. Podobnie pozycja Y jest mierzona w pionie, liczbą kroków w górę (dodatnia) lub w dół (ujemna) od centrum).

Operacje arytmetyczne możesz wykonywać za pomocą bloków operacji z palety Wyrażenia:

Blok **zaokrąglij** zaokrągla 60,2162 ... do 60, a blok + dodaje do tego 100. (Nawiasem mówiąc, blok **zaokrąglij** znajduje się w palecie Wyrażenia, podobnie jak +, ale w tym skrypcie ma jaśniejszy kolor z czarnym napisem, ponieważ Snap!, wykorzystuje jasne i ciemne wersje kolorów palety, gdy blok jest zagnieżdżony w innym bloku z tej samej palety:

Ten sposób poprawiania czytelności jest nazywany kolorowaniem w zebrę). Blok operacji z jego danymi wejściowymi, który zawiera inne bloki operacji, jak zaokrąglij pozycja X + 100, jest nazywany *wyrażeniem*.

## E Predykaty i obliczenia warunkowe

Większość operacji daje w wyniku liczbę, jak ( ) + ( ) lub ciąg tekstowy, jak np. połącz witaj świecie. Predykat jest specjalnym rodzajem operacji, który zawsze daje wynik **prawda** lub **fałsz**. Predykaty mają sześciokątny kształt:

przycisk myszy naciśnięty?

Ten specjalny kształt przypomina, że predykaty generalnie nie mają sensu w polu wejściowym bloków, które oczekują liczby lub tekstu. Nie powiedzielibyśmy przesuń o przycisk myszy naciśnięty? kroków, chociaż (jak widać na obrazku) Snap*!* pozwala to zrobić, jeśli naprawdę chcesz.
Zamiast tego zwykle używamy predykatów ze specjalnym sześciokątnym wejściem jak to:

C-kształtny blok **jeżeli** uruchamia swój skrypt wejściowy, jeśli (i tylko wtedy) wyrażenie w jego sześciokątnym polu wejściowym daje w wyniku **prawda**.

```
jeżeli pozycja Y < 0 to
    powiedz Pomocy! Jestem pod wodą!
```

Bardzo użyteczny w animacjach blok **powtarzaj aż** uruchamia swój skrypt wejściowy *wielokrotnie*, dopóki nie zostanie spełniony warunek (predykat da wynik prawda):

```
ustaw w stronę Duszek(2)
powtarzaj aż dotyka Duszek(2) ?
    przesuń o 3 kroków
```

Jeśli podczas pracy nad projektem chcesz chwilowo pominąć niektóre polecenia w skrypcie, ale nie chcesz o nich zapomnieć, możesz powiedzieć

```
przesuń o 10 kroków
obróć ↻ o 15 stopni
jeżeli fałsz to
    powiedz Nie zrobię tego. przez 2 s
    leć przez 1 s do x: 0 y: 0
ustaw kierunek na 90
```

Czasami chcesz wykonać to samo działanie, niezależnie od tego, czy jakiś warunek jest prawdziwy, czy fałszywy, ale z inną wartością wejściową. W tym celu możesz użyć bloku operacji **jeżeli**:[4]

```
powiedz
    połącz Jestem po jeżeli pozycja X < 0 to lewej inaczej prawej
```

Techniczne mówi się o wartości **prawda** lub **fałsz**, że jest to wartość „Boole'owska"; ma ona wielkie B, ponieważ nazwa pochodzi od osoby George'a Boole'a, który opracował matematyczną teorię wartości Boole'a. Niech Ci się to nie pomiesza; blok sześciokątny jest predykatem, a wartość, którą daje w wyniku, jest wartością logiczną czyli Boole'owską.

Jeszcze kilka manowców słownictwa: Wiele języków programowania rezerwuje nazwę „procedura" dla Komend (poleceń, które wykonują akcję) i używa nazwy „funkcja" dla Operacji i Predykatów. W tym podręczniku *procedura* jest dowolną akcją obliczeniową, łącznie z tymi, które przekazują wartości i tymi, które tego nie robią. Komendy, Operacje i Predykaty są wszystkie procedurami. Słowa „typu Procedury" są skrótem dla „typu Komendy, typu Operacji lub typu Predykatu."

---

[4] Jeśli nie widzisz tego na palecie Kontrola, kliknij przycisk Plik na pasku narzędzi i wybierz „Importuj narzędzia".

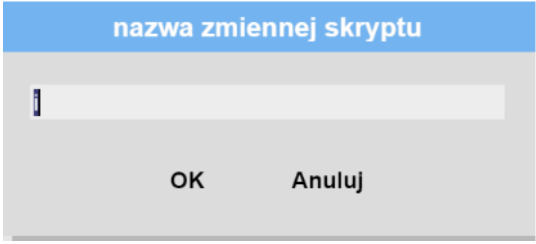
## F Zmienne

Wypróbuj ten skrypt:[5]

Wejście do bloku **przesuń** stanowi pomarańczowy owal. Aby go tam wstawić, przeciągnij pomarańczowy owal, który jest częścią bloku **dla**:

Pomarańczowy owal jest zmienną: symbolem reprezentującym wartość. (Zrobiłem ten zrzut ekranu przed zmianą drugiego pola wejściowy do bloku **dla** z domyślnego 10 na 200 i przed przeciągnięciem bloku **obróć**). **Dla** wielokrotnie wykonuje swój skrypt, podobnie jak **powtórz**, ale przed każdym powtórzeniem ustawia zmienną **i** na liczbę, rozpoczynając od tej, która znajduje się w pierwszym polu wejściowym, dodając do niej 1 przy każdym powtórzeniu, aż osiągnie ona wartość z drugiego pola wejściowego. W naszym skrypcie będzie 200 powtórzeń, najpierw z i = 1, potem z i = 2, potem 3, i tak dalej, aż do i = 200 dla ostatniego powtórzenia. W rezultacie każdy blok przesuń rysuje coraz dłuższy odcinek, dlatego obraz, który widzisz, jest rodzajem spirali. (Jeśli spróbujesz ponownie z kątem 90 stopni zamiast 92, zobaczysz, dlaczego ten rysunek nazywa się „squiral”[6].)

Zmienna **i** jest tworzona przez blok **dla** i może być używana tylko w skrypcie znajdującym się wewnątrz C-kształtnego wejścia bloku. (Nawiasem mówiąc, jeśli nie podoba ci się nazwa **i**, możesz ją zmienić, klikając pomarańczowy owal, nie przeciągając go, co otworzy okno dialogowe, w którym można wprowadzić inną nazwę:



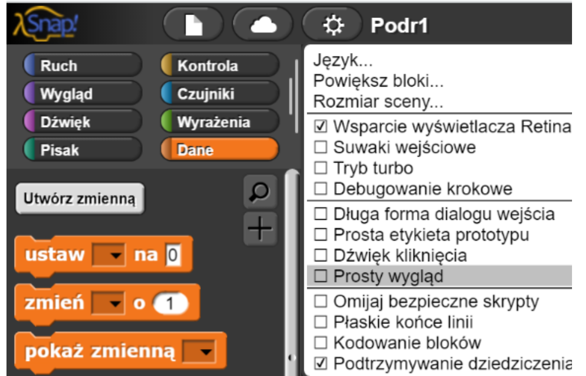

„I” nie jest bardzo opisową nazwą; możesz preferować „długość”, aby wskazać jej rolę w skrypcie. „I” jest nazwą tradycyjną, ponieważ matematycy używają liter od i do n dla reprezentowania wartości całkowitych, ale w językach programowania nie musimy ograniczać się do jednoliterowych nazw zmiennych.)

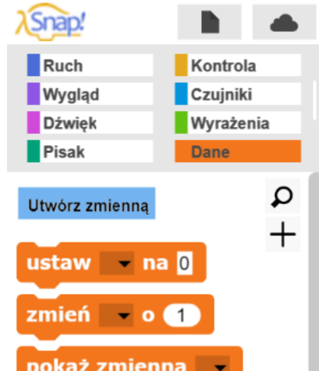
[5] Blok dla znajduje się również w bibliotece narzędzi; wybierz „Importuj narzędzia” z menu Plik, jeśli nie masz go palecie Kontroli.

[6] Zbitka angielskich słów: square (kwadrat) i spiral (spirala), po polsku pewnie kwadrala. [*przypis WtK*]

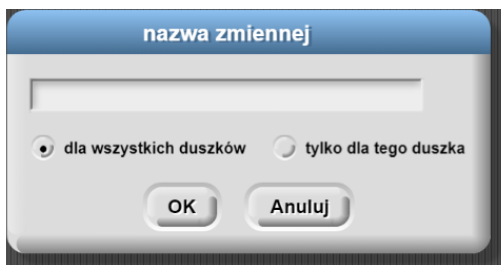
## Zmienne globalne

Możesz tworzyć „ręcznie” zmienne, których użycie nie jest ograniczone do jednego bloku (skryptu). Na górze palety Dane, kliknij przycisk „**Utwórz zmienną**”:[7]



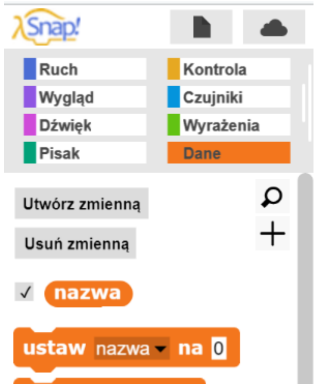




Pojawi się okno dialogowe, w którym możesz nadać swojej zmiennej nazwę:



Okno dialogowe daje ci również możliwość wyboru zmiennej dostępnej dla wszystkich duszków (która jest prawie zawsze tym, czego chcesz) lub uczynienia jej widoczną tylko w bieżącym duszku. Zrobisz to, jeśli zamierzasz nadać kilku duszkom indywidualne zmienne o tej samej nazwie, aby można było dzielić skrypty pomiędzy duszkami (przeciągając je z obszaru skryptu jednego duszka do obrazu innego duszka w zagrodzie duszków), a podczas uruchamiania tego skryptu różne duszki zrobią trochę inne rzeczy, ponieważ każdy ma inną wartość dla tej samej nazwy zmiennej.

Jeśli nadasz zmiennej nazwę „**nazwa**”, wtedy paleta Dane będzie wyglądać następująco:




Jest tu teraz przycisk „**Usuń zmienną**”, a pod nim pomarańczowy owal z nazwą zmiennej, podobny do pomarańczowego owalu w bloku **dla**. Możesz przeciągnąć zmienną do dowolnego skryptu w obszarze skryptów. Obok owalu znajduje się pole wyboru, początkowo zaznaczone. Gdy zostanie zaznaczone, zobaczysz także plakietkę *podglądu zmiennej* na scenie:

[7] Autor tłumaczenia zdecydowanie woli widok „Prosty wygląd”, z niego pochodzi ogromna większość zrzutów. Najbliższe dwa prezentuję podwójnie, żeby pokazać różnice [*przypis WtK*].

Kiedy nadasz zmiennej wartość, pomarańczowe pole jej podglądu wyświetli wartość.

Jak nadajesz jej wartość? Używasz bloku **ustaw**:

Zauważ, że nie przeciągasz owalu zmiennej do bloku ustaw! Klikasz na strzałkę w pierwszym polu wejściowym i uzyskujesz menu ze wszystkimi dostępnymi nazwami zmiennych.

## Zmienne skryptu

W powyższym przykładzie nasz projekt będzie kontynuował interakcję z użytkownikiem i chcemy zapamiętać jego imię podczas całego projektu. Jest to dobry przykład sytuacji, w której przydaje się zmienna globalna (rodzaj zmiennej, którą tworzysz za pomocą przycisku „**Utwórz zmienną**”). Innym częstym przykładem jest zmienna o nazwie „**wynik**” w projekcie gry. Czasami jednak potrzebujesz tylko zmiennej tymczasowej, podczas wykonywania określonego skryptu. W takim przypadku możesz użyć bloku **zmienne skryptu** do utworzenia zmiennej:

Podobnie jak w bloku **dla**, możesz kliknąć bez przeciągania pomarańczowy owal w bloku **zmienne skryptu**, aby zmienić jego nazwę. Możesz także utworzyć więcej niż jedną zmienną tymczasową, klikając strzałkę w prawo na końcu bloku, aby dodać owal kolejnej zmiennej:

## Zmiana nazw zmiennych

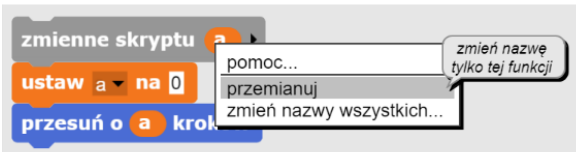
Istnieje kilka powodów, dla których warto zmienić nazwę zmiennej:

1. Ma domyślną nazwę, taką jak „**a**” w zmiennych skryptu lub „**i**” w bloku **dla**.
2. Koliduje z inną nazwą, na przykład zmienną globalną, której chcesz użyć w tym samym skrypcie.
3. Po prostu zdecydujesz, że inna nazwa będzie lepiej samo-dokumentująca.

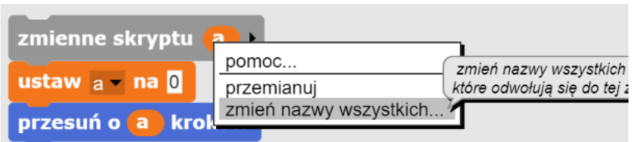
W pierwszym i trzecim przypadku prawdopodobnie chcesz zmienić nazwę wszędzie tam, gdzie pojawia się w tym skrypcie lub nawet we wszystkich skryptach. W drugim przypadku, jeśli już wcześniej zostały użyte w skrypcie obie zmienne, zanim zostało dostrzeżone, że mają taką samą nazwę, będziesz chcieć spojrzeć na każdy przypadek użycia zmiennej osobno, aby zdecydować, które nazwy zmienić. Obie te operacje są możliwe po kliknięciu prawym przyciskiem myszy lub kliknięciu z ctrl na owalu zmiennej.

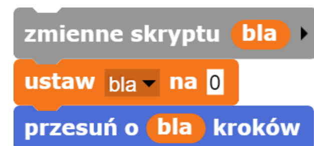
Jeśli klikniesz prawym przyciskiem myszy pomarańczowy owal w miejscu, w którym *użyto* zmiennej, wówczas możesz zmienić nazwę tylko tego jednego pomarańczowego owalu:

Jeśli klikniesz prawym przyciskiem myszy miejsce, w którym *zdefiniowana* jest zmienna (blok **zmienne skryptu**, pomarańczowy owal dla zmiennej globalnej w palecie Dane lub pomarańczowy owal wbudowany w blok, na przykład „**i**” w bloku **dla**), będziesz mieć dwie opcje zmiany nazwy, „zmień nazwę” i „zmień nazwy wszystkich”. Jeśli wybierzesz „zmień nazwę”, nazwa zmieni się tylko w tym pomarańczowym owalu, jak w poprzednim przypadku:



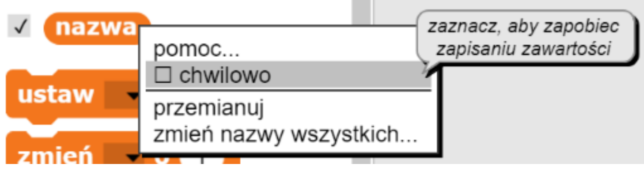
Jeśli jednak wybierzesz opcję „zmień nazwy wszystkich”, nazwa zostanie zmieniona w całym zakresie występowania zmiennej (skrypt dla zmiennej skryptu lub wszędzie dla zmiennej globalnej):




## Zmienne chwilowe

Do tej pory mówiliśmy o zmiennych z wartościami liczbowymi lub o krótkich łańcuchach tekstowych, takich jak np. czyjeś imię. Ale nie ma ograniczenia, co do ilości informacji, które można umieścić w zmiennej; w rozdziale IV zobaczysz, jak używać list do zbierania wielu wartości w jednej strukturze danych, a w rozdziale VIII, jak wczytać informacje z witryn internetowych. Kiedy używasz tych możliwości, twój projekt może zająć w komputerze dużo pamięci. Jeśli zbliżysz się do ilości pamięci dostępnej dla Snap*!*, wtedy może nie być możliwe zapisanie projektu. (Dodatkowe miejsce jest potrzebne do konwersji z wewnętrznej reprezentacji Snap*!* do postaci, w której projekty są eksportowane lub zapisywane). Jeśli twój program wczytuje dużo danych ze świata zewnętrznego, które będą nadal dostępne, kiedy użyjesz go następnym razem, może zechcesz przed zapisaniem projektu usunąć z pamięci wartości zawierające wiele danych. Aby to zrobić, kliknij w palecie Dane pomarańczowy owal zmiennej prawym przyciskiem myszy lub przytrzymując klawisz ctrl, aby wyświetlić takie menu:




Wiesz już o opcjach zmiany nazwy, **pomoc...** wyświetla ekran pomocy na temat zmiennych w ogóle. Tutaj interesuje nas pole wyboru obok **chwilowo**. Jeśli je zaznaczysz, to wartość tej zmiennej nie zostanie zapisana podczas zapisywania projektu. Oczywiście, musisz upewnić się, że gdy twój projekt się ładuje, odtwarza potrzebną wartość i ustawia na nią wartość zmiennej.

## G Debugowanie – poprawianie błędów

Snap*!* udostępnia kilka narzędzi ułatwiających debugowanie programu. Skupiają się wokół idei *pauzowania* działania skryptu, aby można było zbadać wartości zmiennych.

### Przycisk pauzy

Najprostszym sposobem wstrzymania programu jest ręczne kliknięcie przycisku pauzy w prawym górnym rogu okna. Podczas gdy program jest wstrzymany, możesz uruchamiać inne skrypty, klikając je, pokazywać zmienne na scenie zaznaczając pole wyboru obok zmiennej w palecie Dane lub uruchamiając blok **pokaż zmienną** i wykonywać wszystkie inne czynności, które chcesz zrobić, w tym modyfikowanie wstrzymanych skryptów poprzez dodawanie lub usuwanie bloków. Przycisk zmienia kształt na , a ponowne kliknięcie wznawia wstrzymane skrypty.

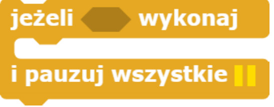
### Pułapki: blok pauzuj wszystko

Przycisk pauzy jest świetny, jeśli program wydaje się trwać w nieskończonej pętli, ale częściej będziesz chcieć ustawić punkt przerwania, konkretne miejsce skryptu, w którym chcesz go zatrzymać.

Blok, pauzuj wszystko znajdujący się u dołu palety Kontrola, można wstawić do skryptu, aby wstrzymać go po uruchomieniu. Na przykład, jeśli twój program nadaje komunikat o błędzie w konkretnym bloku, możesz użyć **pauzuj wszystko** tuż przed tym blokiem, aby przyjrzeć się wartościom zmiennych tuż przed wystąpieniem błędu.

Blok **pauzuj wszystko** zmienia kolor na błękitny podczas pauzy. Ponadto w trakcie pauzy możesz kliknąć uruchomiony skrypt prawym przyciskiem myszy, a podręczne menu daje możliwość wyświetlenia plakietek z wartością dla zmiennych tymczasowych skryptu:

Ale co robić, jeśli blok z błędem jest uruchamiany wiele razy w pętli, a błędy występują tylko, gdy konkretny warunek jest prawdziwy – powiedzmy, że wartość jakiejś zmiennej jest ujemna, co nigdy nie powinno się zdarzyć. W bibliotece narzędzi znajduje się blok przerwania, który umożliwia ustawienie *warunkowego* punktu przerwania i automatyczne wyświetlenie odpowiednich zmiennych przed zatrzymaniem. Wygląda tak:




A to przykład jego użycia:

(W tym wymyślonym przykładzie zmienna **zet** pochodzi spoza skryptu, ale jest związana z jego zachowaniem). Kiedy kontynuujesz (za pomocą przycisku pauzy), plakietki wartości tymczasowych zmiennych są usuwane przez ten blok przed wznowieniem skryptu. Blok warunkowy punktu przerwania nie jest magią; można alternatywnie po prostu wstawić blok **zatrzymaj wszystko** do bloku **jeżeli**.

Bloki **ukryj zmienną** i **pokaż zmienną** mogą być również używane do ukrywania i pokazywania podstawowych (pierwotnych) bloków w palecie. Menu rozwijane nie zawiera bloków podstawowych, ale istnieje ogólnie przydatna technika, aby dać wartości wejściowe bloku, które nie były spodziewane przy użyciu poleceń uruchom lub wywołaj:

uruchom ukryj zmienną z
z parametrami przesuń o 10 kroków

Aby użyć bloku jako wejścia w ten sposób, musisz jawnie umieścić wokół niego szarą obwiednię, klikając go prawym przyciskiem myszy i wybierając **obwiednia**. Więcej informacji na temat obwiedni w rozdziale VI.

## Wizualizacja pracy krokowej

Czasami nie masz pewności, gdzie znajduje się błąd, lub nie rozumiesz, jak program się tam dostał. Aby lepiej zrozumieć, zechcesz zapewne oglądać program podczas pracy z prędkością człowieka, a nie z prędkością komputera. Możesz to zrobić, klikając przed uruchomieniem skryptu lub podczas wstrzymania skryptu przycisk pracy krokowej (). Przycisk zaświeci się (), a suwak kontroli prędkości pojawi się na pasku narzędzi. Kiedy zaczynasz lub kontynuujesz skrypt, jego bloki i pola wejściowe będą świecić się na niebiesko po jednym w każdym kroku:

W tym prostym przykładzie, dane wejściowe do bloków są wartościami stałymi, ale jeśli dane wejściowe byłyby bardziej złożonym wyrażeniem obejmującym kilka bloków dających wyniki, każdy z nich zaświeciłby się, przy wywoływaniu. Zauważ, że dane wejściowe do bloku są obliczane przed wywołaniem samego bloku, więc na przykład **100** świeci się przed **przesuń**.

Prędkość kroku jest kontrolowana za pomocą suwaka. Jeśli przesuniesz suwak do końca w lewo, prędkość wynosi zero, przycisk pauzy zmienia się w przycisk kroku , a skrypt wykonuje jeden krok za każdym naciśnięciem przycisku. Nazywa się to *pracą krokową*.

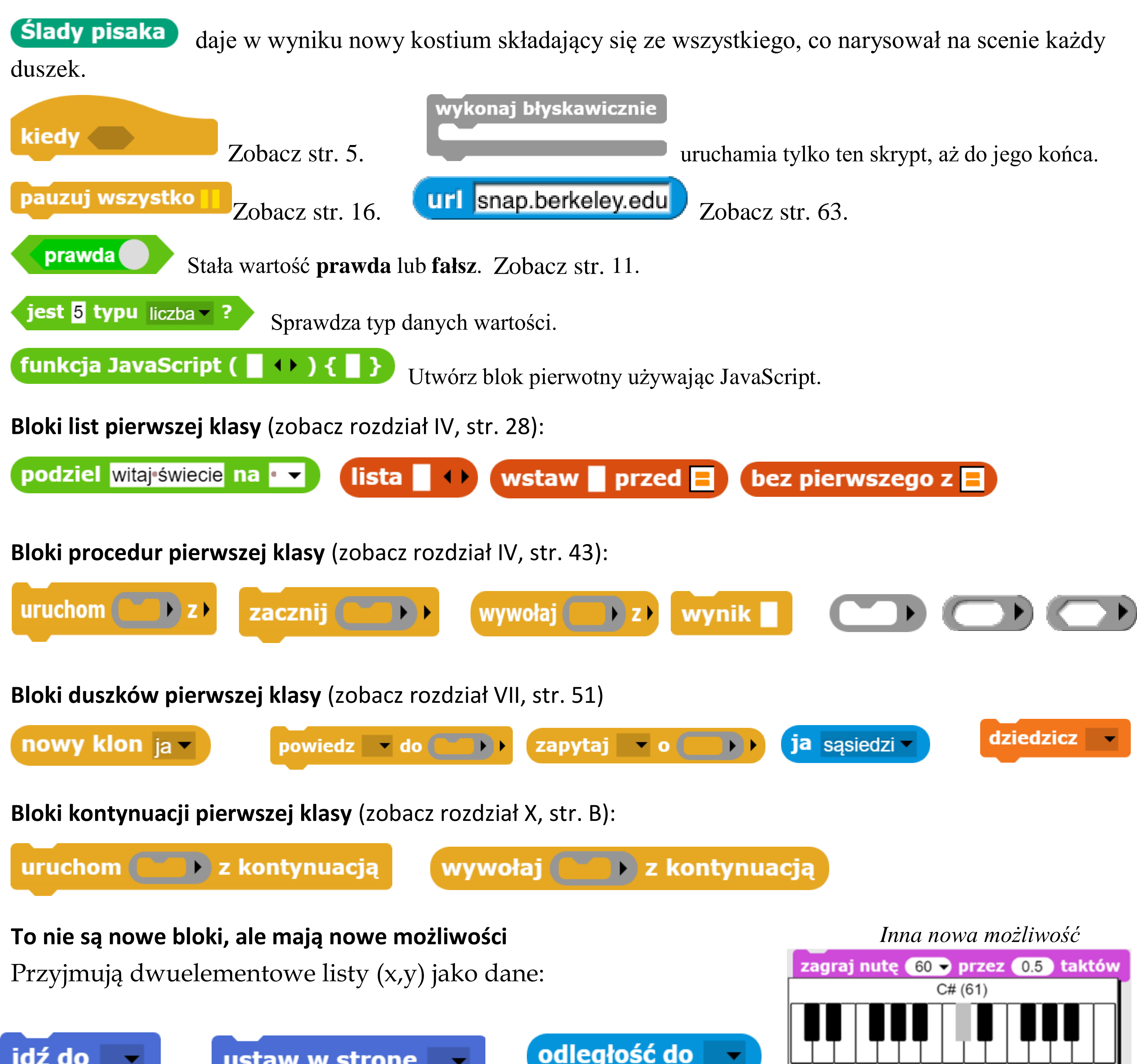
Jeśli kilka skryptów, które są widoczne w obszarze skryptów, jest uruchomionych w tym samym czasie, wszystkie są wykonywane równolegle. Rozważmy jednak przypadek dwóch pętli **powtórz** o różnej liczbie bloków. Podczas gdy nie korzystamy z pracy krokowej, każdy skrypt przechodzi cały cykl swojej pętli w każdym cyklu wyświetlania, pomimo różnicy w długości cyklu. Aby zapewnić, że widoczny wynik programu na scenie jest taki sam, w przypadku pracy krokowej, jak i w przypadku bez niej, krótszy skrypt będzie czekał na dole pętli, aby dłuższy skrypt mógł nadążyć.

Kiedy opowiemy o blokach niestandardowych w rozdziale III, będziemy mieli więcej do powiedzenia na temat pracy krokowej, ponieważ wpływa on na działanie tych bloków.

## H Etcetera

Ten podręcznik nie objaśnia szczegółowo każdego bloku. Istnieje wiele bloków ruchu, bloków dźwięku, bloków kostiumów i efektów graficznych i tak dalej. Możesz dowiedzieć się, co one wszystkie robią eksperymentując, a także czytając „ekrany pomocy”, które można uzyskać, klikając prawym przyciskiem myszy lub klikając blok z wciśniętym klawiszem ctrl i wybierając „**pomoc ...**” z menu, które się pojawi. Jeśli zapomnisz, w której palecie (i jakiego koloru) jest blok, ale pamiętasz przynajmniej część jego nazwy, naciśnij ctrl-F i wpisz nazwę w polu tekstowym, które pojawi się w obszarze palety (możesz też kliknąć lupę na górze w obszarze palety [*dod. WtK*]).

Oto pierwotne, podstawowe bloki, których nie ma w Scratchu:




daje w wyniku nowy kostium składający się ze wszystkiego, co narysował na scenie każdy duszek.

Zobacz str. 5.

uruchamia tylko ten skrypt, aż do jego końca.

Zobacz str. 16.

Zobacz str. 63.

Stała wartość **prawda** lub **fałsz**. Zobacz str. 11.

Sprawdza typ danych wartości.

Utwórz blok pierwotny używając JavaScript.

**Bloki list pierwszej klasy** (zobacz rozdział IV, str. 28):

**Bloki procedur pierwszej klasy** (zobacz rozdział IV, str. 43):

**Bloki duszków pierwszej klasy** (zobacz rozdział VII, str. 51)

**Bloki kontynuacji pierwszej klasy** (zobacz rozdział X, str. B):

**To nie są nowe bloki, ale mają nowe możliwości**

Przyjmują dwuelementowe listy (x,y) jako dane:

*Inna nowa możliwość*

Biblioteka narzędzi (bloki utworzone w Snap!):

etykieta Hello! o rozmiarze 12 — Drukuje tekst w podanym rozmiarze (w punktach) na scenie, w pozycji duszka i wg jego kierunku. Duszek przesuwa się na koniec tekstu. (Dobrze, to nie zawsze, jest to czego chcesz, ale możesz zapisać pozycję duszka przed użyciem i czasami musisz wiedzieć, jak duży będzie się tekst, w krokach żółwia). Jeśli pisak jest przyłożony, tekst zostanie podkreślony.

jeżeli to inaczej — Operacja, odpowiednik bloku **jeżeli/w przeciwnym razie**. Obliczana jest tylko jedna z dwóch gałęzi, w zależności od wartości warunku.

dla i = 1 do 10 — Blok pętli jak w przypadku **powtórz**, ale ze zmienną indeksową.

ignoruj — Czasami chcesz uruchomić operację, nie przejmując się wynikiem. Wklej ją w polu wejścia tego bloku.

**Wyjście nielokalne** (zobacz str. 71):

złap tag　rzuć catchtag　złap tag　rzuć catchtag

połącz słowa — jak pierwotne **połącz**, ale wstawia spację między kolejnymi elementami wejścia.

**Zamiany pomiędzy słowami i listami,** znaki (bloki słów) lub słowa (bloki zdań) są traktowane jak elementy:

słowo → lista　lista → słowo　zdanie → lista　lista → zdanie

Blok bez nazwy. Użyj go, aby umieścić dowolną wartość wejściową w polu wejściowym, które oczekuje liczby, listy lub innego typu tekstu.

liczby od 1 do 10 — Pobiera dwie dane liczbowe i daje w wyniku listę liczb z tego zakresu, włącznie z końcami.

**Funkcje wyższego stopnia na listach** (zobacz rozdział IVD, str. 31):

mapuj na　zachowaj elementy takie że z

potraktuj tym elementy z

dla każdego element z — Wyższego rzędu, ale nie funkcja: Wykonaj akcję dla każdego elementu z listy. Gwarantowane przejście listy od lewej do prawej strony. (Nie dotyczy to znajdujących się powyższej funkcji wyższego rzędu, chociaż podane wartości są prawidłowe).

puste?　is linked? — Funkcje użytkowe stosowane w realizacji HOF (w bibliotece narzędzi nie widzę drugiego bloku [*dod. WtK*]).

# II Zapisywanie i ładowanie projektów i mediów

Po utworzeniu projektu będziesz chcieć go zapisać, aby mieć do niego dostęp przy następnym użyciu Snap*!*. Jest na to kilka sposobów. Możesz zapisać projekt na swoim komputerze lub możesz go zapisać na stronie internetowej Snap*!*. Zaletą zapisywania w sieci jest to, że masz dostęp do swojego projektu, nawet jeśli używasz innego komputera lub urządzenia mobilnego, takiego jak tablet lub smartfon. Zaletą zapisywania na komputerze jest to, że masz dostęp do zapisanego projektu podczas lotu samolotem lub w przypadku gdy brak sieci. Dlatego mamy wiele sposobów zapisu.

## A Zapis lokalny

Istnieją dwa różne sposoby zapisania projektu (lub pliku multimedialnego, takiego jak kostium) na komputerze. Powodem tej złożoności jest JavaScript, w którym Snap*!* jest zaimplementowany, celowo ogranicza on zdolność programów działających w przeglądarce do wpływania na komputer. To dobrze, ponieważ oznacza to, że możesz śmiało uruchomić czyjś projekt Snap*!* bez obawy, że usunie wszystkie pliki lub zainfekuje komputer wirusem. Ale to trochę komplikuje sprawę.

### Magazyn przeglądarki

Znajdź ikonę pliku ( ) na pasku narzędzi. W menu, które pojawi się po kliknięciu, wybierz opcję „Zapisz jako ...”. Pojawi się okno takie jak to:

Wybrana jest opcja „Przeglądarka”, co oznacza, że twój projekt zostanie zapisany w specjalnym pliku, który może być odczytany tylko na tym samym komputerze, przez tę samą przeglądarkę, podłączoną do tej samej strony internetowej (Snap*!*). Nie zobaczysz go na liście plików poza Snap*!*. W ten sposób JavaScript chroni Cię przed złośliwym oprogramowaniem w zapisanych projektach.

Na powyższym obrazku wąskie okno u góry to miejsce, w którym podajesz nazwę, pod którą chcesz zapisać projekt. Po lewej stronie widzisz listę projektów, które zostały już zapisane w pamięci przeglądarki. Po prawej stronie znajduje się obrazek sceny i pole tekstowe „notatek projektu”: wszelkie informacje, które chcesz przekazać użytkownikowi projektu. Obie te rzeczy są zapisywane wraz z projektem.

Ważnym ograniczeniem tego sposobu zapisywania projektów jest to, że przeglądarka ustawi dla wszystkich witryn razem limit całkowitej dostępnej pamięci. (Ten limit można ustawić w preferencjach przeglądarki). Tak więc „zapisywanie projektów w lokalnym magazynie” (magazyn lokalny (local store)) to oficjalna nazwa pamięci przeglądarki) ogranicza się tylko do kilku projektów. Ponadto, jeśli twoja przeglądarka jest skonfigurowana by blokować ciasteczka ze stron internetowych (kolejne ustawienie w preferencjach), to nie pozwoli też na zapis w lokalnym magazynie. Dlatego magazyn lokalny dla projektów jest w zasadzie wypierany przez eksport opisany na następnej stronie.

### Eksport XML

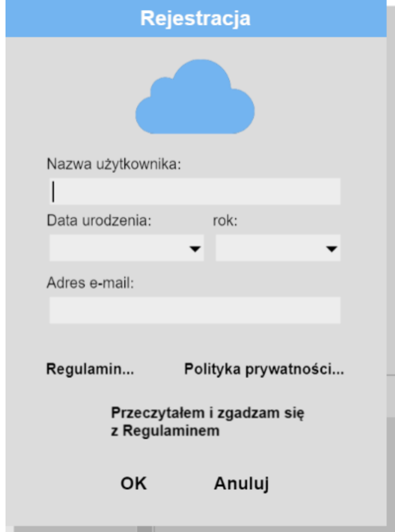
Drugi sposób zapisania projektu na komputerze wymaga dwóch kroków, ale nie ma ograniczeń lokalnego magazynu. Projekty zapisane w ten drugi sposób są normalnymi plikami zapisanymi na komputerze i mogą być udostępniane znajomym, mogą być otwierane w dowolnej przeglądarce i nie mają ograniczeń rozmiaru.

Z menu Plik wybierz „Eksportuj projekt ...” W zależności od przeglądarki nastąpi jedna z dwóch rzeczy. W niektórych przypadkach projekt zostanie pobrany na komputer użytkownika i zapisany w pliku w folderze pobierania. W innych przypadkach całe okno Snap*!* zniknie, zastąpione czegoś, co wygląda jak bełkot. Nie panikuj! Tak powinno się stać. Patrzysz na swój projekt w notacji o nazwie XML. Głównym powodem, dla którego wygląda on na bełkot, jest to, że zawiera kodowanie obrazów i innych mediów zawartych w projekcie. Jeśli przejrzysz kod XML, to zobaczysz, że skrypty projektu są całkiem czytelne, chociaż nie wyglądają jak bloki Snap*!*. Snap*!* otworzył dla tego tekstu XML nową kartę przeglądarki; okno Snap! wciąż tam jest schowane w swojej pierwotnej karcie.

Ale tekst XML nie jest do czytania. Po przejrzeniu tej karty użyj polecenia Zapisz w przeglądarce (w menu Plik lub jak zwykle za pomocą skrótu: cmd-S (Mac) lub ctrl-S (wszystko inne). Możesz wybrać nazwę pliku i zostanie on zapisany w folderze pobierania. Następnie możesz zamknąć tę kartę i wrócić do karty Snap*!*.

## B Zapis w chmurze

Inną możliwością jest zapisanie twojego projektu „w chmurze” na stronie internetowej Snap*!*. Aby to zrobić, trzeba mieć u nas konto. Kliknij przycisk Chmura ( ) na pasku narzędzi. Wybierz opcję „Rejestracja ...”. Pokaże się okno, które wygląda tak




Musisz wybrać nazwę użytkownika, która będzie Cię identyfikować na stronie internetowej, na przykład **Jens** lub **bh**. Jeśli jesteś użytkownikiem Scratcha, możesz użyć także w Snap*!* swojej nazwy ze Scratcha. Jeśli jesteś dzieckiem, nie wybieraj nazwy użytkownika, która zawiera nazwisko, ale imiona i inicjały są w porządku. Nie wybieraj czegoś, co Cię zawstydzi, gdy zobaczą to inni użytkownicy (lub twoi rodzice)! Jeśli wybrana nazwa jest już zajęta, musisz wybrać inną.

Prosimy o twój miesiąc i rok urodzenia; używamy tych informacji tylko, aby zdecydować, czy poprosić o twój własny adres e-mail czy też o adres e-mail rodzica. (Jeśli jesteś dzieckiem, nie możesz rejestrować się w sieci, nawet do Snap*!*, bez wiedzy twojego rodzica). Nie przechowujemy informacji o twojej dacie urodzenia na naszym serwerze; jest wykorzystywany na twoim komputerze tylko podczas tej wstępnej rejestracji. Nie pytamy o twoją dokładną datę urodzenia, nawet ten jeden raz, ponieważ jest to ważny element danych osobowych.

Po kliknięciu przycisku OK na podany adres e-mail zostanie wysłana wiadomość z początkowym hasłem do konta. Twój adres e-mail zachowujemy w pliku, dzięki czemu, jeśli zapomnisz hasła, możemy wysłać Ci link do zresetowania hasła. Prześlemy Ci również wiadomość, jeśli Twoje konto zostanie zawieszone z powodu naruszenia warunków korzystania z usługi. Nie używamy Twojego adresu w żadnym innym celu. Nigdy nie otrzymasz za pośrednictwem tej strony żadnych e-maili marketingowych, ani od nas, ani od osób trzecich. Jeśli jednak obawiasz się podania tych informacji, wyszukaj w sieci „tymczasowy e-mail".

Na koniec musisz przeczytać i zaakceptować „Warunki korzystania z usługi". Krótkie podsumowanie: nie przeszkadzaj nikomu w korzystaniu z witryny i nie umieszczaj w projektach, które udostępniasz innym użytkownikom materiałów chronionych prawami autorskimi ani danych osobowych. I nie jesteśmy odpowiedzialni, jeśli coś pójdzie nie tak. (Nie oczekujemy, że coś pójdzie nie tak, ponieważ Snap*!* działa w JavaScript w przeglądarce i jest mocno odizolowany od reszty twojego komputera, ale prawnicy każą nam to powiedzieć).

Po utworzeniu konta możesz zalogować się do niego, używając opcji „Logowanie..." z menu Chmura:

Użyj nazwy użytkownika i hasła, które zostało przez Ciebie wcześniej podane. Jeśli zaznaczysz pole "Zapamiętaj mnie na tym komputerze aż do wylogowania", zostaniesz automatycznie zalogowany przy następnym uruchomieniu Snap*!* z tej samej przeglądarki i na tym samym komputerze. Zaznacz to pole, jeśli korzystasz z własnego komputera i nie udostępniasz go rodzeństwu. Nie zaznaczaj tego pola, jeśli korzystasz z publicznego komputera w bibliotece, w szkole itp.

Po zalogowaniu możesz wybrać opcję „Chmura" w oknie dialogowym „Zapisz jako ..." pokazanym na str. 20. Wprowadzasz nazwę projektu i opcjonalnie notatki do projektu, podobnie jak w przypadku zapisywania w „magazynie lokalnym" (w przeglądarce), ale twój projekt zostanie zapisany online i można będzie go załadować z dowolnego miejsca z dostępem do sieci.

## C Ładowanie zapisanych projektów

Po zapisaniu projektu, będziesz chcieć załadować go ponownie do Snap*!*. Można to zrobić na dwa sposoby:

1) Jeśli projekt został zapisany w przeglądarce lub na koncie Snap*!*, wybierz z menu Plik opcję „Otwórz ...". Wybierz przycisk „Przeglądarka" lub „Chmura", a następnie wybierz w dużym polu tekstowym projekt z listy i kliknij OK. (Trzeci przycisk „Przykłady" pozwala wybrać przykładowe projekty, które oferujemy. Możesz zobaczyć, o czym jest każdy z nich, klikając go i czytając notatki projektu.)

2) Jeśli zapisałeś projekt na komputerze jako plik XML, wybierz „Importuj ..." z menu Plik. W ten sposób otworzysz zwykłe okno otwierania plików przeglądarki, w którym możesz nawigować tak jak w innym oprogramowaniu. Alternatywnie znajdź plik XML na swoim pulpicie i po prostu przeciągnij go na okno Snap*!*.

Drugi sposób pozwala również na importowanie do projektu mediów (kostiumów i dźwięków). Po prostu wybierz „Importuj...", a następnie wybierz obraz lub plik dźwiękowy zamiast pliku XML.

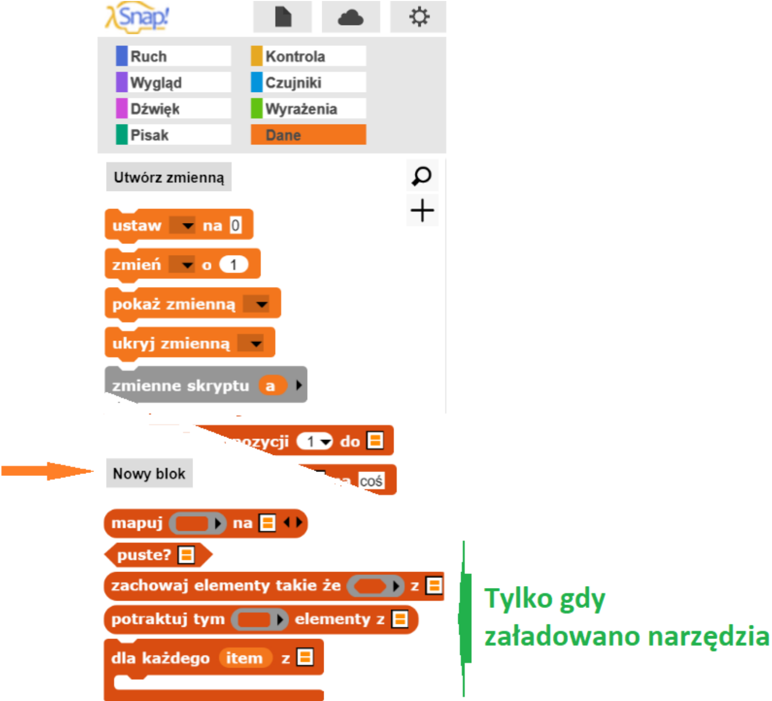
Snap*!* może także importować projekty stworzone w BYOB 3.0 lub w Scratch 1.4 lub 2.0 (z pewnym wysiłkiem, zobacz naszą stronę internetową). Prawie wszystkie takie projekty działają poprawnie w Snap*!*, oprócz niewielkiej liczby niekompatybilnych bloków. Projekty BYOB 3.1, które nie używają duszków pierwszej klasy, jak i w większości projekty BYOB 3.1 będą działać w Snap*!* 4.1.

# III Budowanie nowego bloku

Pierwsza wersja Snap*!* nosiła nazwę BYOB, skrót od „Build Your Own Blocks”. Własne bloki to była pierwsza i wciąż jest najważniejsza możliwość dodana do Scratcha 1.4. (Nazwa programu została zmieniona, ponieważ niektórzy nauczyciele nie mają poczucia humoru. ☹ Wybieraj bitwy rozważnie.). Nowy Scratch 2.0 ma również częściową możliwość tworzenia własnych bloków.

## A Proste bloki

W palecie Dane, na dole znajduje się przycisk oznaczony „Nowy blok”.



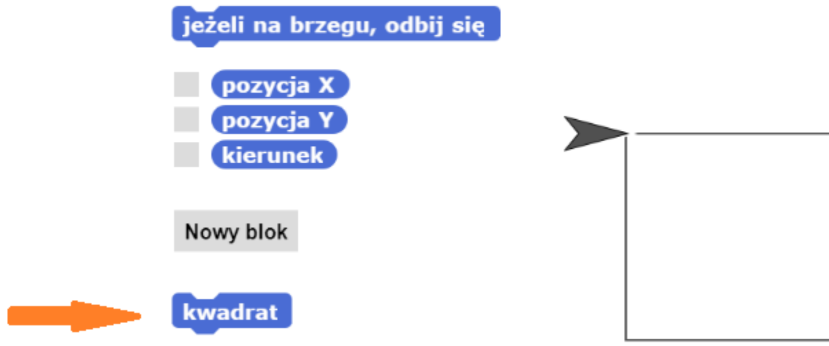

Po kliknięciu tego przycisku wyświetli się okno dialogowe, w którym wybierzesz nazwę bloku, kształt i paletę (kolor bloku). Zdecydujesz również, czy blok będzie dostępny dla wszystkich duszków, czy tylko dla aktualnego duszka i jego dzieci. Uwaga: Możesz również przejść do okna dialogowego „Nowy blok”, klikając w pustym miejscu obszaru skryptów prawym przyciskiem myszy (lub kliknięciem z ctrl), a następnie wybierając „buduj nowy blok” z menu, które się pojawi.

W tym oknie dialogowym możesz wybrać paletę, kształt i nazwę bloku. Z jednym wyjątkiem, jest tylko jeden kolor na paletę, np. wszystkie bloki ruchu są niebieskie. Jednak paleta Dane zawiera pomarańczowe bloki związane z zmiennymi i czerwone bloki związane z listami. Oba kolory są dostępne jest też opcja "Inne", która tworzy szare bloki w palecie Dane dla bloków, które nie pasują do żadnej kategorii.

Istnieją trzy kształty bloków, zgodnie z konwencją, która powinna być znana użytkownikom Scratcha: Bloki w kształcie cegiełek (kawałków układanki) są komendami i nie dają w wyniku wartości. Owalne bloki są operacjami (funkcjami), a sześciokątne predykatami, co jest technicznym terminem dla operacji, które dają w wyniku wartości logiczne (prawda lub fałsz).

Załóżmy, że chcesz utworzyć blok „kwadrat", który rysuje kwadrat. Wybierz Ruch, Komenda i wpisz „kwadrat" w polu nazwy. Po kliknięciu przycisku OK następuje przejście do Edytora bloków. Działa to tak samo, jak tworzenie skryptu w obszarze skryptów duszka, z wyjątkiem tego, że blok „kapeluszowy" znajdujący się u góry ma zamiast czegoś w stylu „kiedy zostanę kliknięty" obrazek (nazwę) bloku, który budujesz. Ten blok kapelusza nazywa się prototypem nowego bloku[8]. Aby zbudować własny blok, przeciągasz bloki pod ten kapelusz, a następnie klikasz OK:

Twój blok pojawi się u dołu palety Ruch. Oto blok i wynik jego użycia:

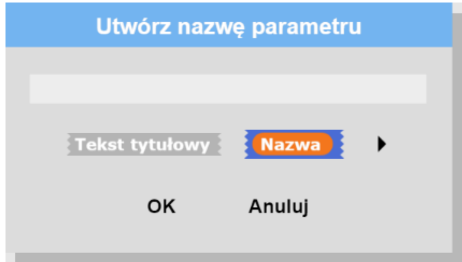



[8] Takie użycie słowa „prototyp" nie ma związku z programowaniem zorientowanym obiektowo, omówionym później.

## Nowe bloki z polem wejścia (parametrem)

Załóżmy jednak, że chcesz rysować kwadraty o różnych rozmiarach. Kliknij prawym klawiszem myszy na bloku, wybierz „edytuj…”, a otworzy się Edytor bloków. Zwróć uwagę na znaki plus przed i za słowem kwadrat w bloku prototypowym. Jeśli najedziesz myszą na jeden z nich, zmieni on kolor:

Kliknij plus z prawej strony. Zostanie wyświetlone okno dialogowe „Utwórz nazwę parametru”:




Wpisz nazwę „**bok**” i kliknij OK. W tym oknie dialogowym znajdują się inne opcje; możesz wybrać „**tekst tytułowy**”, jeśli chcesz dodać słowa do nazwy bloku, aby za polem wejścia pojawił się tekst, jak w bloku „**przesuń () kroków**”. Możesz też wybrać bardziej rozbudowane okno dialogowe z wieloma opcjami dotyczącymi nazwy wejścia. Ale zostawmy to na później. Po kliknięciu przycisku OK wprowadzone dane pojawiają się w prototypie bloku:

Możesz teraz przeciągnąć pomarańczową zmienną do skryptu, a następnie kliknąć OK:

Twój blok pojawi się teraz na palecie Ruch z oknem wejściowym:

Możesz narysować kwadrat dowolnej wielkości, wpisując długość jego boku w polu i uruchamiając blok, jak zwykle, przez ~~dwukrotne~~ kliknięcie lub umieszczenie go w skrypcie (skreślenie WtK).

## B Rekurencja

Ponieważ nowy blok pojawia się w palecie, gdy tylko zaczniesz go edytować, możesz budować bloki rekurencyjne (bloki, które wywołują siebie), przeciągając blok do jego definicji:

Jeśli rekurencja jest dla ciebie nowa, oto kilka wskazówek: Ważne jest, aby rekurencja miała podstawowy przypadek, to znaczy mały (najmniejszy) przypadek, który blok może zrealizować bez użycia rekurencji. W tym przykładzie jest to **głębokość (n)** = 0, dla której blok nie robi nic, ponieważ skrypt jest zamknięty w bloku **jeżeli**. Bez podstawowego przypadku rekursja działałaby zawsze, wołając siebie w kółko.

Nie próbuj prześledzić dokładnej sekwencji kroków, które komputer wykonuje w programie rekurencyjnym. Zamiast tego wyobraź sobie, że wewnątrz komputera jest wielu małych ludzi i jeśli Dorota rysuje drzewo rozmiaru 100, głębokości 6, wynajmuje Darka, aby utworzyć drzewo o rozmiarze 70, głębokości 5, a później wynajmuje Darię, aby stworzyć kolejne drzewo rozmiaru 70, głębokości 5. Darek z kolei zatrudnia Dominika i Daniela i tak dalej. Każda mała osoba ma swoje lokalne zmienne rozmiaru i głębokości o innych wartościach.

Możesz także budować operacje rekurencyjne, takie jak blok obliczający funkcję silnia:

Zwróć uwagę na użycie bloku **wynik**. Kiedy blok operacji realizuje ten blok, kończy swoją pracę i przekazuje podaną wartość; żadne dalsze bloki w skrypcie nie są wykonywane. Tak więc blok **jeżeli … w przeciwnym razie** w powyższym skrypcie mógłby być zwykłym **jeżeli** z drugim blokiem **wynik** pod nim zamiast wewnątrz, a rezultat byłby taki sam, ponieważ kiedy pierwszy **wynik** jest realizowany w przypadku podstawowym (n=0), to kończy wywoływanie bloku, a drugi **wynik** jest ignorowany. Istnieje również blok **zatrzymaj ten blok**, który ma podobny cel, skończyć wcześniej wywołanie bloku dla bloków poleceń. (W przeciwieństwie do tego blok **zatrzymaj ten skrypt** zatrzymuje nie tylko bieżący blok wywołujący, ale także wszystkie skrypty, które go wywołały do najwyższego poziomu).

Oto nieco bardziej zwarty sposób na napisanie funkcji silni:

+silnia+ n +
wynik jeżeli n = 0 to 1 inaczej n × silnia n − 1

(Jeśli w palecie Kontrola nie widzisz bloku operacji **jeżeli**, kliknij przycisk pliku na pasku narzędzi i wybierz „Importuj narzędzia.”)
Aby uzyskać więcej informacji na temat rekurencji, zobacz *Thinking Recursively* Erica Roberts’a. (Pierwotne wydanie to ISBN 978-0471816522; nowsze *Thinking Recursively in Java* to ISBN 978-0471701460.)

## C Biblioteki bloków

Podczas zapisywania projektu (patrz Rozdział II powyżej) wszystkie utworzone przez ciebie bloki niestandardowe są zapisywane razem z nim. Ale czasami chcesz zapisać kolekcję bloków, które możesz wykorzystać w więcej niż jednym projekcie. Przykładem jest biblioteka narzędzi, z której korzystamy w tym podręczniku. Być może Twoje bloki implementują określoną strukturę danych (stos, słownik, itp.) lub są podstawą do budowy wielopoziomowej gry. Taki zbiór bloków nazywa się *biblioteką bloków*.

Tworzenie biblioteki bloków odbywa się za pomocą mechanizmu eksportu XML opisanego na stronie 21, z tą różnicą, że wybierasz „Eksportuj bloki ...” z menu Plik zamiast „Eksportuj projekt ...”. Gdy to zrobisz, zobaczysz okno takie jak to:

Okno pokazuje wszystkie twoje bloki niestandardowe. Możesz odznaczyć niektóre pola wyboru, aby wybrać tylko te bloki, które chcesz uwzględnić w swojej bibliotece. (Możesz kliknąć okno eksportu prawym przyciskiem myszy lub przytrzymując klawisz ctrl, aby otworzyć menu, które pozwala zaznaczyć lub odznaczyć wszystkie pola naraz.) Następnie naciśnij OK. Zobaczysz nową kartę z definicjami blokowymi zakodowanymi w XML, którą zapiszesz za pomocą polecenia Zapisz w przeglądarce.

Aby zaimportować bibliotekę bloków, użyj polecenia „Importuj ...” w menu Plik lub po prostu przeciągnij plik XML do okna Snap*!*.

# IV Listy pierwszej klasy

W języku programowania typ danych jest pierwszej klasy, jeśli dane tego typu mogą być

- wartością zmiennej
- parametrem procedury
- wartością przekazywaną przez procedurę
- elementem zbioru danych
- anonimowe (nie nazwane)

W Scratchu liczby i ciągi tekstowe są pierwszej klasy. Możesz wstawić liczbę do zmiennej, użyć jej jako parametru bloku, wywołać funkcję, która przekazuje liczbę lub umieść liczbę na liście.

Ale listy Scratcha nie należą do pierwszej klasy. Możesz utworzyć listę za pomocą przycisku „**Utwórz listę**", który wymaga podania nazwy listy. Nie możesz umieścić listy w zmiennej, w polu wejściowym bloku lub w elemencie listy – nie możesz mieć listy list. Żadna z funkcji Scratcha nie przekazuje wartości listy. (Możesz użyć redukcji listy do ciągu tekstowego jako danych wejściowych do innych bloków, ale to spowoduje utratę struktury listy, a dane wejściowe to tylko ciąg tekstowy, a nie agregat danych).

Podstawową zasadą na której zbudowano Snap*!* jest to, że ***wszystkie dane powinny być pierwszej klasy***. Jeśli coś występuje w języku, powinniśmy móc z tego korzystać w pełni i swobodnie. Uważamy, że ta zasada pozwala uniknąć potrzeby dodawania wielu specjalnych narzędzi, zamiast tego sami użytkownicy mogą je napisać w Snap*!*.

Zauważ, że to typ danych jest pierwszej klasy, a nie pojedyncza wartość. Nie myśl, na przykład, że niektóre listy są pierwszej klasy, a inne nie. W Snap*!*, listy są pierwszej kategorii, kropka.

## A Blok lista

W samym sercu zagadnienia konstrukcji list pierwszej klasy jest możliwość utworzenia "anonimowej" listy, aby utworzyć listę bez jednoczesnego nadania jej nazwy. Blok funkcyjny **lista** to robi.

lista ◂▸ — Jak pojawia się w palecie Dane

lista She Loves You ◂▸

Elementy listy nie muszą być tego samego typu

lista (lista A imię jego ◂▸) 44 ◂▸

lista ▸ — Kliknięcie lewej strzałki umożliwia utworzenie pustej listy

Po prawej stronie bloku znajdują się dwie strzałki: w lewo i prawo. Kliknięcie ich powoduje zmianę liczby pól wejść do listy, tj. liczby elementów na liście, którą budujesz.

Możesz wstawić ten blok do pola wejściowego wielu innych bloków:

Snap*!* nie ma przycisku „**Utwórz listę**” takiego jak w Scratchu. Jeśli chcesz mieć globalną listę z nazwą, utwórz zmienną globalną i użyj bloku **ustaw**, aby umieścić listę w zmiennej.

## B Listy list

Listy można wstawiać jako elementy w większych listach. W razie potrzeby możemy łatwo tworzyć struktury ad hoc:

| 4 | A | B |
|---|---|---|
| 1 | John | Lennon |
| 2 | Paul | McCartney |
| 3 | George | Harrison |
| 4 | Ringo | Star |

Zauważ, że ta lista jest prezentowana w innym formacie niż lista powyżej „Ona cię kocha”. Dwuwymiarowa lista jest nazywana tabelą i domyślnie jest wyświetlana w widoku tabeli. Będziemy mieli więcej do powiedzenia na ten temat później.

Możemy również zbudować dowolną klasyczną informatyczną strukturę danych z list list, definiując konstruktory (bloki tworzące przykładową strukturę), selektory (bloki przekazujące fragment struktury) i mutatory (bloki zmieniające zawartość struktury), jeśli tylko tego potrzebujemy. Tutaj tworzymy drzewa binarne z selektorami, które sprawdzają wprowadzanie poprawnego typu danych; pokazany jest tylko jeden selektor, ale te dla lewych i prawych potomków są analogiczne.

```
drzewo binarne dane, dane , lewy potomek lewy , prawy potomek prawy
wynik lista "drzewo-binarne" dane lewy prawy
```

```
dbin-dane drzewo
jeżeli jest drzewo typu lista ? to
  jeżeli element 1 z drzewo = "drzewo-binarne" to
    wynik element 2 z drzewo
  w przeciwnym razie
    powiedz połącz drzewo nie jest drzewem binarnym
    zatrzymaj ten skrypt
w przeciwnym razie
  powiedz połącz drzewo nie jest drzewem binarnym
  zatrzymaj ten skrypt
```

## C Funkcyjne i imperatywne programowanie list

Istnieją dwa sposoby tworzenia listy w programie. Użytkownicy Scratcha będą zaznajomieni z imperatywnym stylem programowania, który opiera się na zestawie bloków poleceń, które modyfikują listę:

dodaj coś do
usuń 1 z
wstaw coś na pozycji 1 do
zamień element 1 z na coś

Jako przykład jest tutaj blok, który pobiera listę liczb jako dane wejściowe i daje w wyniku nową listę zawierającą tylko liczby parzyste z oryginalnej listy:

+parzyste+z+ lista liczb +
zmienne skryptu wynik
ustaw wynik na lista
dla i = 1 do długość lista liczb
jeżeli element i z lista liczb modulo 2 = 0 to
dodaj element i z lista liczb do wynik
wynik wynik

W tym skrypcie najpierw tworzymy zmienną tymczasową, umieszczamy w niej pustą listę, a następnie przechodzimy przez elementy listy wejściowej używając bloku **dodaj … do (wynik)**, aby zmodyfikować listę wynikową, dodając po jednym elemencie i wreszcie przekazać wynik.

Programowanie *funkcyjne* jest innym podejściem, które staje się ważne w programowaniu w „realnym świecie" z powodu równoległości, tj. faktu, że różne procesory mogą manipulować tymi samymi danymi w tym samym czasie. To sprawia, że używanie mutacji (zmieniającej wartość związaną ze zmienną) jest problematyczne, ponieważ niemożliwe jest poznanie dokładnej sekwencji zdarzeń, więc wynik mutacji może nie być tym, czego spodziewa się programista. Nawet bez równoległości programowanie funkcyjne jest czasami prostszą i bardziej efektywną techniką, szczególnie w przypadku struktur danych zdefiniowanych rekurencyjnie. Używa do budowania wartości listy bloków operacji (dających wynik), a nie bloków komend:

wstaw przed
element 1 z
bez pierwszego z

W programie funkcyjnym często używamy rekurencji do tworzenia listy, po jednym elemencie naraz. Blok **wstaw … przed …** tworzy listę, która ma jeden element dodany na początku istniejącej listy, bez zmiany wartości oryginalnej listy. Niepusta lista jest przetwarzana przez podzielenie jej na pierwszą pozycję (**element 1 z**) i całą resztę pozycji (**bez pierwszego z …**), które są przetwarzane przez wywołanie rekurencyjne:

Snap*!* używa dwóch różnych wewnętrznych reprezentacji list, jednej (tablicy dynamicznej) do programowania imperatywnego i drugiej (listy linkowanej) do programowania funkcyjnego. Każda reprezentacja powoduje, że odpowiednie wbudowane bloki list (odpowiednio komendy lub funkcje) są najbardziej wydajne. Możliwe jest mieszanie stylów w tym samym programie, ale jeśli ta sama lista jest używana w obie strony, program będzie działał wolniej, ponieważ wielokrotnie konwertuje z jednej reprezentacji na drugą (**element () z [ ]** nie zmienia reprezentacji). Nie musisz znać szczegółów wewnętrznej reprezentacji, ale warto używać każdej listy w spójny sposób.

## D Operacje wyższego rzędu na listach i obwiednie

Jest prostszy sposób wybrania liczb parzystych z listy liczb:

(Jeśli nie masz bloku **zachowaj** u dołu palety Dane, kliknij przycisk Plik na pasku narzędzi i wybierz „**Importuj narzędzia...**")

Blok **zachowaj** (**keep**) przyjmuje predykat jako pierwsze wejście, a listę jako drugie wejście. Daje w wyniku listę zawierającą te elementy listy wejściowej, dla których predykat ma wartość **prawda**. Zwróć uwagę na dwie rzeczy dotyczące pola wejściowego dla predykatów: Po pierwsze, wokół niego jest szara obwiednia. Po drugie, blok **modulo** ma puste pole wejścia. **Zachowaj** umieszcza każdy element listy wejściowej, po jednym na raz, w tym pustym wejściu przed obliczeniem wartości predykatu. (Puste wejście powinno przypominać Ci o notacji „pudełkowej" dla zmiennych jeszcze ze szkoły podstawowej: ☐ + 3 = 7). Szara obwiednia jest częścią bloku **zachowaj**, w takiej postaci pojawia się on w palecie:

Obwiednia oznacza, że w tym miejscu jest blok (w tym przypadku blok predykatu, ponieważ wnętrze obwiedni jest sześciokątem), a nie wartość przekazywana przez ten blok. Oto różnica:

Obliczanie bloku = bez obwiedni daje wynik **prawda** lub **fałsz**; obliczanie bloku z obwiednią daje sam blok. Pozwala to blokowi **zachowaj** obliczać wartość predykatu = wielokrotnie, jeden raz dla każdego elementu listy. Blok, który przyjmuje inny blok jako wejście, nazywany jest blokiem *wyższego rzędu* (procedurą wyższego rzędu lub funkcją wyższego rzędu).

Snap*!* dysponuje trzema blokami wyższego rzędu do operacji na listach:

Przyjrzeliśmy się już zachowaj. **Mapuj** (**map**) pobiera blok operacji i listę jako dane wejściowe. Daje w wyniku nową listę, w której każda pozycja ma wartość przekazywaną przez blok operacji, która została zastosowana do jednego elementu z listy wejściowej. To jest kupa słów, ale przykład powinien sprawę wyjaśnić:

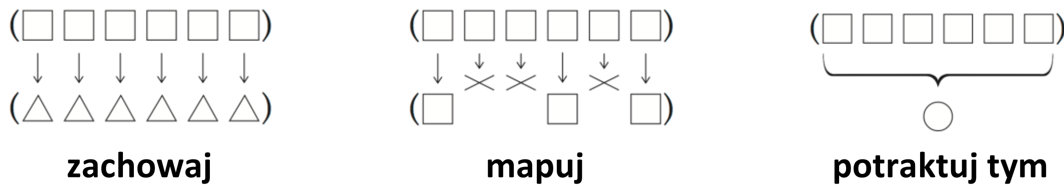
Nawiasem mówiąc, używaliśmy przykładów arytmetycznych, ale elementy listy mogą być dowolnego typu i można użyć dowolnej operacji. Obliczmy liczbę mnogą kilku słów:

Te przykłady używają małych list, aby zmieściły się na tej stronie, ale bloki wyższego rzędu działają dla listy dowolnych rozmiarów.

Trzeci blok wyższego rzędu, **potraktuj tym** (**combine**), oblicza pojedynczy wynik dla wszystkich elementów listy, używając operacji z dwoma polami wejścia jako pierwszego wkładu. W praktyce istnieje tylko kilka bloków, z których będziesz korzystać stosując **potraktuj tym**:

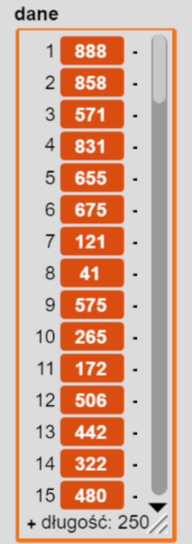
Bloki te będą obliczać sumę elementów listy, ich iloczyn, połączą je w jedno słowo, połączą je w zdanie (ze spacjami między elementami), sprawdzą, czy wszystkie pozycje listy Boolowskiej są prawdziwe, lub sprawdzą, czy któryś z elementów jest prawdziwy.

Dlaczego + ale nie **-**? Sensowne jest łączenie elementów listy za pomocą funkcji asocjacyjnej: takiej, która nie dba o to, w jakiej kolejności elementy są łączone (od lewej do prawej, czy od prawej do lewej). (2+3)+4 = 2+(3+4), ale (2−3)−4 ≠ 2−(3−4).



## E Widok tabeli a widok listy

Wspomnieliśmy wcześniej, że istnieją dwa sposoby wizualnego przedstawiania list. W przypadku list jednowymiarowych (list, których elementy same nie są listami) różnice wizualne są niewielkie:



W przypadku list jednowymiarowych to nie wygląd jest tak naprawdę ważny. Liczy się to, że widok listy umożliwia bardzo wszechstronną, bezpośrednią manipulację listą na jej obrazku: możesz edytować poszczególne elementy, możesz usuwać elementy, klikając małe przyciski obok każdego elementu, a na końcu możesz dodawać nowe elementy, klikając mały znak plus w lewym dolnym rogu. (Trudno zauważyć, że przyciski usuwania przedmiotów mają znaki minus). Nawet jeśli masz kilka obrazów dla tej samej listy, po zmianie czegokolwiek wszystkie zostaną zaktualizowane. Z drugiej strony za tą wszechstronność płaci się obniżeniem wydajności; podgląd listy dla długiej listy byłby zbyt powolny. W częściowym podglądzie widok listy może zawierać tylko 100 elementów naraz; strzałka skierowana w dół otwiera menu, w którym możesz wybrać, które 100 elementów wyświetlić.

W przeciwieństwie do tego, widok tabeli może pomieścić tysiące elementów i nadal je wydajnie przewijać ponieważ nie pozwala na bezpośrednią edycję. Widok tabeli ma bardziej płaską grafikę dla elementów, aby przypomnieć, że nie można ich kliknąć, aby edytować wartości.

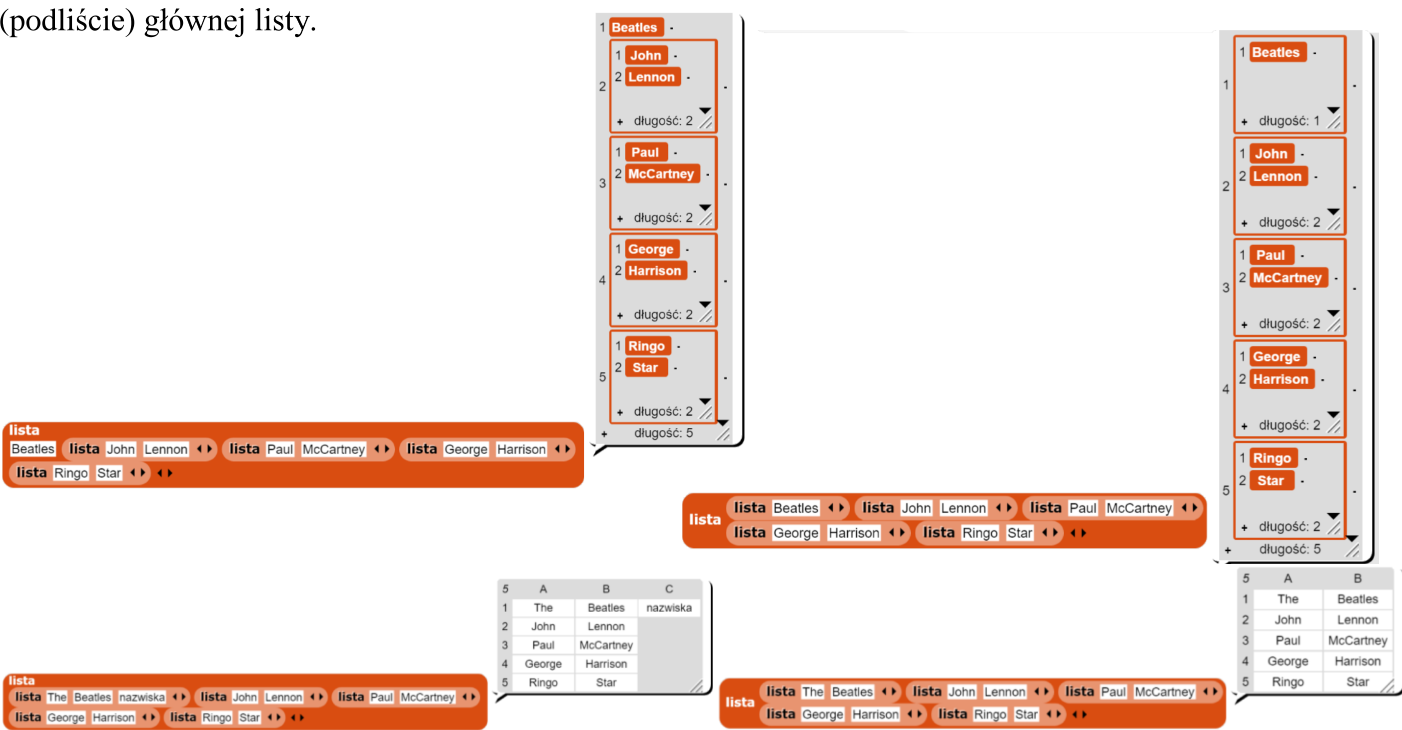
Kliknięcie widoku listy (w dowolnej postaci) prawym przyciskiem myszy umożliwia przejście do drugiej postaci. Menu pod prawym przyciskiem myszy ma również pozycję **otwórz w nowym oknie dialogowym**, która otwiera widok tabeli poza sceną, ponieważ widoki list mogą zająć dużo miejsca na scenie, a to może utrudniać zobaczenie, co faktycznie robi twój program. Gdy jest otwarte okno dialogowe poza sceną, możesz zamknąć okno widoku na scenie. W oknie dialogowym poza sceną jest przycisk OK, powodujący jego zamknięcie, jeśli zechcesz. Możesz też kliknąć go prawym przyciskiem myszy, aby otworzyć kolejne okno widoku, co jest przydatne, jeśli chcesz obejrzeć dwie części listy naraz, przewijając ją w drugim oknie w inne miejsce.

Widok tabeli jest domyślny, jeśli lista zawiera więcej niż 100 elementów, lub jeśli pierwszą pozycją listy jest lista, w którym to przypadku tworzy ona zupełnie odmienny dwuwymiarowy obraz:

W tym formacie kolumna czerwonych elementów została zastąpiona przez tabelę, która przypomina arkusz kalkulacyjny. W przypadku krótkich, szerokich list ten widok sprawia, że zawartość listy jest bardzo przejrzysta. Pionowy widok, z dużą ilością miejsca zajmowanego przez „maszynerię” na dole każdej z podlist, utrudniłby wyświetlanie całego tekstu na raz. (Kosztem pedagogicznym jest to, że struktura nie jest już wyraźna, nie możemy powiedzieć, po prostu patrząc, że jest to lista list wierszy, a nie lista list kolumn lub prymitywny dwuwymiarowy typ tablicy. Ale możesz wybrać widok listy, aby zobaczyć strukturę.)

Poza prostymi przypadkami, w których każdy element głównej listy jest listą o tej samej długości, należy pamiętać, że postać widoku tabeli musi spełniać dwa cele, nie zawsze ze sobą zgodne: (1) przekonujące wizualnie wyświetlanie dwuwymiarowych tablic oraz (2) bardzo wydajne generowanie widoku, dzięki czemu Snap*!* może obsługiwać bardzo duże listy, ponieważ „duże dane” są ważnym tematem badań. Aby idealnie osiągnąć pierwszy cel w przypadku „nierównych prawych” tablic, w których podlisty mogą mieć różne długości, Snap*!* skanowałby całą listę, aby znaleźć maksymalną szerokość przed wyświetleniem czegokolwiek, ale to naruszyłoby drugi cel.

Snap*!* używa najprostszego możliwego kompromisu między tymi dwoma celami: sprawdza tylko pierwszy element listy, aby zdecydować o formacie. Jeśli pierwszy element nie jest listą lub jest listą jednoelementową, a całkowita długość listy nie jest większa niż 100, używany jest widok listy. Jeśli pierwszy element jest listą, używany jest widok tabeli, a liczba kolumn w tabeli jest równa liczbie pozycji w pierwszym elemencie (podliście) głównej listy.



Pierwsze dwa powyższe przykłady pokazują, że lista, której pierwszym elementem jest nie-lista lub lista o długości co najwyżej jeden domyślnie jest wyświetlana w widoku listy. Dwa ostatnie przykłady pokazują, że jeśli pierwszy element jest listą o długości większej niż jeden, używany jest widok tabeli, z liczbą kolumn określoną przez liczbę pozycji na pierwszej podliście. W ostatnim przykładzie zwróć uwagę na poszarpany prawy margines pozycji 2 i 3. Oznacza to, że pierwsza podlista ma więcej pozycji niż pozostałe. Aby to zobaczyć, przełącz się do widoku listy. W trzecim przykładzie powyżej puste komórki są wyświetlane w całości w kolorze tła, co kładzie nacisk na elementy, a nie na zmienną szerokości tabeli. W menu Ustawienia możesz zaznaczyć "Linie tabeli", aby przyciemnić granice między komórkami: (*nie widzę tego w programie* WtK).

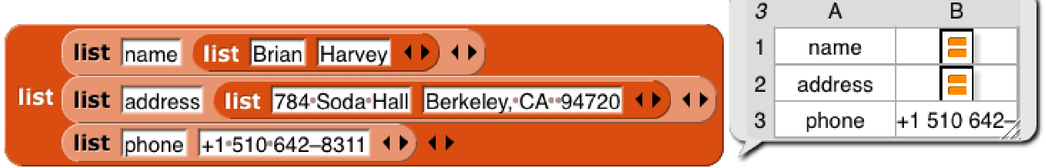
Widoki tabel otwierają się ze standardowymi wartościami dla szerokości i wysokości komórki, niezależnie od rzeczywistych danych. Możesz zmienić te wartości, przeciągając litery kolumn lub numery wierszy. Każda kolumna ma swoją własną szerokość, ale zmiana wysokości wiersza powoduje zmianę wysokości wszystkich wierszy. (Różnica ta wynika nie z różnego znaczenia wierszy i kolumn, ale z tego, że stała wysokość wiersza powoduje bardziej wydajne przewijanie dużej listy.) Przeciąganie etykiety kolumny z klawiszem Shift powoduje zmianę szerokości wszystkich kolumn.

Jeśli sprawdzisz przeciąganie omówione w poprzednim paragrafie, to zauważysz, że po najechaniu myszą na literę kolumny zmienia się ona w liczbę. Etykietowanie wierszy i kolumn w inny sposób sprawia, że odwołania do komórek, takie jak „komórka 4B”, są jednoznaczne; nie musisz umawiać się, czy najpierw powiedzieć wiersz, czy kolumnę. („Komórka B4” to to samo co „komórka 4B”). Z drugiej strony, aby uzyskać w swoim programie wartość z kolumny B, musisz wziąć **element 2**, a nie element B. Warto więc móc znaleźć numer kolumny, umieszczając kursor nad jej literą.

Każda wartość, która może pojawić się w programie, może być wyświetlana w komórce tabeli:

Ten zrzut pokazuje, że standardowe rozmiary komórek mogą być niewystarczające dla obrazów o dużej wielkości. Po powiększeniu całego dymku wyświetlania, a następnie drugiej kolumny i wszystkich wierszy, możemy sprawić, że wynik będzie dopasowany.

Ale robimy wyjątek dla przypadków, w których wartość w komórce jest listą (tak, że cała tabela jest trójwymiarowa). Ponieważ listy mogą być wizualnie bardzo duże, nie próbujemy pokazać całej wartości w komórce:




Nawet jeśli powiększysz rozmiar komórek, Snap*!* nie będzie wyświetlać podlist w podglądzie tabeli. Istnieją dwa sposoby wyświetlania tych wewnętrznych podlist: Możesz przejść do widoku listy lub kliknąć dwukrotnie ikonę listy w tabeli, aby otworzyć okno dialogowe pokazujące tylko tę podrzędną listę w widoku tabeli.

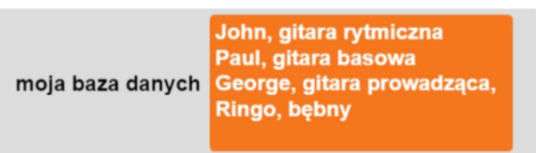
Ostatni szczegół: jeśli pierwszym elementem listy jest lista (jest więc używany widok tabeli), ale kolejny element nie jest listą, to ten element będzie wyświetlany na czerwonym tle, jak element pojedynczej listy kolumn:

Zatem, w szczególności, jeśli tylko pierwszy element jest listą, tabela będzie wyglądać prawie tak, jak tabela z jedną kolumną.

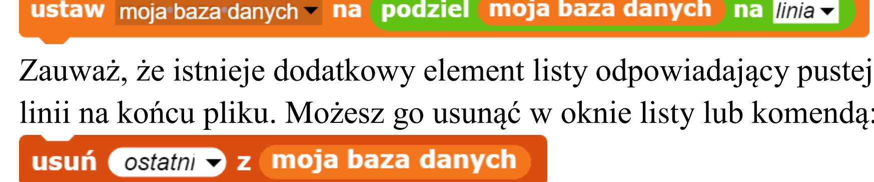
## CSV wartości oddzielone przecinkami

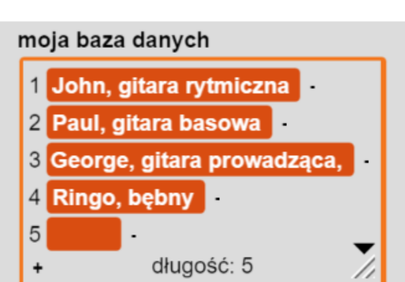
Programy arkusza kalkulacyjnego i bazy danych zazwyczaj oferują opcję eksportu swoich danych jako listy CSV (wartości rozdzielane przecinkami). Możesz importować te pliki do Snap*!* i przekształcać je w tabele (listy list).

1. Utwórz zmienną z plakietką na scenie. moja baza danych 0
2. Kliknij na plakietce prawym przyciskiem myszy i wybierz opcję „**importuj**". Wybierz plik z danymi CSV.




3. Podziel tekst na linie.




Zauważ, że istnieje dodatkowy element listy odpowiadający pustej linii na końcu pliku. Możesz go usunąć w oknie listy lub komendą:

4. Teraz podziel na pola każdy wiersz wykorzystując przecinki:

ustaw moja baza danych na mapuj (podziel [ ] na csv) na moja baza danych

moja baza danych

| 4 | A | B |
|---|---|---|
| 1 | John | gitara rytmiczna |
| 2 | Paul | gitara basowa |
| 3 | George | gitara prowadząca |
| 4 | Ringo | bębny |

(Opcja **csv** w bloku **podziel** jest podobna do dzielenia na przecinkach, podziel [ ] na , z wyjątkiem tego, że rozpoznaje znaki i łańcuchy w cudzysłowie).

Oczywiście możesz połączyć te kroki składając funkcje:

ustaw moja baza danych na mapuj (podziel [ ] na csv) na podziel moja baza danych na linia

# V Typy pól wejść (parametrów)

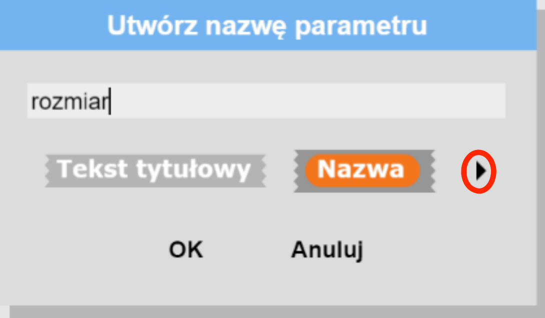
## A Notacja typów w Scratchu

Pola wejściowe bloków w Scratchu mogą być dwóch typów: typu tekst-lub-liczba i typu liczba. Pierwsze z nich ma kształt prostokątny, a drugie okrągły: litera 1 z świecie. Trzeci typ Scratch, Boolowski (prawda/fałsz) może być użyty w niektórych blokach Kontroli z sześciokątnymi polami wejścia.

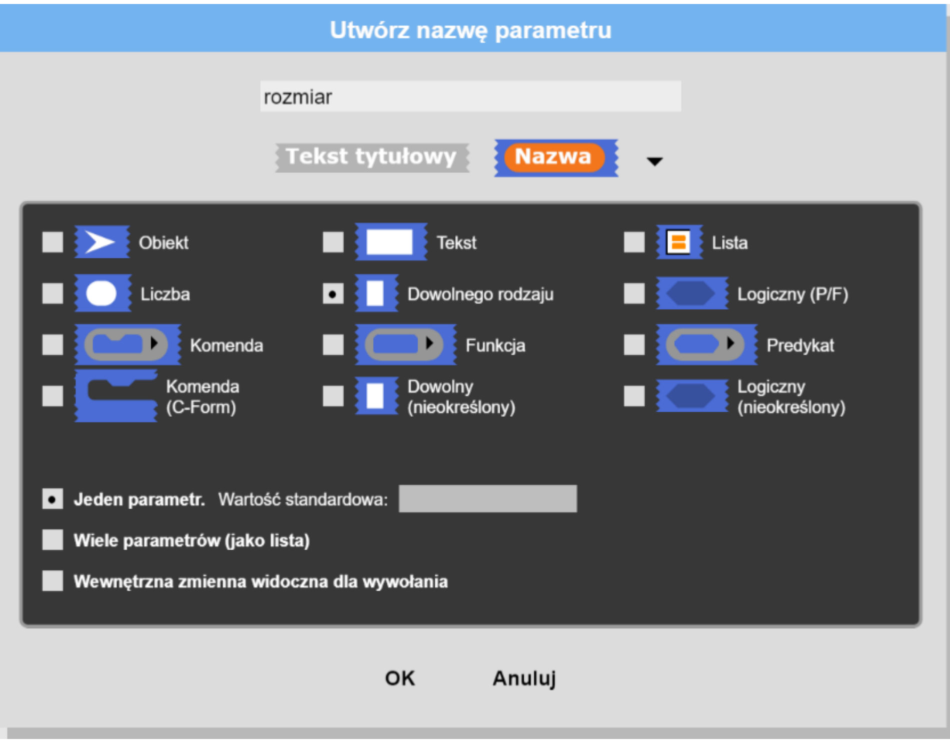
Snap*!* ma rozszerzoną kolekcję typów, obejmującą typy Procedur, List i Obiektów. Zauważ, że z wyjątkiem typów Procedur, wszystkie kształty typów wejściowych są tylko przypomnieniem użytkownikowi tego, czego oczekuje blok; nie są narzucane przez język.

## B Okno dialogowe nazwy parametru w Snap*!*

W oknie dialogowym Utwórz nazwę parametru w Edytorze bloków znajduje się strzałka na prawo od opcji „Nazwa":



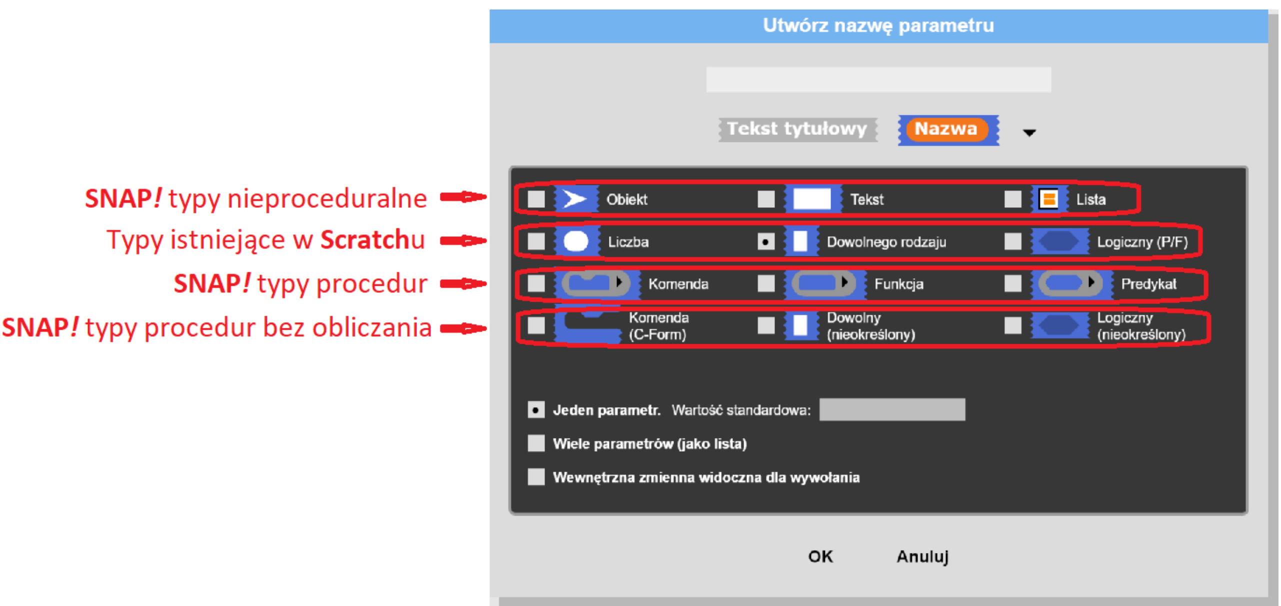

Kliknięcie tej strzałki powoduje otwarcie „długiego" okna dialogowego Utwórz nazwę parametru:




Dostępnych jest dwanaście typów kształtów pól wejściowych oraz trzy wykluczające się wzajemnie kategorie, jako dodatek do podstawowego wyboru między **Tekstem tytułowym** i **Nazwą**. Domyślnym typem, który

otrzymasz, jeśli nie wybierzesz niczego innego, jest „dowolnego rodzaju", co oznacza, że to pole wejściowe ma akceptować dowolną wartość dowolnego typu. Jeśli parametr **rozmiar** w twoim bloku powinien mieć kształt owalny, a nie prostokątny, kliknij „**Liczba**".

Rozmieszczenie typów wejść jest usystematyzowane. Jak pokazują obrazki na tej i następnej stronie, każdy wiersz typów jest kategorią, ale też części każdej kolumny tworzą kategorię. Zrozumienie tej aranżacji ułatwi znalezienie odpowiedniego typu parametru.




Drugi wiersz typów parametrów zawiera te znane ze Scratcha: liczba, dowolny (liczba, tekst) i logiczny. (Powód, dla którego są one w drugim rzędzie, a nie pierwszym, stanie się jasny, gdy spojrzymy na układ kolumn). Pierwszy rząd zawiera nowe typy w Snap*!* inne niż procedury: obiekt, tekst i lista. Ostatnie dwa wiersze to typy związane z procedurami, które omówiono dokładniej poniżej.

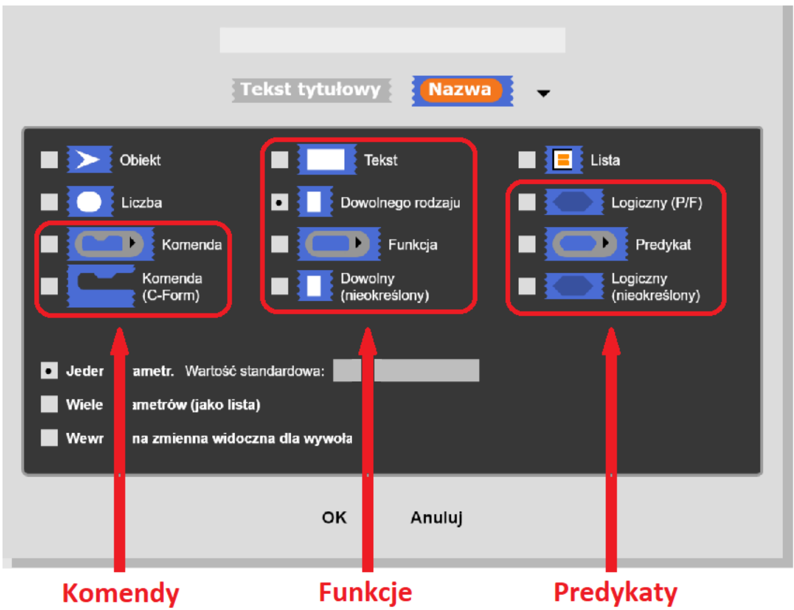
Typ lista jest używany w listach pierwszej klasy, omówionych w rozdziale IV powyżej. Czerwone prostokąty w polu wejściowym mają przypominać wygląd list, tak jak Snap*!* wyświetla je na scenie: każdy element w czerwonym prostokącie.
Typ obiekt dotyczy duszków, kostiumów, dźwięków i podobnych typów danych.
Typ tekst jest tak naprawdę tylko wariantem typu dowolny, używa on kształtu sugerującego wejście tekstowe.[9]

## Typy procedur

Chociaż typy procedur są omówione bardziej szczegółowo później, są one kluczem do zrozumienia rozmieszczenia kolumn w oknie tworzenia nazwy parametru. Podobnie jak Scratch, Snap*!* ma trzy kształty bloków: kawałek układanki (cegiełka) dla bloków poleceń, owal dla funkcji i sześciokątny dla predykatów. (Predykat to funkcja, która zawsze daje wynik prawda lub fałsz). W Snap*!* te bloki są danymi pierwszej klasy; dane wejściowe do bloku mogą być typu komendy, typu funkcji lub predykatu. Każdy z tych typów znajduje się bezpośrednio pod typem wartości, który przekazuje ten blok, z wyjątkiem komend, które w ogóle nie przekazują wartości. W związku z tym owalne funkcje są powiązane z typem dowolnego rodzaju, podczas gdy heksagonalne predykaty są powiązane z typem logicznym (prawda lub fałsz).

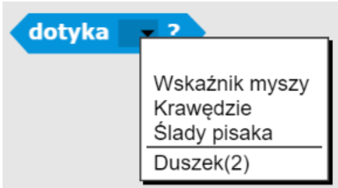
[9] W Scratchu każdy blok, który przyjmuje dane wejściowe typu tekst ma domyślną wartość, która powoduje, że prostokąty dla tekstu mają większą szerokość niż wysokość. Bloki, które nie dotyczą konkretnie tekstu, są typu liczba lub nie mają wartości domyślnej, mają prostokąty o większej wysokości niż szerokości. Na początku myśleliśmy, że tekst jest oddzielnym typem, który zawsze ma szerokie pole wejściowe; okazuje się, że to nie jest prawdą (usuń domyślny tekst i prostokąt zwęża się), ale pomyśleliśmy, że to dobry pomysł, więc pozwalamy na pola w kształcie tekstu nawet dla pustych pól wejściowych. (Z tego powodu typ tekst pojawia się tuż powyżej typu dowolny w oknie dialogowym tworzenia nazwy parametru.).

Typy procedur bez obliczania w czwartym rzędzie wyjaśniono w rozdziale VI E poniżej. W jednym zdaniu: łączą *znaczenie* typów procedur z *wyglądem* przekazywanych typów wartości o dwa rzędy wyżej. (Oczywiście nie jest to właściwe dla typu wejścia bloku w kształcie litery C, ponieważ komendy nie przekazują wartości, ale później zobaczysz, że to oddaje sedno sprawy).

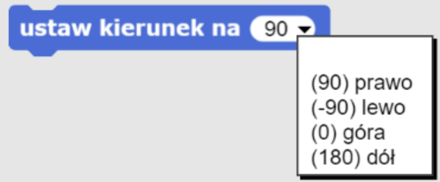



## Rozwijane pola wejścia

Niektóre bloki pierwotne mają wejścia z rozwijanym menu, albo tylko do odczytu, jak wejście do bloku **dotyka**:



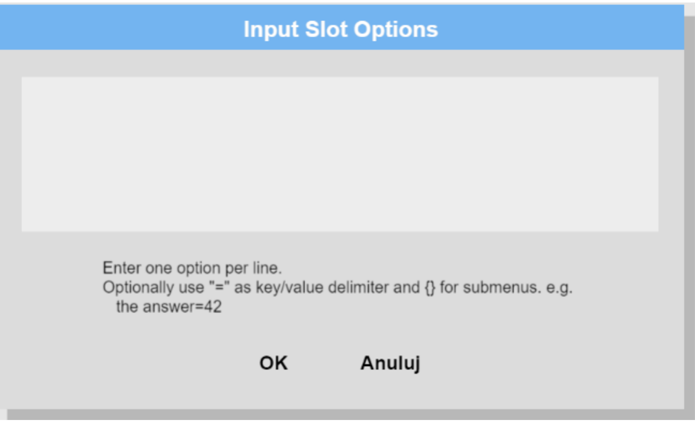

(sygnalizowane przez pole wejściowe w kolorze takim samym jak bryła bloku (niebieski, w tym przypadku), albo takim, w którym można coś wpisać, jak wejście do bloku **ustaw kierunek**:




(sygnalizowane przez białe pole wejściowe), co oznacza, że użytkownik może wprowadzić dowolną wartość zamiast używać menu rozwijanego.

Niestandardowe bloki mogą również mieć takie dane wejściowe. To jest funkcja eksperymentalna, a interfejs użytkownika prawdopodobnie się zmieni! Aby można było wprowadzić dane z rozwijanego menu, otwórz długie okno dialogowe wprowadzania nazwy parametru i kliknij z klawiszem ctrl / kliknij prawym przyciskiem myszy w ciemnoszarym obszarze. Zobaczysz menu:

Kliknij pole wyboru (read-only), jeśli chcesz mieć rozwijane menu tylko do odczytu. Kliknij ponownie z klawiszem ctrl / kliknij prawym przyciskiem myszy i wybierz **options ...**, aby uzyskać to okno dialogowe:




Na razie możesz mieć tylko ustalone opcje, a nie na przykład „wszystkie duszki”, nie mówiąc już o „wszystkie duszki, które znajdują się teraz po lewej połowie sceny”.

Każda linia w polu tekstowym reprezentuje jedną pozycję menu. Jeśli linia nie zawiera żadnego ze znaków

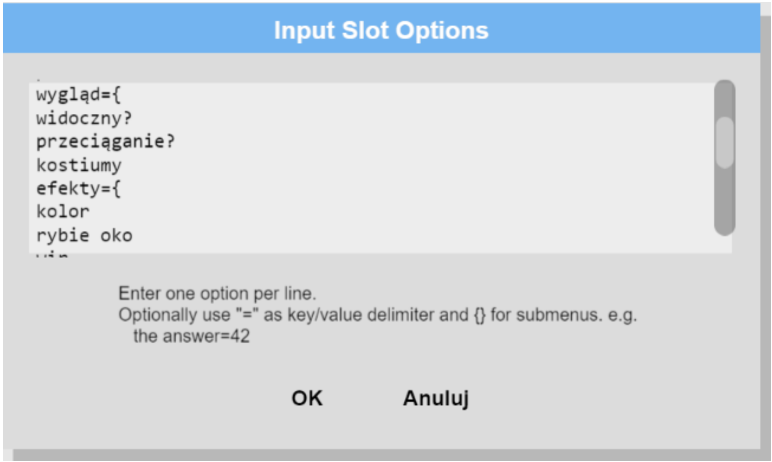
**=~{}**

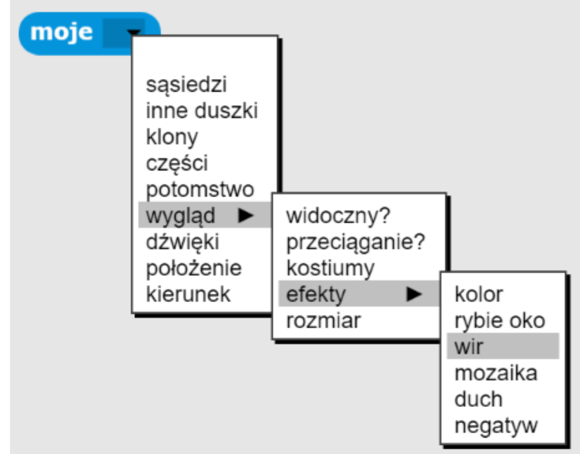
wtedy tekst jest zarówno tym, co widać w menu, jak i wartością wejścia, jeśli ten wpis został wybrany.

Jeśli wiersz zawiera znak równości =, tekst z lewej strony znaku równości jest wyświetlany w menu, a tekst po prawej pojawia się w polu wejścia, jeśli wybrano ten wpis, i jest również wartością wejściową widzianą przez procedurę.

Jeśli linia składa się z tyldy ~, oznacza ona w menu oddzielacz (poziomą linię), służący do podziału długich menu na widoczne kategorie. W tej linii nie powinno być nic więcej. Tego separatora nie można wybrać, więc nie ma odpowiadającej mu wartości wejściowej.

Jeśli linia kończy się dwoma znakami: równości i otwierającym nawiasem klamrowym **={**, wówczas reprezentuje podmenu. Tekst przed znakiem równości jest nazwą podmenu i będzie wyświetlany w menu ze strzałką ► na końcu wiersza. Ta linia nie jest klikalna, ale przemieszczanie nad nią kursora myszy powoduje wyświetlenie podmenu obok pierwotnego menu. Linia zawierająca nawias zamykający **}** kończy podmenu; w tej linii nie powinno być nic innego. Podmenu można zagnieżdżać na dowolną głębokość.



### Warianty wejścia

Przechodzimy teraz do trzech wzajemnie wykluczających się opcji, które znajdują się poniżej tablicy typów.

Opcja „**Jeden parametr**": w Scratchu wszystkie wejścia należą do tej kategorii. W bloku znajduje się jedno pole wejściowe, które pojawia się w jego kształcie. Jeśli jedno pole wejścia ma typ dowolny, liczbowy, tekstowy lub logiczny, wówczas można określić wartość domyślną, która będzie wyświetlana w tym polu w palecie, podobnie jak „10" w bloku **przesuń o** (10) **kroków**. W prototypowym bloku u góry skryptu w edytorze bloku pole wejściowe o nazwie "rozmiar" i wartości domyślnej 10 wygląda następująco:

+ kwadrat + rozmiar=10 +

Opcja „**Wiele parametrów**": Wprowadzony wcześniej blok listy akceptuje dowolną liczbę wejść określających pozycje nowej listy. Aby na to pozwolić, Snap*!* wprowadza notację ze strzałkami ( ◀ ▶ ), które poszerzają lub zwężają blok, dodając i usuwając pola wejściowe. Niestandardowe bloki tworzone przez użytkownika Snap*!* mogą też z tego korzystać. Jeśli wybierzesz przycisk „**Wiele parametrów**", za polem wejściowym pojawią się strzałki. Można użyć więcej lub mniej pól (nawet zero). Po uruchomieniu bloku wszystkie wartości we wszystkich polach dla tej nazwy wejścia są gromadzone na liście, a wartość wejścia widoczna w skrypcie jest listą tych wartości:

+ wejścia + wszystkie... +
wynik wszystkie
→ wejścia Rapete Papete Bum ◀▶
1 Rapete
2 Papete
3 Bum
+ długość: 3

Trzy kropki (...) w pomarańczowym polu nazwy parametru w prototypie bloku wskazują możliwość użycia wielu parametrów.

Trzecia kategoria, „Wewnętrzna zmienna widoczna dla wywołania", w rzeczywistości nie jest w ogóle wejściem, ale raczej rodzajem wyjścia z bloku do jego użytkownika. Wygląda raczej na pomarańczową zmienną owalną w bloku, niż na pole wejściowe. Oto przykład; strzałka w górę (↑) w prototypie wskazuje tego rodzaju nazwę zmiennej wewnętrznej:

+ dla + i ↑ + = + start # = 1 + do + end # = 10 + action λ +
→ dla i = 1 do 10

Zmienna **i** (w bloku po prawej stronie powyżej) może być przeciągnięta z bloku **dla** do bloków używanych w polu poleceń w kształcie litery C. Ponadto, klikając pomarańczowe **i**, użytkownik może zmienić nazwę zmiennej, jak widać w skrypcie wywołującym (chociaż nazwa nie zmieniła się w definicji bloku). Ten rodzaj zmiennej nazywa się po angielsku krótko upvar, ponieważ jest przekazywany w górę z niestandardowego bloku do skryptu, który go używa.

### Oznaczenia w prototypach bloków

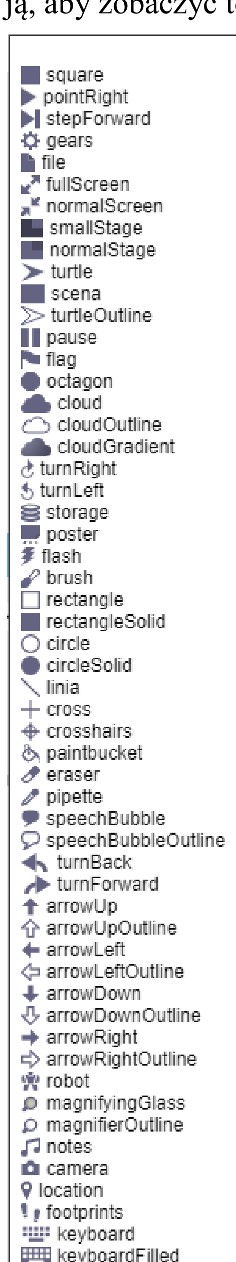
Przedstawiliśmy trzy notacje, które mogą pojawić się w polu wejściowym w prototypie, aby przypomnieć Ci, jakiego rodzaju to jest wejście. Oto pełna lista takich sposobów notacji:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| = | wartość domyślna | … | wiele wejść | ↑ | upvar | # | liczba |
| λ | typy procedur | ⋮ | lista | ? | logiczna | ⋗ | obiekt |

## Tekst tytułowy i symbole

Niektóre pierwotne bloki zawierają symbole jako część nazwy bloku:
Bloki niestandardowe (użytkownika) mogą również używać symboli. W edytorze bloków kliknij znak plus w prototypie bloku w miejscu, w którym chcesz wstawić symbol. Następnie kliknij obrazek Tekst tytułowy poniżej pola tekstowego, w które należy wpisać nazwę parametru. Okno zmieni wygląd na następujący:

Ważną rzeczą, którą należy zauważyć, jest strzałka, która pojawiła się w prawym końcu pola tekstowego. Kliknij ją, aby zobaczyć to menu:




Wybierz jeden z symboli. Rezultat będzie wyglądał trochę brzydko w prototypie bloku:

ale sam blok będzie miał wybrany symbol:

Dostępne są symbole, które mniej lub bardziej przypominają ikony wykorzystywane w Snap*!*.

Ale ja chciałbym mieć strzałkę większą i żółtą, więc edytuję jej nazwę:

Dopisany tekst mówi, że strzałka ma być 1.5 raza większa od liter tekstu w bloku i mieć kolor z wartościami czerwony-zielony-niebieski (RGB) 255-255-150 (każda pomiędzy 0 i 255). Oto rezultat:

Kontrolki rozmiaru i koloru mogą być również używane ze zwykłym tekstem:

**$ bum-8-255-120-0** zrobi bardzo duże pomarańczowe słowo „bum”.

# VI Procedury jako dane

## A Wywołaj i uruchom

W jednym z przykładów powyżej pokazującym blok **dla** pole wejściowe o nazwie **akcja** zostało zadeklarowane jako typ „Komenda (C-form, C-kształtna)”; dlatego gotowy blok ma kształt litery "C". Ale w jaki sposób blok faktycznie mówi Snapowi żeby wykonywać polecenia wewnątrz kształtu C? Oto prosta wersja skryptu bloku:

+ dla + i ↑ + = + start = 1 + do + koniec = 10 + akcja λ +
ustaw i na start
powtarzaj aż i > koniec
uruchom akcja
zmień i o 1

Jest to uproszczona wersja, ponieważ zakłada, bez sprawdzania, że wartość końcowa jest większa niż wartość początkowa; jeśli nie, blok powinien (w zależności od celów projektanta) albo nie działać w ogóle, albo zmienić zmienną o -1 zamiast o 1. (Blok **dla** w bibliotece narzędzi Snap*!*, działa dla rosnących lub malejących wartości; można odczytać jego skrypt klikając go prawym przyciskiem myszy lub klikając z wciśniętym klawiszem ctrl i wybierając opcję Edytuj).

Istotną częścią tego skryptu jest znajdujący się prawie na końcu blok **uruchom**. Jest to pierwotny blok poleceń Snap*!*, który przyjmuje jako wartość wejściową komendę (skrypt) i wykonuje instrukcje (bloki skryptu). (W tym przykładzie wartością wejściową akcja jest skrypt, który użytkownik umieścił wewnątrz kształtu C bloku). Istnieje podobny blok funkcji – **wywołaj** wywołujący bloki funkcji lub predykatu. Bloki **wywołaj** i **uruchom** są sercem funkcji pierwszej klasy Snap*!*; umożliwiają one wykorzystywanie skryptów i bloków jako danych – w tym przykładzie jako danych wejściowych do bloku – i ostatecznie przetwarzanych pod kontrolą programu użytkownika.

Oto inny przykład, tym razem przy użyciu danych typu funkcja w uproszczonym bloku **mapuj** (patrz strona 32):

+ mapuj + funkcja λ + na + dane ⋮ +
jeżeli puste? dane to
wynik lista
wynik
wstaw wywołaj funkcja z parametrami element 1 z dane przed
mapuj funkcja na bez pierwszego z dane

mapuj ( × 10) na lista 7 8 1

| | |
|---|---|
| 1 | 70 |
| 2 | 80 |
| 3 | 10 |

\+ długość: 3

W tym miejscu wywołujemy funkcję „pomnóż przez 10” trzy razy, raz dla każdej pozycji listy podanej jako jej dane wejściowe i zbieramy wyniki na liście. (Przekazywana w wyniku lista będzie zawsze miała taką samą długość jak lista wejściowa). Zauważ, że blok mnożenia ma dwa wejścia, ale tutaj wstawiliśmy konkretną wartość do jednego z nich (10), więc blok **wywołaj** wie, że trzeba użyć danych z podanej listy w celu wypełnienia innego (pustego) pola wejściowego w bloku mnożenia. W definicji **mapuj** funkcja wejściowa jest zadeklarowana jako typ **Funkcja**, a dane są typu **Lista**.

### Wywołaj/uruchom z polami wejścia

Blok **wywołaj** (podobnie jak blok **uruchom**) ma na prawym końcu strzałkę; kliknięcie jej dodaje wyrażenie „z parametrami”, a następnie pole, do którego można wstawić dane wejściowe:

wywołaj ( ) z parametrami

Starzaka w lewo może być użyta do usunięcia ostatniego pola wejściowego, zniknie również napis „z parametrami”. Strzałkę w prawo można kliknąć tyle razy, ile potrzeba, by uzyskać liczbę wejść wymaganych przez blok funkcji.

Jeśli liczba wejść dla bloku **wywołaj** (nie licząc pierwszego wejścia dla funkcji) jest taka sama jak liczba pustych pól wejściowych, to puste pola są wypełniane od lewej do prawej podanymi wartościami wejściowymi. Jeśli **wywołaj** ma dokładnie jedno wejście, to każde puste pole wejściowe wywoływanego bloku jest wypełniane tą samą wartością:

Jeśli podana liczba wejść nie wynosi jeden i nie jest liczbą pustych pól, nie ma automatycznego zapełniania pustych pól. (Zamiast tego musisz użyć parametrów w obwiedni, jak omówiono w podrozdziale C poniżej.)

Jeszcze ważniejszą rzeczą, na którą należy zwrócić uwagę w tych przykładach, jest obwiednia wokół gniazd wejściowych typu funkcja w blokach **wywołaj** i **mapuj** powyżej. Ta notacja wskazuje, że wejściem jest *sam blok*, a nie liczba lub inna wartość, którą blok daje w wyniku po wywołaniu. Jeśli chcesz użyć samego bloku w polu wejściowym typu innego niż funkcja (np. typ dowolny), możesz go jawnie umieścić w obwiedni, znajdującej się u góry palety Wyrażenia.

Na skróty, jeśli klikniesz na bloku (takim jak blok + w tym przykładzie) prawym przyciskiem myszy lub klikniesz z klawiszem ctrl, jedną z możliwości w pojawiającym się menu będzie „**obwiednia**” lub

„**bez obwiedni**”. Obwiednia wskazująca typ wejścia funkcyjny lub predykatowy, jest w zasadzie tym samym pomysłem dla funkcji, co pole wejściowe w kształcie litery C, które już znasz; w polu w kształcie litery C, umieszczasz skrypt, który staje się wejściem do bloku w kształcie litery C.

### Zmienne w obwiedniach

Zauważ, że blok **uruchom** w definicji bloku **dla** (strona 43) nie ma obwiedni wokół jego zmiennej wejściowej **akcja**. Kiedy przeciągasz zmienną do pola wejściowego z obwiednią, zazwyczaj wolisz używać wartości zmiennej, jako bloku lub skryptu, który chcesz uruchomić lub wywołać, niż samą pomarańczową zmienną. Więc Snap*!* w tym przypadku automatycznie usuwa obwiednię. Jeśli kiedykolwiek chcesz użyć zmiennej jako samego bloku, a nie wartości zmiennej, jako wejście typu procedura, możesz przeciągnąć zmienną do pola wejściowego, następnie kliknąć prawym przyciskiem myszy lub przytrzymując klawisz ctrl, a następnie wybrać opcję „**obwiednia**” z menu, które się pojawi. (Podobnie, jeśli kiedykolwiek chcesz wywołać funkcję, która przekaże blok jako dane wejściowe, możesz wybrać z menu „**bez obwiedni**”. Ale prawie zawsze Snap*!* zrobi bez pomocy to, co masz na myśli).

## B Pisanie procedur wyższego rzędu

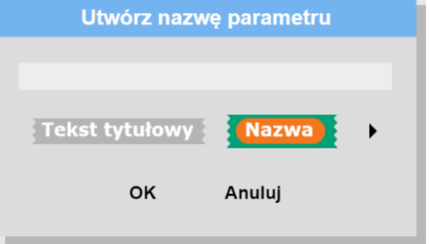
*Procedura wyższego rzędu* to procedura, która przyjmuje inną procedurę jako dane wejściowe lub daje w wyniku procedurę. W tym dokumencie słowo „procedura” obejmuje skrypty, pojedyncze bloki i zagnieżdżone funkcje. (O ile nie zaznaczono inaczej, „funkcja” oznacza też predykat.). Gdy słowo w zdaniu jest pisane wielką literą, oznacza to konkretnie bloki o kształcie owalnym, więc „zagnieżdżone funkcje” obejmują predykaty, ale „dane wejściowe typu Funkcja” nie). Chociaż wejście typu Dowolnego rodzaju (uzyskane w przypadku korzystania z małego okna dialogowego) akceptuje jako dane procedury, nie wstawia automatycznie obwiedni, jak opisano powyżej. Tak więc deklaracja danych typu Procedura sprawia, że korzystanie z niestandardowego bloku wyższego rzędu jest o wiele wygodniejsze.

Dlaczego chcesz, aby blok przyjmował procedurę jako dane wejściowe? W rzeczywistości nie jest to czymś dziwnym; pierwotne bloki warunkowe i pętle (te w kształcie litery C w palecie Kontrola) przyjmują skrypt jako dane wejściowe. Użytkownicy zazwyczaj nie myślą o tym w takich kategoriach! Moglibyśmy napisać blok **powtórz** jako blok użytkownika w ten sposób, gdyby Snap! go nie miał:

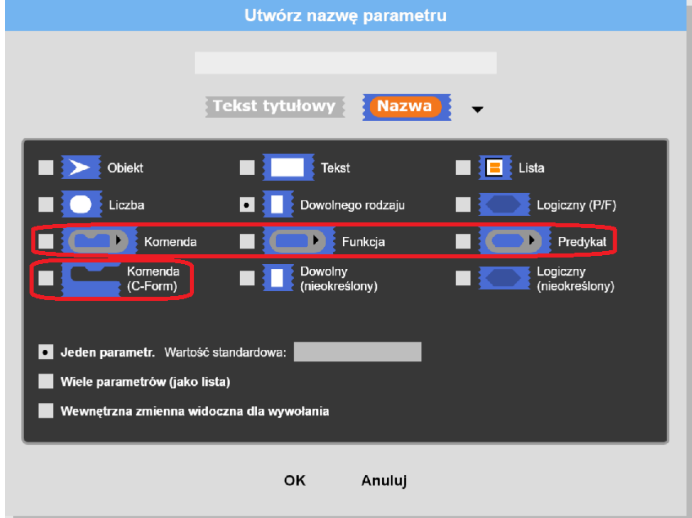
+powtarzaj+ ile + akcja λ +
jeżeli ile > 0 to
uruchom akcja z
powtarzaj ile − 1 akcja

Lambda (λ) obok nazwy **akcja** w prototypie wskazuje, że jest to blok w kształcie litery C, a skrypt wewnątrz C, gdy blok jest używany, jest parametrem o nazwie **akcja** w prototypie skryptu. Jedynym sposobem poznania sensu zmiennej **akcja** jest zrozumienie, że jej wartość jest skryptem.

Aby zadeklarować dane wejściowe jako typ Procedury, otwórz okno dialogowe nazwy parametru i kliknij strzałkę:




Następnie w długim oknie dialogowym wybierz odpowiedni typ Procedury. Trzeci rząd typów wejściowych ma obwiednie w kształcie każdego typu bloku (układanki dla Komend, owal dla Funkcji i sześciokątny dla Predykatów). W praktyce jednak w przypadku Komend częściej wybiera się pole w kształcie litery C w czwartym rzędzie, ponieważ ten „kontener” dla skryptów poleceń jest znany użytkownikom Scratcha. Technicznie, pole w kształcie litery C jest nieokreślonym typem procedury, co omówiono w części E poniżej. Obydwa typy danych wejściowych związane z Komendami (w kształcie liniowym i litery C) łączy fakt, że jeśli zmienna, blok **element (#) z [lista]** lub niestandardowy blok Funkcji zostanie upuszczony w polu w kształcie litery C, to zamienia się ono w pole liniowe. Tak jak to się stało w rekurencyjnym wywołaniu bloku **powtarzaj** powyżej. (Inne wbudowane Funkcje nie mogą przekazywać skryptów, więc nie są akceptowane w polu w kształcie litery C).



Czy kiedykolwiek może przydać się liniowe pole Komendy zamiast C-kształtnego? Poza omawianym poniżej blokiem **uruchom**, jedyny przypadek, jaki mogę sobie wyobrazić, to coś takiego jak pętla **for** w C / C ++ / Java, która faktycznie ma trzy pola wejścia skryptów poleceń (i jedno predykatów), z których tylko jeden jest „wpisany” w ciało pętli:

Dobra, skoro mamy procedury jako wejścia do naszych bloków, to jak ich używać? Używamy bloków **uruchom** (dla poleceń) i **wywołaj** (dla funkcji). Skrypt wejściowy bloku **uruchom** ma obwiednię liniową, nie w kształcie litery C, ponieważ przewidujemy, że rzadko będzie używany specyficzny, literalny skrypt jako dane wejściowe. Zamiast tego, dane wejściowe będą na ogół zmienną, której *wartością* jest skrypt.

Bloki **uruchom** i **wywołaj** mają na końcu strzałki, które mogą służyć do otwierania pól wejść do wywoływanych procedur. Skąd Snap*!* wie, gdzie użyć tych danych wejściowych? Jeśli wywołana procedura (blok lub skrypt) ma puste gniazda wejściowe, Snap*!* „robi co trzeba”. Ma to kilka możliwych znaczeń:

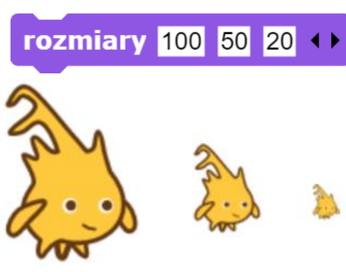
1. Jeśli liczba pustych pól jest dokładnie równa liczbie wprowadzonych wejść, to Snap*!* wypełnia puste miejsca od lewej do prawej:
2. Jeśli podano dokładnie jedno wejście, Snap*!* wypełni jego wartością dowolną liczbę pustych pól:
3. W przeciwnym razie Snap*!* nie zapełni żadnego pola, ponieważ zamiary użytkownika nie są jasne.

Jeśli użytkownik chce nadpisać te reguły, rozwiązaniem jest użycie obwiedni z jawnie wprowadzonymi nazwami parametrów, które można umieścić w danym bloku lub skrypcie, aby wskazać sposób użycia danych wejściowych. Zostanie to omówione bardziej szczegółowo poniżej.

## Rekurencyjne wywołania bloków z wieloma wejściami

Stosunkowo rzadką sytuacją, która nie została jeszcze rozważona, jest przypadek bloku rekurencyjnego, który ma zmienną liczbę wejść. Załóżmy, że użytkownik twojego projektu wywołuje twój blok raz z pięcioma wejściami, a innym razem z 87 wejściami. Jak napisać rekurencyjnie wywołanie twojego bloku, gdy nie wiesz, ile będzie mieć wejść? Odpowiedź brzmi, że zbierasz dane wejściowe na liście (pamiętaj, że po zadeklarowaniu nazwy wejścia reprezentującej zmienną liczba wejść, twój blok widzi te wejścia w pierwszej kolejności jako listę wartości), a następnie, w wywołaniu rekurencyjnym, upuszczasz tę listę wejściową na strzałki, które wskazują możliwość zmiany liczby parametrów, a nie na polu wejściowym:

Zwróć uwagę, że halo, które widzisz podczas przeciągania na strzałki, jest czerwone, a nie białe, i obejmuje zarówno pole wejściowe, jak i strzałki. A gdy upuścisz wyrażenie na strzałki, słowa „**parametr - lista:**" zostaną dodane do tekstu bloku, a strzałki znikną (tylko w tym wywołaniu), aby przypomnieć ci, że lista reprezentuje wszystkie z wejść, a nie tylko pojedyncze wejście. Każdy element listy jest traktowany jako dana wejściowa do skryptu. Ponieważ **liczby** są listą liczb, każdy pojedynczy element jest liczbą, dokładnie taką, jakiej oczekuje blok **rozmiary**. Ten blok przyjmie dowolną liczbę liczb jako dane wejściowych i sprawi, że duszek odpowiednio powiększy się lub zmniejszy:



Użytkownik tego bloku używa go z dowolną liczbą danych wejściowych. Ale w definicji bloku wszystkie te liczby tworzą listę, która ma jedną nazwę wejściową, **liczby**. Ta rekurencyjna definicja najpierw powoduje sprawdzenie, czy w ogóle istnieją jakieś dane wejściowe. Jeśli tak, przetwarza pierwsze wejście (**element 1** listy), a następnie ma wykonać wywołanie rekurencyjne z wszystkimi oprócz pierwszego elementu. Ale blok **rozmiary** nie uwzględnia listy jako danych wejściowych; bierze liczby jako dane wejściowe! Tak byłoby źle:

## C Parametry formalne

Obwiednie wokół wejść typu Procedura mają strzałki po prawej stronie. Kliknięcie strzałek pozwala nadać wejściom jawne nazwy bloków lub skryptów, zamiast używać pustych pól wejściowych, jak robiliśmy to do tej pory.

Nazwy **# 1**, **# 2** itp. są domyślnie dostępne, ale możesz zmienić nazwę, klikając jej pomarańczowy owal na liście nazwy **parametrów**. Uważaj, aby nie przeciągać owalu po kliknięciu; w ten sposób używasz wejścia wewnątrz obwiedni. Nazwy zmiennych wejściowych nazywane są parametrami formalnymi enkapsulowanej procedury.

Oto prosty, ale przemyślny przykład użycia jawnych nazw, aby kontrolować, jak dane wejście wchodzą do wnętrza obwiedni:

Tutaj chcemy po prostu umieścić jedno z wejść w dwóch różnych polach. Jeśli zostawilibyśmy wszystkie trzy miejsca puste, Snap*!* nie wypełniłby żadnego z nich, ponieważ liczba wprowadzonych danych wejściowych (2) nie byłaby równa liczbie pustych miejsc (3).

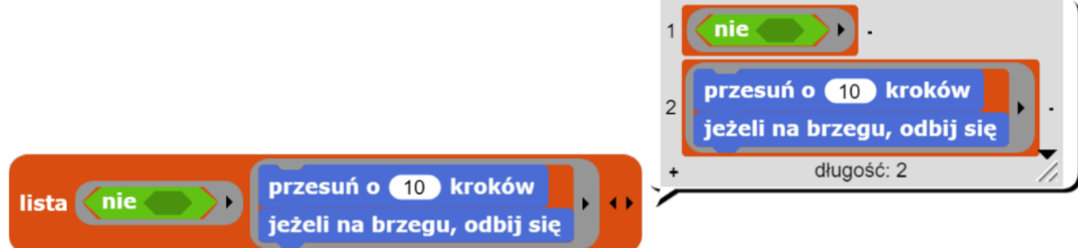
Oto bardziej realistyczny i znacznie bardziej zaawansowany przykład:

Jest to definicja bloku, który przyjmuje dowolną liczbę list i przekazuje listę wszystkich możliwych kombinacji jednego elementu z każdej listy. Ważną częścią tej dyskusji jest to, że w dolnej części znajdują się dwa zagnieżdżone wywołania **mapuj** (**map**), funkcji wyższego rzędu, która stosuje funkcję wejściową do każdego elementu listy wejściowej. W bloku wewnętrznym odwzorowywana funkcja jest wstawiana do bloku **wstaw … przed** (**in front of**), a ten blok ma dwa wejścia. Drugie, puste pole typu listowego, otrzyma w każdym wywołaniu swoją wartość od elementu na liście wejściowej wewnętrznego **mapuj**. Ale zewnętrzne **mapuj** nie może przekazać wartości pustym gniazdom w bloku na **wstaw … przed**. Musimy jednoznacznie podać nazwę, **nowyelem**, wartości, jaką zewnętrzne **mapuj** nadaje wewnętrznemu, a następnie przeciągnąć tę zmienną do bloku **wstaw … przed**.

Nawiasem mówiąc, gdy wywołany blok podaje nazwy dla swoich wejść, Snap*!* nie wypełni automatycznie pustych miejsc na podstawie teorii, że użytkownik przejął kontrolę. W rzeczywistości jest to kolejny powód, dla którego możesz chcieć jawnie nazwać wejścia: aby powstrzymać Snap*!* od wypełnienia pola, które naprawdę powinno pozostać puste.

## D Procedury jako dane

Oto przykład sytuacji, w której procedura musi być jawnie oznaczona jako dane przez wyciągnięcie obwiedni z palety Wyrażenia i umieszczenie w niej procedury (bloku lub skryptu):



Tutaj tworzymy listę procedur. Ale blok listy akceptuje wejścia każdego typu, więc jego gniazda wejściowe nie mają obwiedni. Musimy wyraźnie stwierdzić, że chcemy, aby sam blok był wartością wejściową, a nie jakąkolwiek wartością wynikającą z wykonania bloku.

Oprócz bloku **lista** w powyższym przykładzie, inne bloki, w które możesz chcieć wstawić procedury to **ustaw…na** (aby ustawić procedurę jako wartość zmiennej), **powiedz** i **pomyśl** (aby wyświetlić użytkownikowi procedurę) i **wynik** (dla funkcji przekazującej procedurę):

## E Formy specjalne

Pierwotny blok **jeżeli … w przeciwnym razie** ma dwa pola wejściowe poleceń w kształcie litery C i wybiera jeden lub drugi w zależności od wyniku testu logicznego. Ponieważ w Scratchu nie ma nacisku na programowanie funkcyjne, brakuje mu odpowiedniego bloku funkcji umożliwiającego wybór między dwoma wyrażeniami. Możemy napisać taki blok w Snap*!*:

Nasz blok działa na tych prostych przykładach, ale jeśli spróbujemy użyć go do napisania funkcji rekurencyjnej, to zawiedzie:

Problem polega na tym, że po wywołaniu dowolnego bloku wszystkie jego dane wejściowe są obliczane (oceniane) przed wykonaniem samego bloku. A blok zna tylko wartości swoich danych wejściowych, a nie wyrażeń użytych do ich obliczenia. W szczególności, wszystkie dane wejściowe do naszego bloku **jeśli … w przeciwnym razie** są obliczane jako pierwsze. Oznacza to, że nawet w przypadku podstawowym silnia spróbuje wywołać się rekurencyjnie, powodując nieskończoną pętlę. Potrzebujemy aby nasz blok, **jeśli … w przeciwnym razie** wybrał tylko jedną z dwóch alternatyw do obliczenia.

Mamy mechanizm, który pozwala na to: trzeba zadeklarować, że dane wejściowe bloku **jeśli … w przeciwnym razie** będą typu Funkcja, a nie typu Dowolny. Wtedy przy wywołaniu bloku, te wejścia będą zamknięte w obwiednie, tak że same wyrażenia, a nie ich wartości, staną się wejściami:

W tej wersji program działa bez nieskończonej pętli. Ale zapłaciliśmy wysoką cenę: ta funkcja - **jeśli** nie jest już tak intuicyjnie oczywista, jak polecenie **jeżeli** ze Scratcha. Musisz coś wiedzieć o procedurach jako danych, o obwiedniach i o sztuczce, aby uzyskać stałą wartość w polu z obwiednią. (Blok **id** implementuje funkcję tożsamościową, która przekazuje swoje wejście. Potrzebujemy tego, ponieważ obwiednie przyjmują tylko funkcje, a nie liczby jako dane wejściowe.). Chcielibyśmy, aby funkcja - **jeśli** tak się zachowywała, opóźniając obliczanie swoich wejść, ale wygląda na naszą pierwszą wersję, która była łatwa w użyciu, z tym wyjątkiem, że nie działała.

Takie bloki są rzeczywiście możliwe. Blok, który wydaje się przyjmować proste wyrażenie jako dane wejściowe, ale opóźnia obliczanie tego wyrażenia poprzez otoczenie go „niewidoczną obwiednią" (i, jeśli to konieczne, przekształcenie stałych danych w stałe funkcje jak to robi **id**) jest nazywany formą specjalną. Aby przekształcić nasz blok **jeśli** w specjalną formę, edytujemy prototyp bloku, deklarując, że dane wejściowe **tak** i **nie** mają typ "Dowolny (nieokreślony)" zamiast typu Funkcja. Skrypt dla bloku jest nadal w wersji drugiej, włącznie z wykorzystaniem **wywołaj** do oceny **tak** lub **nie**, ale nie obu. Ale pola pojawiają się jako białe prostokąty, jak dla wejścia Dowolnego rodzaju, a nie obwiednie typu Funkcja, a blok **silnia** będzie wyglądał jak nasza pierwsza wersja.

W prototypie specjalnej formy, nieobliczane pola wejściowe są oznaczone przez lambda (λ) obok nazwy wejścia, tak jak gdyby zostały zadeklarowane jako typ procedury. One *są* typu procedury, naprawdę; po prostu są przystrojone dla użytkownika bloku.

Specjalne formy są rodzajem handlu między wyrafinowanym implementatorem a zaawansowanym użytkownikiem. Oznacza to, że musisz rozumieć wszystkie procedury jako dane, aby nadać sens implementacji specjalnej formy w bloku **jeśli…to…w przeciwnym razie**. Ale każdy doświadczony programista Scratcha może wykorzystać, **jeśli…to…w przeciwnym razie**, bez myślenia, jak to działa wewnętrznie.

### Specjalne formy w Scratchu

Specjalne formy w rzeczywistości nie są nowym wynalazkiem w Snap*!*. Wiele bloków warunkowych i pętli Scratcha to naprawdę specjalne formy. Sześciokątny otwór wejściowy w bloku **jeżeli** jest zwykłą wartością logiczną, ponieważ wartość można obliczyć jeden raz, zanim blok **jeżeli** podejmie decyzję o tym, czy uruchomić dane wejściowe, czy też nie. Ale w blokach **powtarzaj…aż** i **czekaj…aż** wejścia warunku nie mogą być zwykłymi wartościami logicznymi; muszą być typu „logiczny (nieobliczany)", aby Scratch mógł je wielokrotnie obliczać. Ponieważ Scratch nie ma niestandardowych (definiowanych przez użytkownika) bloków funkcji, może pozwolić sobie na zaniechanie rozróżnienia między obliczanymi i nieobliczanymi typami logicznymi, ale Snap*!* nie. O wartości pedagogicznej form specjalnych świadczy fakt, że żaden użytkownik Scratcha nigdy nie zauważa, że jest coś dziwnego w sposobie oceny sześciokątnych danych wejściowych w blokach Kontroli.

Ponadto gniazdo w kształcie litery C, znane użytkownikom Scratcha, jest typu procedury nieobliczanej; nie musisz używać obwiedni, aby nie obliczać poleceń wewnątrz C przed uruchomieniem bloku C-kształtnego. Same polecenia, a nie wynik ich działania, stanowią dane wejściowe do bloku Kontroli w kształcie litery C. (Jest to oczywiste dla użytkowników Scratch, szczególnie dlatego, że polecenia nie przekazują wartości, więc nie ma sensu myślenie o umieszczaniu poleceń w polu o kształcie litery C jako układu funkcji). Właśnie dlatego ma to sens że typ „w kształcie litery C (C-form)" znajduje się w czwartym rzędzie typów w długim oknie dialogowym wprowadzania nazwy parametru, z innymi typami nieobliczanymi.

# VII Programowanie obiektowe i duszki

Programowanie obiektowe (OOP – Object Oriented Programming) jest oparte na abstrakcyjnym pojęciu *obiektu* czyli zbioru danych i metod (procedur, które z naszego punktu widzenia oznaczają więcej danych), z którym użytkownik komunikuje się, wysyłając mu *wiadomość* (to tylko nazwa, może mieć postać ciągu tekstowego i być może dodatkowych danych wejściowych). Obiekt odpowiada na wiadomość, wykonując metodę, która może, ale nie musi, przekazać wartość z powrotem do pytającego. Niektórzy podkreślają aspekt *ukrywania danych* w OOP (ponieważ każdy obiekt ma zmienne lokalne, do których inne obiekty mogą uzyskać dostęp tylko poprzez wysyłanie komunikatów z pytaniem do obiektu będącego właścicielem), podczas gdy inni podkreślają aspekt *symulacji* (w którym każdy obiekt reprezentuje coś w rzeczywistości, a interakcje obiektów w programie modelują rzeczywiste interakcje ludzi lub rzeczy). Ukrywanie danych jest ważne w dużych projektach przemysłowych tworzonych przez wielu programistów, ale dla użytkowników Snap*!* ważny jest aspekt symulacji. Nasze podejście jest zatem mniej restrykcyjne niż w przypadku niektórych innych języków OOP; dajemy obiektom łatwy dostęp do danych i metod innych obiektów.

Z technicznego punktu widzenia programowanie obiektowe opiera się na trzech filarach: (1) *Przekazywanie komunikatów*: Istnieje notacja, dzięki której dowolny obiekt może wysłać wiadomość do innego obiektu. (2) *Stan lokalny*: każdy obiekt może pamiętać istotną historię wykonanych obliczeń. („Istotną" oznacza, że nie musi pamiętać wszystkich wiadomości, które obsłużył, ale tylko trwałe efekty tych wiadomości, które wpłyną na późniejsze obliczenia.) (3) *Dziedziczenie*: Byłoby niepraktyczne, gdyby każdy pojedynczy obiekt musiał zawierać metody, wiele z nich identycznych jak w innych obiektach, dla wszystkich wiadomości, które może zaakceptować. Zamiast tego potrzebujemy sposobu, aby powiedzieć, że nowy obiekt jest taki, jak stary, z wyjątkiem kilku różnic, tak że tylko te różnice muszą być jawnie oprogramowane.

## A Duszki pierwszej kategorii

Podobnie jak Scratch, Snap*!* ma rzeczy, które są naturalnymi obiektami: swoje duszki. Każdy duszek może posiadać zmienne lokalne; każdy duszek ma własne skrypty (metody). Animacja w Scratchu jest po prostu symulacją interakcji postaci w grze. Z dwóch powodów duszki Scratcha są mniej wszechstronne niż obiekty języka obiektowego. Po pierwsze, przekazywanie wiadomości jest prymitywne pod trzema względami: wiadomości mogą być **nadawane**, nie adresowane do pojedynczego duszka; wiadomości nie mogą przyjmować danych wejściowych; wreszcie metody nie mogą zwracać wartości do wywołującego. Po drugie i bardziej podstawowe, w paradygmacie OOP obiekty są *danymi*; mogą być wartością zmiennej, elementem listy i tak dalej, ale tak nie jest w przypadku duszków Scratcha.

W SNAP*!* duszki są danymi pierwszej klasy. Mogą być tworzone i usuwane przez skrypt, przechowywane w zmiennej lub liście i mogą wysyłać indywidualnie wiadomości. Dzieci duszka mogą odziedziczyć lokalne zmienne duszka, metody (lokalne procedury duszka) i inne atrybuty (np. **pozycja X**).

Podstawowym sposobem uzyskania przez program dostępu do duszków jest blok funkcji **ja**. Ma on wejście z rozwijanym menu, które po kliknięciu daje dostęp do wszystkich duszków i sceny. ja sam daje w wyniku duszka, zadającego pytanie. ja inne duszki daje listę wszystkich duszków innych niż ten, który zadaje pytanie. ja sąsiedzi daje listę wszystkich duszków, które są blisko pytającego – na przykład tych, które za chwilę mogą się z nim zderzyć. Blok **ja** ma wiele innych opcji, omówionych poniżej.

Obiekt lub lista obiektów przekazanych przez **ja** może być użyta jako wejście do bloku, akceptującego dowolny typ wejścia, jak drugie wejście bloku **ustaw**. Jeśli uruchomisz **powiedz** obiekt, uzyskany dymek będzie zawierał mniejszy obraz kostiumu obiektu lub (dla sceny) tła.

## B Klony stałe i tymczasowe

Blok **nowy klon** ja służy do tworzenia i przekazywania instancji (klona) każdego duszka. (Istnieje również z przyczyn historycznych wersja będąca komendą). Są dwa różne rodzaje sytuacji, w których używane są klony. Jedna z nich jest taka: stworzyłeś przykładowego duszka i kiedy uruchamiasz projekt, potrzebujesz dość dużej liczby prawie identycznych duszków, które zachowują się jak przykładowy. (Stąd nazywamy przykładowego duszka „rodzicem", a inne „dziećmi" [potomkami]). Po zakończeniu gry lub animacji, nie potrzebujesz już kopii. (Jak zobaczymy, „kopie" to niewłaściwe słowo, ponieważ rodzic i dzieci mają wiele wspólnych właściwości, dlatego używamy słowa „klon", a nie „kopia", aby opisać dzieci). Są to klony ***tymczasowe***. W Scratchu 2.0 wszystkie klony są tymczasowe.

Innym rodzajem sytuacji jest to, co się dzieje, gdy chcesz specjalizacji duszków. Na przykład, powiedzmy masz duszka o imieniu Pies. Ma on pewne zachowania, takie jak przybieganie do osoby, która się do niego zbliża. Teraz decydujesz, że rodzina w twojej historyjce naprawdę lubi psy, więc adoptuje ich dużo. Niektóre z nich to cocker spaniele, które machają ogonem, kiedy cię widzą. Inne to rottweilery, które warczą, kiedy cię widzą. Robisz więc klona psa, zmienisz jego nazwę na Cocker Spaniel, nadajesz mu nowy kostium i scenariusz, co ma robić, gdy ktoś się zbliża. Robisz kolejnego klona psa, zmienisz jego nazwę na Rottweiler, nadajesz mu nowy kostium itp. Następnie tworzysz trzy klony Cocker Spaniela (więc jest ich w sumie cztery) i dwa klony Rottweilera. Może po tym wszystkim ukrywasz duszka psa, ponieważ to nie jest żadna rasa. Każdy pies ma swoją własną pozycję, specjalne zachowania i tak dalej. Chcesz zapisać wszystkie te psy w projekcie. Są to ***trwałe*** klony. W programie BYOB 3.1, poprzedzającym Snap*!*, wszystkie klony były trwałe.

Zaletą tymczasowych klonów jest to, że nie spowalniają one działania SNAP*!* nawet gdy masz ich dużo. (Jeśli cię to ciekawi, jeden z powodów jest taki, że stałe klony pojawiają się w zagrodzie duszków, gdzie ich obrazki muszą być aktualizowane w celu odzwierciedlenia aktualnego stroju, kierunku, itd.). Próbowaliśmy przewidzieć twoje potrzeby, tak jak następuje: Kiedy robisz klona w skrypcie, używając bloku **nowy klon** ja wtedy jest on „urodzony" jako chwilowy. Ale kiedy zrobisz klona z interfejsu użytkownika, na przykład klikając prawym przyciskiem myszy ikonkę i wybierając „klonuj", rodzi się trwały. Powód, dla którego ma to sens, jest taki, że nie tworzy się automatycznie 100 rodzajów psów. Każdy rodzaj ma wiele różnych cech. Ale kiedy projekt jest uruchomiony, może utworzyć 100 rottweilerów, które będą identyczne, chyba że zmienisz je w programie.

Możesz zmienić tymczasowego duszka na trwałego, klikając go prawym przyciskiem myszy i wybierając „edytuj". (Nazywa się to „edytuj", a nie „trwały", ponieważ przenosi obszar skryptu tak, aby odzwierciedlał tę ikonkę, tak jakby wciśnięty został przycisk [ikona duszka, *WtK*] w zagrodzie duszków). Możesz zmienić trwałego duszka na tymczasowego, klikając go [w zagrodzie duszków, *WtK*] prawym przyciskiem myszy i wybierając „wydanie" [powinno być raczej „uwolnij", *WtK*].

## C Wysyłanie komunikatów do duszków

Wiadomości, które akceptuje duszek to bloki w jego paletach, zarówno bloki odnoszące się do wszystkich duszków, jak i tylko do tego jednego. (W przypadku bloków niestandardowych odpowiednimi metodami są skrypty widoczne w edytorze bloków).

Scratch umożliwia duszkowi uzyskanie pewnych własności drugiego: blok <**właściwość**> **z** <**duszek**> w palecie Czujniki. Snap*!* rozszerza ten blok, aby umożliwić dostęp do dowolnej własności duszka. Ze względu na kompatybilność ze Scratchem dwa pola wejściowe w tym bloku są rozwijanymi menu, ale można przeciągnąć dowolne wyrażenie o wartości obiekt do prawego, a każdy blok w *obwiedni* lub skrypt do lewego pola. Wynik może być używany z blokami **wywołaj** lub **uruchom**, aby zastosować odpowiednią metodę.

Z perspektywy użytkownika blok **przesuń** ma charakter globalny; kiedy jest używany w skrypcie, porusza tego duszka, którego skrypt jest uruchamiany. Ale sposób, w jaki został faktycznie zaimplementowany, polega na tym, że każdy duszek ma własny zestaw bloków ruchu i innych bloków pierwotnych specyficznych dla duszków. Tak więc w palecie Duszek(4) znajduje się blok „przesuń duszek(4) o 10 kroków". Dlatego właśnie gdy użyje się bloku **…z…**, jak na obrazku powyżej po lewej, wynikiem jest blok lokalny duszka, który zawsze porusza Duszka(4) niezależnie od tego, który duszek go uruchomi. (Nie musisz go chwytać z palety Duszka(4), ponieważ blok **…z…** uruchomi go tak, jakby został z niej wyciągnięty).

Dla wygody Snap*!* zawiera blok **zacznij**, który jest identyczny do **uruchom**, z tym wyjątkiem, że wywołuje metodę jako osobny skrypt, więc skrypt wywołujący może w międzyczasie zrobić coś innego. Blok **zacznij** może być uważany za ulepszony blok **nadaj** <komunikat>, podczas gdy **uruchom** metodę innego duszka jest bardziej analogiczny do **nadaj** <komunikat> **i czekaj**.

Zauważ, że gdy dany blok lub skrypt zostanie wprowadzony jako dane wejściowe, blok **…z…** daje w wyniku ten blok lub skrypt, gotowy do uruchomienia przez **uruchom**. Więc nie potrzeba dodatkowej obwiedni, którą blok **uruchom** ma w swoim polu wejścia. Aby te przykłady wyglądały jak obrazki, należy kliknąć prawym przyciskiem myszy blok i wybrać „bez obwiedni" (w nowszej wersji dzieje się to automatycznie, *WtK*). Alternatywnie może się tym dla ciebie zająć blok **powiedz** (lub blok **zapytaj**, dla funkcji):

Chociaż powyższe obrazki pokazują blok **przesuń** z wypełnionym polem wejściowym, wiadomości z pustymi wejściami również mogą być używane i można im jak zwykle podawać wartości wejściowe przez **wywołaj** lub **uruchom**. Zarówno blok **…z…**, jak i blok **powiedz** mogą wziąć jako dane wejściowe albo nazwę duszka, albo rzeczywistego duszka:

## Polimorfizm

Załóżmy, że masz duszka Pies z dwoma klonami CockerSpaniel i PitBull. W duszku Pies definiujesz tę metodę (lokalny blok duszka):

Zwróć uwagę na symbol pinezki przed nazwą bloku. Symbol nie jest częścią tytułu bloku; to wizualne przypomnienie, że jest to blok lokalny duszka (tylko dla tego duszka – przy tworzeniu bloku, *WtK*).

Ale nie definiujesz **witaj … jako przyjaciela** lub **witaj … jako wroga** w Psie. Każdy rodzaj psa ma inne zachowanie.

Oto co robi CockerSpaniel:

witaj gość jako przyjaciela
machaj ogonem

witaj gość jako wroga
machaj ogonem

A to co robi PitBull:

witaj gość jako przyjaciela
machaj ogonem

witaj gość jako wroga
warcz

**Witaj** jest zdefiniowane tylko w duszku Psie. Jeśli Fido jest szczególnym cocker spanielem, a ty poprosisz Fido – **witaj** się z kimś, Fido odziedziczy metodę **witaj** z Psa, ale sam Pies nie może uruchomić tej metody, ponieważ pies nie zna **witaj jako przyjaciela** lub **witaj jako wroga**. A być może tylko pojedyncze psy, takie jak Fido, mają metodę **przyjaciel?**. Blok **witaj** Psa jest nazywany metodą *polimorficzną*, ponieważ oznacza różne rzeczy dla różnych psów, nawet jeśli wszystkie mają ten sam skrypt.

## D Stan lokalny duszka: zmienne i atrybuty

Pamięć duszka o własnej przeszłości ma dwie główne formy. Posiada on *zmienne*, utworzone jawnie przez użytkownika za pomocą przycisku „Utwórz zmienną"; ma również atrybuty, cechy, które każdy duszek ma przydzielane automatycznie, takie jak pozycja, kierunek i kolor pisaka. Wartość każdej zmiennej można uzyskać za pomocą własnego owalnego pomarańczowego bloku; jest jeden blok **ustaw** do modyfikowania wszystkich zmiennych. Atrybuty w Scratchu mają jednak mniej jednolity interfejs programistyczny. Kierunek duszka można sprawdzić za pomocą bloku **kierunek** i zmodyfikować za pomocą bloku **ustaw kierunek na** <***kier***>. Można go również modyfikować mniej bezpośrednio za pomocą bloków **obróć**, **ustaw w stronę** i **jeśli na brzegu, odbij się**. Nie ma sposobu, aby skrypt sprawdził kolor pisaka, ale istnieją bloki **ustaw kolor pisaka na** <***kolor***>, **ustaw kolor pisaka na** <*liczba*> i **zmień kolor pisaka o** <*liczba*>, pozwalające go zmodyfikować. Nazwa duszka nie może być ani badana, ani modyfikowana przez skrypty; można ją zmieniać, wpisując nową nazwę bezpośrednio w polu, które wyświetla nazwę, w zagrodzie duszków pod sceną. Blok, jeśli taki istnieje, badający zmienną lub atrybut jest nazywany getter (pobieracz); blok (może być więcej niż jeden, jak w powyższych przykładach), który modyfikuje zmienną lub atrybut jest nazywany setter (ustalacz).

W Snap*!* mamy bardziej jednolity interfejs dla atrybutów. Menu bloku **ja** zawiera wiele ustawialnych atrybutów duszka. Zachowaliśmy jednak większość bloków pobierających i ustawiających atrybuty ze Scratcha dla wygody użytkownika.

## E Prototypowanie: rodzice i dzieci

Większość obecnych języków OOP używa do tworzenia obiektów podejścia klasa / instancja. Klasa jest szczególnym rodzajem obiektu, a instancja jest faktycznym obiektem tego typu. Na przykład może istnieć klasa Pies i kilka instancji Fido, Muszka i Pimpek. Klasa zwykle określa metody wspólne dla wszystkich psów (Waruj, Służ, doNogi itd.), a instancje zawierają dane takie jak gatunek, kolor i przyjazność. Snap*!* stosuje inne podejście zwane prototypowaniem, w którym nie ma rozróżnienia pomiędzy klasami i instancjami. Prototypowanie lepiej pasuje do eksperymentalnego, badawczego stylu pracy: Tworzysz pojedynczego duszka pies, używając obu metod (bloków) i danych (zmiennych), możesz go oglądać i wchodzić w interakcje z nim na scenie, a kiedy ci się spodoba używasz go jako prototypu do klonowania innych psów. Jeśli później odkryjesz błąd w zachowaniu psów, możesz edytować metodę rodzica, a wszystkie dzieci automatycznie będą również miały nową wersję bloku metod.

Programiści obeznani w podejściu typu klasa / instancja mogą na początku uznać prototypowanie za dziwne, ale tak naprawdę to jest bardziej ekspresyjny system, ponieważ można łatwo zasymulować hierarchię klasa / instancja, ukrywając prototypowego duszka! Prototypowanie jest również lepiej dopasowane do zasady Scratcha, że wszystko w projekcie powinno być konkretne i widoczne na scenie; w podejściu OOP klasa / instancja proces programowania rozpoczyna się od abstrakcyjnego, niewidzialnego bytu, klasy, którą należy zaprojektować, zanim możliwe będzie wykonanie konkretnych obiektów.[10]

Istnieją trzy sposoby utworzenia potomka duszka. Jeśli klikniesz z ctrl lub klikniesz prawym klawiszem myszy duszka w „zagrodzie duszków" w prawym dolnym rogu okna, otrzymasz menu zawierające opcję „klonuj". W palecie Kontrola znajduje się blok **nowy klon**, który tworzy i daje w wyniku potomka duszka. Duszki mają atrybut „rodzic", który można ustawić, jak każdy atrybut, zmieniając w ten sposób rodzica istniejącego duszka.

## F Dziedziczenie przez delegowanie

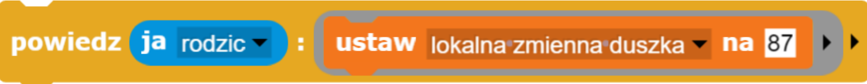
Klon *dziedziczy* własności swojego rodzica. Do „własności" zaliczamy skrypty, bloki zdefiniowane przez użytkownika, zdefiniowane listy, atrybuty systemowe, kostiumy i dźwięki. Każda własność może zostać przekazana przez rodzica potomkom lub nie (wtedy potomek ma inną jej wartość). Blok getter (pobierający) dla własności współdzielonej, jest wyświetlany w jaśniejszym kolorze w palecie potomka; oddzielne właściwości potomka są wyświetlane w tradycyjnych kolorach.

Kiedy tworzony jest nowy klon, domyślnie dziedziczy tylko swoje metody, garderobę i szafą grającą rodzica. Wszystkie pozostałe własności są kopiowane do klona, ale nie są udostępniane. (Jednym wyjątkiem jest to, że nowy trwały klon ma losową pozycję, innym jest to, że tymczasowe klony współdzielą skrypty w obszarze skryptowym swojego rodzica, a trzeci to to, że zmienne lokalne, które rodzic tworzy po klonowaniu, są współużytkowane z jego potomkami). Jeśli wartość własności wspólnej zostanie zmieniona w rodzicu, wówczas jego dzieci zobaczą nową wartość. Jeśli wartość własności współużytkowanej zostanie zmieniona w potomku, łącze udostępniania zostanie zerwane, a nowa prywatna wersja zostanie utworzona w tym potomku. (Jest to mechanizm, dzięki któremu potomek decyduje się nie dzielić własności z rodzicem). „Zmienione" w tym kontekście oznacza użycie bloku **ustaw** lub **zmień** dla zmiennej, edycję bloku w edytorze bloku, edycję kostiumu lub dźwięku lub wstawianie, usuwanie lub zmianę kolejności kostiumów lub dźwięków. Aby zmienić własność z nieudostępnionej na wspólną, potomek korzysta z bloku polecenia **dziedzicz**. Rozwijane menu tego bloku zawiera listę wszystkich rzeczy, które dany duszek może odziedziczyć po rodzicu (co może być puste, jeśli ten duszek nie ma rodzica) i jeszcze ich nie dziedziczy. Ale to uniemożliwiłoby potokowi dziedziczenie, więc jeśli blok **dziedzicz** jest w obwiedni, jego menu rozwijane

[10] Niektóre języki popularne w dzisiejszym „realnym świecie", takie jak JavaScript, udają, że używają prototypów, ale ich system obiektowy jest znacznie bardziej skomplikowany niż to, co opisujemy (domyślamy się przyczyny - zostały zaprojektowane przez ludzi zbyt dobrze znających programowanie w podejściu klasa / instancja), i w niektórych kręgach prototypowanie ma złą sławę. Nasze podejście do prototypowania pochodzi z Object Logo, a wcześniej od Henry'ego Liebermana. [Lieberman, H., Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems, First Conference on Object-Oriented Programming Languages, Systems, and Applications [OOPSLA-86], ACM SigCHI, Portland, OR, September, 1986. Także w Object-Oriented Computing, Gerald Peterson, Ed., IEEE Computer Society Press, 1987.]

zawiera wszystkie rzeczy, które potomek może odziedziczyć po tym duszku. Kliknięcie prawym przyciskiem myszy obszaru skryptowego trwałego klona daje opcję menu udostępniającą cały zbiór skryptów od rodzica, takich jakie ma tymczasowy klon.

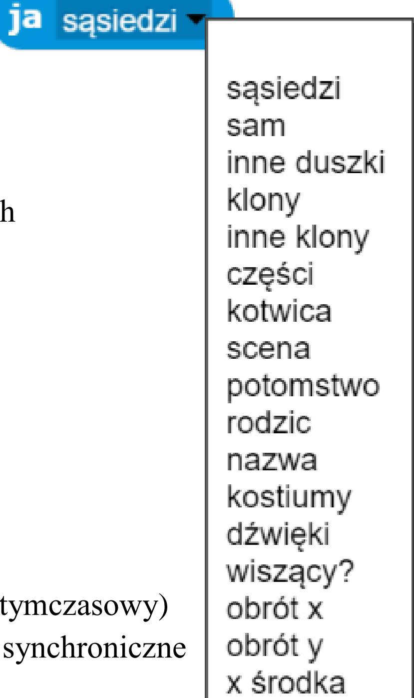
Zasady są pełne szczegółów, ale podstawowa idea jest prosta: rodzice mogą zmienić swoje dzieci, ale dzieci nie mogą bezpośrednio zmienić swoich rodziców. Tego można się spodziewać po słowie "dziedziczenie": wpływ po prostu idzie w jednym kierunku. Kiedy potomek zmienia jakąś własność, deklaruje niezależność od swojego rodzica (w odniesieniu do tej jednej własności). A co, jeśli naprawdę chcesz, aby potomek mógł dokonać zmiany w rodzicu (a więc w sobie i całym jego rodzeństwie)? Pamiętaj, że w tym systemie każdy obiekt może powiedzieć (blok **powiedz**) dowolnemu innemu obiektowi, aby coś zrobił:




Kiedy duszek dostaje wiadomość, dla której nie ma odpowiedniego bloku, wiadomość jest delegowana do rodzica tego duszka. Kiedy duszek ma odpowiedni blok, to wiadomość nie jest delegowana. Jeśli skrypt implementujący delegowaną wiadomość odnosi się do **ja** (***sam***), oznacza to potomka, do którego wiadomość została pierwotnie wysłana, a nie rodzica, któremu przekazano wiadomość.

## G Lista atrybutów

Po prawej stronie znajduje się obrazek rozwijanego menu atrybutów w bloku **ja**. Kilka z nich nie jest prawdziwymi atrybutami, ale raczej *listami* rzeczy związanych z atrybutami:




- **sąsiedzi**: lista *sąsiadujących* duszków
- **inne duszki**: lista duszków z wyjątkiem mnie
- **klony**: lista moich *tymczasowych* klonów
- **inne klony**: lista mojego *tymczasowego* rodzeństwa
- **części**: lista duszków dla których ten jest **kotwicą**
- **potomstwo**: lista moich klonów, tymczasowych i trwałych
- **kostiumy**: lista kostiumów duszka
- **dźwięki**: lista dźwięków duszka

Inne nie są listami:

- **sam**: ten duszek
- **kotwica**: duszek, którego jestem (zagnieżdżoną) częścią
- **scena**: scena, która też jest pierwszej klasy, jak duszek
- **rodzic**: duszek, którego jestem klonem
- **nazwa**: moja nazwa (taka jak nazwa rodzica, jeśli jestem tymczasowy)
- **wiszący?**: prawda, jeśli jestem częścią i nie poruszam się synchroniczne

Nie przywiązuj się do tych pogiętych, które zapewne ulegną zmianie:

- **obrót x**, **obrót y**: to samo co **pozycja X**, **pozycja Y**
- **x środka**, **y środka**: zaokrąglona pozycja x i y środka prostokąta duszka

# VIII Programowanie obiektowe i procedury

Idea programowania obiektowego jest często nauczana w sposób, który sprawia wrażenie, że potrzebny byłby specjalny język programowania obiektowego. W rzeczywistości każdy język z pierwszorzędnymi procedurami i leksykalnym polem działania umożliwia jawne zaimplementowanie obiektów; jest to przydatne doświadczenie, pomocne w demistyfikacji obiektów.

Główną ideą tej implementacji jest to, że obiekt jest reprezentowany jako *procedura wysyłania*, która pobiera komunikat jako dane wejściowe i przekazuje odpowiednią metodę. W tym rozdziale rozpoczynamy od gołego przykładu, aby pokazać, jak działa stan lokalny i ubrać go, budując pełną implementację klasy / instancji i prototypowania OOP.

## A Stan lokalny ze zmiennymi skryptu

Ten skrypt implementuje obiekt *klasę*, typ obiektu, czyli klasę licznik. W tej pierwszej wersji uproszczonej jest tylko jedna metoda, więc nie jest konieczne jawne przekazywanie komunikatów. Kiedy wywoływany jest blok **utwórz licznik**, przekazuje on procedurę, w postaci skryptu w obwiedni w jego ciele. Ta procedura implementuje określony obiekt licznika, *instancję* klasy licznika. Po wywołaniu instancja licznik zwiększa i przekazuje swoją zmienną licz. Każdy licznik ma swoją lokalną zmienną licz:



Ten przykład spłaci staranne studiowanie, ponieważ nie jest oczywiste, dlaczego każda instancja ma osobne licz. Z punktu widzenia procedury **utwórz licznik** każde wywołanie powoduje utworzenie nowej zmiennej ***licz***. Zwykle takie *zmienne skryptu* są tymczasowe, kończą istnienie po zakończeniu skryptu. Ale ta jest wyjątkowa, ponieważ **utwórz licznik** zwraca inny skrypt, który odwołuje się do zmiennej ***licz***, więc pozostaje aktywny.

(Blok **zmienne skryptu** tworzy zmienne lokalne dla skryptu, można go używać w obszarze skryptów duszka lub w Edytorze bloków. Zmienne skryptu można „wyeksportować”, wykorzystując je w przekazywanej procedurze, tak jak tutaj).

W tym podejściu do OOP reprezentujemy zarówno klasy, jak i instancje jako procedury. Blok **utwórz licznik** reprezentuje klasę, podczas gdy każda instancja jest reprezentowana przez bezimienny skrypt tworzony za każdym razem, gdy wywoływany jest **utwórz licznik**. Zmienne skryptu utworzone wewnątrz bloku **utwórz licznik**, ale poza obwiednią, są *zmiennymi instancji* należącymi do określonego licznika.

## B Wiadomości i procedury wysyłania

W powyższej uproszczonej klasie istnieje tylko jedna metoda, więc nie ma żadnych wiadomości; wystarczy wywołać instancję, aby wykonać jej jedyną metodę. Oto bardziej dopracowana wersja, która wykorzystuje przekazywanie wiadomości:

Ponownie, blok **stwórz licznik** reprezentuje klasę **licznik** i ponownie skrypt tworzy lokalną zmienną ***licz*** za każdym razem, gdy jest wywoływany. Duża zewnętrzna obwiednia reprezentuje instancję. Jest to *procedura wysyłania*: pobiera wiadomość (tylko słowo tekstu) jako dane wejściowe i przekazuje metodę. Dwie mniejsze obwiednie to metody. Górna jest metodą **następny**; dolna to metoda **reset**. Ta ostatnia wymaga wejścia, nazwanego **wartość**.

We wcześniejszej wersji wywołanie instancji wykonało całą pracę. W tej wersji wywołanie instancji daje dostęp do metody, która musi zostać wywołana, aby zakończyć pracę. Możemy zbudować blok do wykonywania obu wywołań procedur w jednym kroku:

Blok **poproś** wymaga dwóch wejść: obiektu i wiadomości. Przyjmuje również opcjonalnie dodatkowe wejścia, które Snap*!* umieszcza na liście; ta lista nazywa się ***arg*** wewnątrz bloku. **Poproś** ma dwa zagnieżdżone bloki **wywołaj**. Wewnętrzny wywołuje obiekt, tj. procedurę wysyłania. Procedura wysyłania zawsze ma dokładnie jedno wejście, a mianowicie ***wiadomość***. Przekazuje metodę, która może przyjąć dowolną liczbę danych wejściowych; zauważ, że to jest sytuacja, w której upuszczamy listę wartości na

strzałki wielu wejść (w zewnętrznym bloku wywołania). Zauważ też, że jest to jeden z nielicznych przypadków, w których musimy zdjąć obwiednię wewnętrznego bloku **wywołaj**, którego wartość po wywołaniu podaje metodę.

```
ustaw licznik1 na utwórz licznik
powiedz poproś licznik1 następny przez 2 s        (1)
powiedz poproś licznik1 następny przez 2 s        (2)
powiedz poproś licznik1 reset 4 przez 2 s         (4)
powiedz poproś licznik1 następny przez 2 s        (5)
```

## C Dziedziczenie przez delegowanie

Nasze obiekty mają teraz zmienne stanu lokalnego i przekazywanie wiadomości. A co z dziedziczeniem? Możemy zapewnić tę możliwość za pomocą techniki *delegowania*. Każda instancja klasy potomnej zawiera instancję klasy nadrzędnej i po prostu przekazuje wiadomości, których nie chce specyfikować:

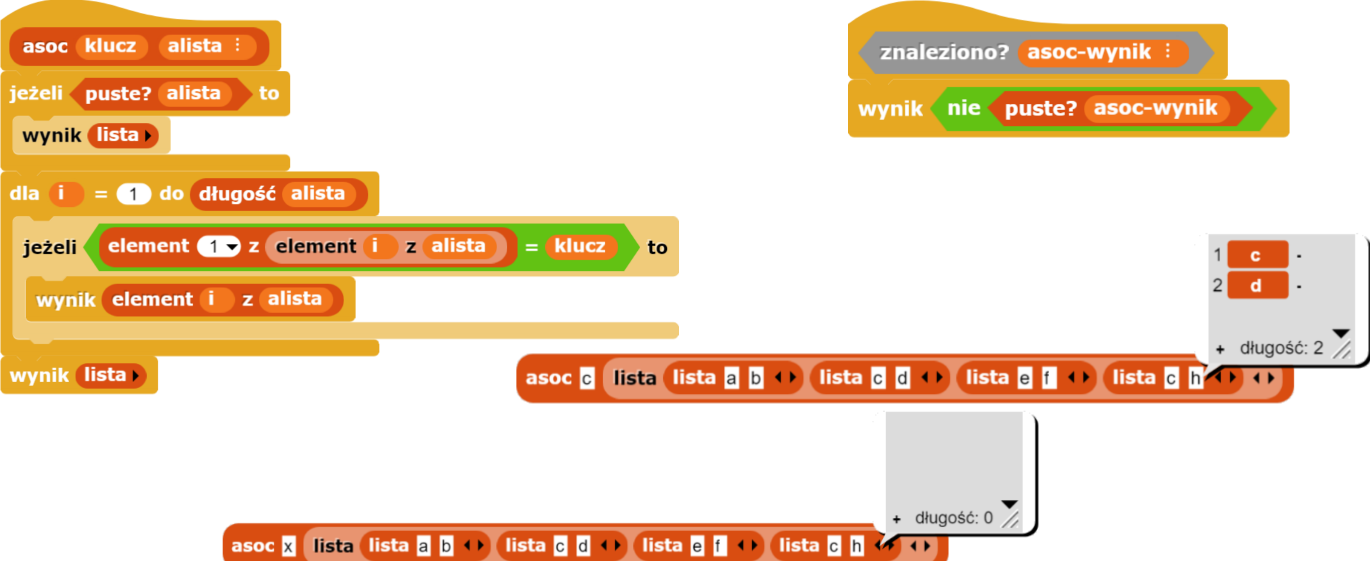
```
stwórz brzęczyk
zmienne skryptu licznik
ustaw licznik na stwórz licznik
wynik
    jeżeli wiadomość = następny to
        wynik
            zmienne skryptu temp
            ustaw temp na poproś licznik następny
            jeżeli temp modulo 7 = 0 to
                wynik drrryń
            w przeciwnym razie
                wynik temp
    wynik wywołaj licznik z parametrami wiadomość
    nazwy parametrów: wiadomość
```

Ten skrypt implementuje klasę **brzęczyk**, która jest potomkiem **licznika**. Zamiast lokalnej zmiennej stanu ***licz*** (liczby) każdy brzęczyk ma **licznik** (obiekt) jako lokalną zmienną stanu. Klasa specjalizuje się w metodzie **następny**, dając w wyniku to, co przekazuje licznik, chyba że wynik jest podzielny przez 7, w którym to przypadku raportuje „***drrryń***". (Tak, powinien również sprawdzić występowanie cyfry 7 w liczbie, ale ten kod jest już wystarczająco skomplikowany.). Jeśli komunikat jest inny niż **następny**, jak np. **reset**, brzęczyk po prostu wywołuje procedurę wysyłania **licznik**a. Tak więc licznik obsługuje każdą wiadomość, której brzęczyk nie obsługuje jawnie. (Zauważ, że w przypadku nie-**następny** wywołujemy licznik, bez **poproś**, ponieważ chcemy zgłosić metodę, a nie wartość, którą wiadomość przekazuje). Tak więc, jeśli wywołamy **poproś brzęczyk** o **reset** do wartości podzielnej przez 7, to w końcu przekaże tę liczbę, a nie „***drrryń***".

## D Implementacja prototypowania OOP

W powyższym systemie klasy / instancji konieczne jest zaprojektowanie pełnego zachowania klasy, zanim będzie można wykonać dowolną instancję klasy. Jest to dobre w przypadku projektowania metodą z góry do dołu (top-down), ale nie nadaje się do eksperymentowania. Tutaj szkicujemy implementację systemu prototypowania OOP: tworzysz obiekt, majstrujesz przy nim, tworzysz jego klony i ciągle majstrujesz. Wszelkie zmiany dokonane w rodzicu są dziedziczone przez jego potomków. W efekcie, tym pierwszym obiektem jest zarówno klasa, jak i instancja klasy. W poniższej implementacji obiekty potomne podzielają własności (metody i zmienne lokalne) swojego rodzica, chyba że potomek zmieni jakąś własność, wtedy potomek uzyskuje jej własną kopię. (Jeśli potomek chce coś zmienić w całej rodzinie, musi poprosić rodzica).

Ponieważ chcemy dynamicznie tworzyć i usuwać własności, nie będziemy używać zmiennych Snap*!* do przechowywania zmiennych lub metod obiektu. Zamiast tego, każdy obiekt ma dwie *tabele*, zwane **metodami** i **danymi**, z których każda jest listą asocjacji: listą dwuelementowych list, w której każda z tych ostatnich zawiera *klucz* i odpowiadającą mu *wartość*. Wprowadzimy procedurę wyszukiwania, aby zlokalizować parę klucz-wartość odpowiadającą danemu kluczowi w danej tabeli.




Są też komendy do wstawiania i usuwania wejść:

```
wstaw nazwa wartość do tabela
  zmienne skryptu para-kw
  ustaw para-kw na asoc nazwa tabela
  jeżeli znaleziono? para-kw to
    zamień element 2 z para-kw na wartość
  w przeciwnym razie
    wstaw lista nazwa wartość na pozycji 1 do tabela

usuń klucz z tabela
  dla indeks = 1 do długość tabela
    jeżeli element 1 z element indeks z tabela = klucz to
      usuń indeks z tabela
      zatrzymaj ten blok
```

Podobnie jak w wersji klasa / instancja, obiekt jest reprezentowany jako procedura wysyłania, która pobiera komunikat jako dane wejściowe i przekazuje odpowiednią metodę. Gdy obiekt otrzyma wiadomość, najpierw szuka tego słowa kluczowego w swojej tabeli **metod**. Jeśli klucz zostanie znaleziony, odpowiednia wartość jest pożądaną metodą. Jeśli nie, obiekt zagląda do swojej tabeli **danych**. Jeśli wartość zostanie tam znaleziona, to co przekazuje obiekt, nie jest tą wartością, ale raczej funkcją metody, która po wywołaniu przekaże wartość. Co znaczy, że to co obiekt przekazuje jest zawsze metodą.

Jeśli obiekt nie ma ani metody, ani danej o żądanej nazwie, ale ma rodzica, wówczas rodzic (to znaczy procedura nadająca rodzica) jest wywoływany z wiadomością jako jego daną wejściową. W końcu albo znalezione zostanie dopasowanie albo obiekt bez rodzica; ten drugi przypadek jest błędem, co oznacza, że użytkownik wysłał obiektowi wiadomość nie znajdującą się w jego repertuarze.

Wiadomości mogą przyjmować dowolną liczbę wejść, tak jak w systemie klasy / instancji, ale w wersji prototypowej każda metoda automatycznie pobiera obiekt, do którego wiadomość została pierwotnie wysłana jako dodatkowe pierwsze wejście. Musimy to tak zrobić, aby gdy znajdzie się metodę w rodzicu (lub dziadku itp.) pierwotnego odbiorcy, a ta metoda odnosi się do zmiennej lub metody, to jednak użyta została zmienna lub metoda potomka, jeśli potomek ma swoją własną wersję.

Blok **klon z** obok pobiera obiekt jako wejście i tworzy obiekt potomny. Należy go uznać za wewnętrzną część wdrożenia; preferowanym sposobem utworzenia obiektu potomnego jest wysłanie do tego obiektu wiadomości **klon**.

```
klon z rodzic
zmienne skryptu sam metody dane
ustaw metody na lista
ustaw dane na lista
wstaw ustaw (wstaw nazwa wartość do dane  nazwy parametrów: sam nazwa wartość) do metody
wstaw metoda (wstaw nazwa wartość do metody  nazwy parametrów: sam nazwa wartość) do metody
wstaw klon (wynik klon z sam  nazwy parametrów: sam) do metody
wstaw usuń-zm (usuń nazwa z dane  nazwy parametrów: sam nazwa) do metody
wstaw usuń-met (usuń nazwa z metody  nazwy parametrów: sam nazwa) do metody
wstaw rodzic rodzic do dane
ustaw sam na
    zmienne skryptu para-kw
    ustaw para-kw na asoc wiadomość metody
    jeżeli znaleziono? para-kw to
        wynik element 2 z para-kw
    ustaw para-kw na asoc wiadomość dane
    jeżeli znaleziono? para-kw to
        wynik (element 2 z para-kw  nazwy parametrów: sam)
    jeżeli jest rodzic typu funkcja ? to
        wynik wywołaj rodzic z parametrami wiadomość
    wynik lista
  nazwy parametrów: wiadomość
wynik sam
```

Każdy obiekt jest tworzony za pomocą predefiniowanych metod: **ustaw**, **metoda**, **usuń-zm**, **usuń-met** i **klon**. Ma jedną predefiniowaną zmienną, **rodzic**. Obiekty bez rodzica są tworzone przez wywołanie **nowy obiekt**:

```
nowy obiekt
wynik (klon z (lista ▸))
```

Tak jak poprzednio, tworzymy procedury wywoływania procedury wysyłania obiektu, a następnie wywołujemy metodę. Ale w tej wersji podajemy żądany obiekt jako pierwsze wejście metody. Tworzymy jedną procedurę dla metod poleceń i jedną dla metod funkcji:

```
poproś (obiekt λ) (wiadomość) (arg...)
wynik (wywołaj (wywołaj (obiekt) z parametrami (wiadomość) ◂▸)
       parametr - lista: (wstaw (obiekt) przed (arg)))

powiem (obiekt λ) (wiadomość) (arg...)
uruchom (wywołaj (obiekt) z parametrami (wiadomość) ◂▸)
parametr - lista: (wstaw (obiekt) przed (arg))
```

(Pamiętaj, że wariant „**parametr-lista**" bloków **wywołaj** i **uruchom** jest tworzony przez przeciągnięcie wyrażenia wejściowego na strzałki, a nie na pole wejściowe).

Poniższy skrypt pokazuje, w jaki sposób ten system prototypowania może być używany do tworzenia liczników. Zaczynamy od jednego prototypowego licznika, zwanego **licznik1**. Licznik ten uruchamiamy kilka razy, po czym tworzymy potomny **licznik2** i nadajemy mu własną zmienną **licz**, ale nie własną zmienną **total**. Metoda **nast** zawsze ustawia zmienną **total licznika1**, która w ten sposób trzyma pełną liczbę razy, gdy którykolwiek licznik jest inkrementowany. Uruchomienie tego skryptu powinno [**powiedz**] i (**pomyśl**) następujące listy:

**[1 1] [2 2] [3 3] [4 4] (1 5) (2 6) (3 7) [5 8] [6 9] [7 10] [8 11]**

```
ustaw licznik1 na (nowy obiekt)
powiem (licznik1) [ustaw] [licz] [0] ◂▸
powiem (licznik1) [ustaw] [total] [0] ◂▸
powiem (licznik1) [metoda] [nast] (
    powiem (sam) [ustaw] [licz] ((poproś (sam) [licz] ▸) + (1)) ◂▸
    powiem (licznik1) [ustaw] [total] ((poproś (sam) [total] ▸) + (1)) ◂▸
    wynik (lista (poproś (sam) [licz] ▸) (poproś (sam) [total] ▸) ◂▸)
  nazwy parametrów: (sam) ◂▸ ) ◂▸
powtórz (4) razy
    powiedz (poproś (licznik1) [nast] ▸) przez (2) s
ustaw licznik2 na (poproś (licznik1) [klon] ▸)
powiem (licznik2) [ustaw] [licz] [0] ◂▸
powtórz (3) razy
    pomyśl (poproś (licznik2) [nast] ▸) przez (2) s
powtórz (4) razy
    powiedz (poproś (licznik1) [nast] ▸) przez (2) s
```

[*Tu niestety mam gdzieś na poziome licznik2 robaczka i nie mogę go wyeliminować, przyp WtK*]

# IX Świat zewnętrzny

To o czym mówiliśmy do tej pory było przeznaczone dla projektów, które są realizowane w całości na ekranie komputera. Ale możesz chcieć pisać programy, które wchodzą w interakcje z urządzeniami fizycznymi (czujnikami lub robotami) lub z siecią World Wide Web. Do tych celów Snap*!* ma jeden pierwotny blok:

url snap.berkeley.edu

To może wydawać się niewystarczające, ale w rzeczywistości da się wykorzystać do realizacji tego co chcemy uzyskać.

## A World Wide Web

Dane wejściowe do bloku **url** są adresem URL (Uniform Resource Locator) strony internetowej. Blok przekazuje *bez interpretacji* treść odpowiedzi serwera WWW (bez nagłówka http). Oznacza to, że w większości przypadków odpowiedzią jest opis strony w HTML (HyperText Markup Language). Często, zwłaszcza w komercyjnych witrynach internetowych, rzeczywiste informacje, które próbujesz znaleźć na stronie, znajdują się pod innym adresem URL zawartym w przekazywanym kodzie HTML. Strona sieci Web jest zazwyczaj bardzo długim ciągiem tekstowym, więc można wykorzystać pierwotny blok **podziel**, aby uzyskać tekst w postaci możliwej do ogarnięcia, a mianowicie jako listę linii:

podziel url snap.berkeley.edu na linia

```
1  <!DOCTYPE html>
2  <!-- Woot, not HTML2! -->
3  <html><head>
4  <meta http-equiv="content-type" content="text/html; c
5   <title>Snap! (Build Your Own Blocks) 4.1</title>
6   <meta coding="utf-8">
7  <link rel="shortcut icon" href="snapsource/favicon.ico">
8       <link href="liam4/bootstrap.css" rel="stylesheet">
9   <link rel="stylesheet" href="liam4/index.css">
10
11 <script type="text/javascript">
12
13  var _gaq = _gaq || [];
14  _gaq.push(['_setAccount', 'UA-30925559-2']);
15  _gaq.push(['_trackPageview']);
16
17  (function() {
18   var ga = document.createElement('script'); ga.type = 't
19   ga.src = ('https:' == document.location.protocol ? 'http
20 '.google-analytics.com/ga.js';
21   var s = document.getElementsByTagName('script')[0];
22  })();
23
długość: 384
```

Drugim wejściem do bloku **podziel** jest znak, który ma być użyty do rozdzielenia ciągu tekstowego na listę linii lub jeden z zestawu typowych przypadków (np. **linia**, która powoduje podział na znaku powrotu karetki i / lub znaku nowego wiersza).

To może być dobre miejsce na przypomnienie, że oglądający listę przeglądają tylko 100 elementów naraz. Strzałka w dół w prawym dolnym rogu dymka na obrazku rozwija menu z kolejnymi setkami elementów. (Może się to wydawać niepotrzebne, ponieważ pasek przewijania powinien zezwalać na dowolną liczbę elementów, ale robienie tego w ten sposób powoduje, że Snap*!* jest znacznie szybszy). W widoku tabeli zawarta jest cała lista.

Jeśli w danych wejściowych do bloku **url** podasz nazwę protokołu (np. **http: //** lub **https: //**), użyty zostanie ten protokół. Jeśli nie, blok najpierw próbuje **HTTPS**, a następnie, jeśli to się nie powiedzie, **HTTP**.

Ograniczenia związane z bezpieczeństwem w JavaScript ograniczają możliwość strony do zainicjowania komunikacji z inną witryną. Istnieje oficjalne obejście tego ograniczenia, zwane protokołem CORS (Cross Source Resource Sharing), ale strona docelowa musi explicite pozwalać na otwieranie snap.berkeley.edu, lecz oczywiście większość z nich nie ma takiego wpisu. Aby obejść ten problem, można używać witryn innych firm ("proxy cors"), które nie są ograniczane przez JavaScript i które przesyłają żądania.

## B Urządzenia sprzętowe

Kolejne ograniczenie bezpieczeństwa JavaScript nie pozwala aby Snap*!* miał bezpośredni dostęp do urządzeń podłączonych do twojego komputera, takich jak czujniki i roboty. (Urządzenia mobilne, takie jak smartfony, mogą również mieć wbudowane przydatne urządzenia, takie jak akcelerometry i odbiorniki GPS). Blok **url** jest również używany do łączenia Snap! z tymi zewnętrznymi urządzeniami.

Pomysł polega na uruchomieniu oddzielnego programu, który łączy się z urządzeniem i udostępnia lokalny serwer HTTP, który Snap*!* może wykorzystywać do wysyłania żądań do urządzenia. *W przeciwieństwie do Snap! te programy mają dostęp do wszystkiego na komputerze, więc musisz zaufać autorowi oprogramowania!* Nasza strona internetowa `snap.berkeley.edu` zawiera linki do sterowników dla kilku urządzeń, w tym, Lego NXT, Finch, Hummingbird i Parallax S2; roboty Nintendo Wiimote i Leap Motion, mikrokomputer Arduino i Super-Awesome Sylvia's Water Colour Bot. Ta sama technika serwerowa może być wykorzystana do uzyskania dostępu do bibliotek oprogramowania innych firm, takich jak pakiet syntezy mowy, do którego odsyłacz jest na naszej witrynie internetowej.

Zainstalowanie większości z tych pakietów wymaga trochę wiedzy; są linki do repozytoriów kodu źródłowego. Ta sytuacja będzie poprawiać się z czasem.

## C Data i czas

Blok **obecnie** w palecie Czujniki może być użyty do uzyskania aktualnej daty lub czasu. Każde wywołanie tego bloku przekazuje jeden składnik daty lub czasu, więc prawdopodobnie połączymy kilka wywołań, na przykład:

dla amerykanów, lub jak tu:

dla Europejczyków.

# X Kontynuacje

Bloki są zwykle używane w skrypcie. *Kontynuacja* bloku w obrębie określonego skryptu jest częścią obliczeń, które należy wykonać po tym, jak blok wykona swoje zadanie. Kontynuację można przedstawić jako skrypt w obwiedni. Kontynuacje są zawsze częścią interpretacji dowolnego programu w dowolnym języku, ale zazwyczaj te kontynuacje są ukryte w strukturach danych interpretera lub kompilatora języka. Jawne tworzenie kontynuacji jest zaawansowaną, ale wszechstronną techniką programowania, która pozwala użytkownikom tworzyć struktury kontrolne, takie jak nielokalne wyjście i wielowątkowość.

W najprostszym przypadku kontynuacja bloku polecenia może być po prostu częścią skryptu po bloku. Na przykład w skrypcie

```
przyłóż pisak
przesuń o 50 kroków
obróć ↻ o 30 stopni
przesuń o 100 kroków
obróć ↻ o 20 stopni
przesuń o 40 kroków
podnieś pisak
```

Kontynuacją bloku przesuń o 100 kroków jest

```
obróć ↻ o 20 stopni
przesuń o 40 kroków
podnieś pisak
```

Ale niektóre sytuacje są bardziej skomplikowane. Na przykład, co jest kontynuacją przesuń o 100 kroków w poniższym skrypcie?

```
powtórz 4 razy
    przesuń o 100 kroków
    obróć ↻ o 90 stopni
```

To podchwytliwe pytanie; blok **przesuń** jest uruchamiany cztery razy i za każdym razem ma inną kontynuację. Za pierwszym razem, jego kontynuacja to

```
obróć ↻ o 90 stopni
powtórz 3 razy
    przesuń o 100 kroków
    obróć ↻ o 90 stopni
```

Zauważ, że w rzeczywistym skrypcie nie ma bloku z **powtórz 3**, ale kontynuacja musi przedstawić fakt, że pętla ma jeszcze do przejścia trzy powtórzenia. Za czwartym razem, kontynuacja to tylko

```
obróć ↻ o 90 stopni
```

Liczy się nie to, co fizycznie znajduje się poniżej bloku w skrypcie, ale jakie obliczenia pozostaną do wykonania.

Gdy blok jest używany wewnątrz niestandardowego bloku, jego kontynuacja może obejmować części więcej niż jednego skryptu. Na przykład, jeśli utworzymy niestandardowy blok kwadrat

I użyjemy tego bloku w skrypcie:

wtedy kontynuacją pierwszego użycia bloku przesuń o 100 kroków jest:

w której część pochodzi z wnętrza bloku **kwadrat**, a część pochodzi ze skryptu wywołującego **kwadrat**. Niemniej jednak, gdy przedstawiamy kontynuację, pokazujemy tylko część z bieżącego skryptu.

Kontynuacja bloku poleceń, jak widzieliśmy, jest prostym skryptem bez pól wejściowych. Ale kontynuacja bloku funkcji musi coś zrobić z wartością przekazywaną przez blok, więc przyjmuje tę wartość jako dane wejściowe. Na przykład w skrypcie

kontynuacją bloku **3+4** jest

Oczywiście nazwa **obliczone** na obrazku jest arbitralna; może zostać użyta dowolna nazwa lub może nie być jej wcale, gdy użyjemy notacji z pustymi polami wejścia do wstawiania danych.

## A Styl przekazywania kontynuacji

Podobnie jak wszystkie języki programowania, Snap*!* oblicza układy zagnieżdżonych funkcji od środka. Na przykład w wyrażeniu (3 × (4 + 5)) Snap*!* najpierw dodaje 4 i 5, a następnie mnoży 3 przez tę sumę. Często oznacza to, że kolejność, w jakiej wykonywane są operacje, jest odwrotna od kolejności, w jakiej występują w wyrażeniu: Kiedy czytasz powyższe wyrażenie, mówisz „razy” zanim powiesz „plus”. W języku polskim, zamiast powiedzieć „trzy razy cztery plus pięć”, co faktycznie sprawia, że kolejność operacji jest niejednoznaczna, można powiedzieć, „weź sumę czterech i pięciu, a następnie weź iloczyn trzech i tej sumy”. Brzmi to nieco niezręcznie, ale ma zaletę umieszczania operacji w takiej kolejności, w jakiej są wykonywane.

To może wydawać się przesadą przy prostym wyrażeniu, ale przypuśćmy, że próbujesz przekazać wyrażenie

silnia 3 × silnia 2 + 2 + 5

przyjacielowi przez telefon. Jeśli powiesz „silnia trzy razy silnia z dwóch plus dwa plus pięć”, może oznaczać dowolne wyrażenie z poniższych:

silnia 3 × silnia 2 + 2 + 5

silnia 3 × silnia 2 + 2 + 5

silnia 3 × silnia 2 + 2 + 5

silnia 3 × silnia 2 + 2 + 5

Czy nie byłoby lepiej powiedzieć: „Dodaj dwa i dwa, weź z tego silnię, dodaj do niej pięć, pomnóż to przez trzy i weź silnię wyniku”? Możemy dokonać podobnego uporządkowania wyrażenia, jeśli najpierw zdefiniujemy wersje wszystkich funkcji, które kontynuują swoją działalność jako jawne wejścia. Na poniższym obrazku zauważ, że nowe bloki są *komendami*, a nie funkcjami.

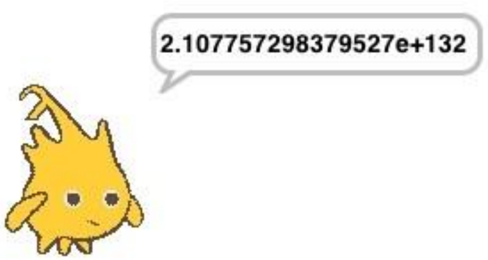
```
+dodaj+ a + b + kont λ +
uruchom kont z parametrami (a + b)

+odejmij+od+ a + b + kont λ +
uruchom kont z parametrami (a − b)

+pomnóż+ a + b + kont λ +
uruchom kont z parametrami (a × b)

+równe?+ a + b + kont λ +
uruchom kont z parametrami (a = b)

+silnia+ n + kont λ +
równe? n 0
    jeżeli wynRów to
        uruchom kont z parametrami 1
    w przeciwnym razie
        odejmij od n 1
            silnia wynOdejm
                pomnóż n wynSiln kont  nazwy parametrów: wynSiln
            nazwy parametrów: wynOdejm
nazwy parametrów: wynRów
```

Możemy sprawdzić, że te bloki dają wynik zgodny z oczekiwaniami:

Nasze wyrażenie może być teraz zaprezentowane jako:

```
dodaj 2 2
  silnia pierwSum
    dodaj pierwSil 5
      pomnóż 3 drugSum
        silnia iloczyn  powiedz drugSil  nazwy parametrów: drugSil
        nazwy parametrów: iloczyn
      nazwy parametrów: drugSum
    nazwy parametrów: pierwSil
  nazwy parametrów: pierwSum
```

Czy jeśli przeczytasz to od góry do dołu, to będzie: „Dodaj dwa i dwa, weź z tego silnię, dodaj do niej pięć, pomnóż to przez trzy i weź silnię wyniku”? Właśnie tego chcieliśmy. Ten sposób działania, w którym każdy blok jest poleceniem, które przyjmuje kontynuację jako jedno z wejść, nazywany jest stylem przekazywania kontynuacji (*continuation-passing style CPS*). Okej, wygląda okropnie, ale ma subtelne zalety. Jedną z nich jest to, że każdy skrypt ma długość tylko jednego bloku (z resztą pracy schowaną w kontynuacji przekazywanej do tego bloku), więc każdy blok nie musi pamiętać, co jeszcze zrobić - w słownictwie tego rozdziału kontynuacja (niejawna) każdego bloku jest pusta. Zamiast zwykłego obrazu rekurencji, z grupą małych ludzi czekających na siebie, z CPS każda mała osoba przekazuje problem następnej i idzie na plażę, więc w każdym momencie jest aktywna tylko jedna mała osoba. W tym przykładzie zaczynamy od Darka specjalisty od **dodaj**, który wylicza wartość 4, a następnie przekazuje resztę problemu Sylwii, specjalistce od **silnia**. Ona oblicza wartość 24, następnie przekazuje problem Dorocie, kolejnej specjalistce od **dodaj**, która oblicza 29. I tak dalej, aż w końcu Paweł, specjalista od **powiedz**, mówi wartość $2.107757298379527 \times 10^{132}$, która jest bardzo dużą liczbą!




Wróćmy do definicji tych bloków. Takie jak **dodaj**, które odpowiadają pierwotnym funkcjom są proste; po prostu wywołują funkcję, a następnie wywołują kontynuację z otrzymanym wynikiem. Ale definicja **silni** jest bardziej interesująca. Nie tylko wywołuje nasz oryginalną funkcję **silnia** i wysyła wynik do jej kontynuacji. CPS jest również używany wewnątrz **silni**! Mówi: „Sprawdź, czy moje dane wejściowe są zerowe. Wyślij wynik (prawda lub fałsz) do **jeżeli**. Jeśli wynikiem jest **prawda**, wywołaj moją kontynuację z wartością 1. W przeciwnym razie odejmij 1 od mojego wejścia. Wyślij wynik do **silni**, z kontynuacją, która mnoży silnię mniejszej liczby przez moje oryginalne dane wejściowe. Na koniec wywołaj moją kontynuację z iloczynem”. Możesz użyć CPS, aby rozwinąć nawet najbardziej skomplikowane rozgałęzione wywołania rekurencyjne.

Przy okazji, powyżej trochę oszukałem. Blok **jeżeli / w przeciwnym razie** powinien również używać CPS; powinien przyjmować jedno wejście **prawda / fałsz** i dwie kontynuacje. Powinien przejść do jednej lub drugiej kontynuacji w zależności od wartości wejścia. Ale w rzeczywistości bloki w kształcie litery C (lub E-kształtne, jak **jeżeli / w przeciwnym razie**) naprawdę używają CPS, ponieważ pośrednio zawijają pierścienie wokół podskryptów w swoich gałęziach. Sprawdź, czy możesz zrobić jawny blok CPS, **jeżeli / w przeciwnym razie**.

## B Wywołaj/Uruchom z kontynuacją

Aby użyć jawnego stylu przekazywania kontynuacji, musieliśmy zdefiniować specjalne wersje wszystkich funkcji, **dodaj** i tak dalej. Snap*!* zapewnia pierwotny mechanizm przechwytywania kontynuacji, gdy jest to konieczne, bez korzystania z przechodzenia kontynuacji przez cały projekt.

Oto klasyczny przykład. Chcemy napisać blok rekurencyjny, który pobiera listę liczb jako dane wejściowe i przekazuje iloczyn wszystkich liczb:

```
+iloczyn+ listaLiczb ⋮ +
jeżeli puste? listaLiczb to
    wynik 1
wynik element 1 z listaLiczb × iloczyn bez pierwszego z listaLiczb
```

Ale możemy poprawić wydajność tego bloku, w przypadku listy zawierającej zero; jak tylko zobaczymy zero, wiemy, że cały iloczyn wynosi zero.

```
+iloczyn+ listaLiczb ⋮ +
jeżeli puste? listaLiczb to
    wynik 1
jeżeli element 1 z listaLiczb = 0 to
    wynik 0
wynik element 1 z listaLiczb × iloczyn bez pierwszego z listaLiczb
```

Ale to nie jest tak efektywne, jak mogłoby się wydawać. Rozważmy na przykład listę 1,2,3,0, 4,5. Znajdujemy zero w trzecim rekurencyjnym wywołaniu (łącznie czwarte wywołanie), jako pierwszy element podlisty 0,4,5. Jaka jest kontynuacja bloku **wynik 0**? To jest:

```
wynik 1 × 2 × 3 × rezultat
nazwy parametrów: rezultat ◂▸
```

Chociaż już wiemy, że **rezultat** jest zero, będziemy wykonywać trzy niepotrzebne mnożenia, podczas gdy zwijania wywołań rekurencyjnych.

Możemy to ulepszyć, przechwytując kontynuację wywołania najwyższego poziomu do **iloczyn**:

```
+iloczyn-pomoc+ ListaL ⋮ + kont λ +
jeżeli puste? ListaL to
    wynik 1
jeżeli element 1 z ListaL = 0 to
    uruchom kont z parametrami 0 ◂▸
wynik element 1 z ListaL ×
      iloczyn-pomoc bez pierwszego z ListaL kont
```

```
+iloczyn+ listaLiczb ⋮ +
wynik
    wywołaj [ wynik iloczyn-pomoc listaLiczb iloczynKont
              nazwy parametrów: iloczynKont ◂▸ ] z kontynuacją
```

Blok wywołaj z kontynuacją przyjmuje jako wejście skrypt z jednym wejściem, jak pokazano na przykładzie bloku **iloczyn**. Wywołuje ten skrypt jako kontynuację samego bloku **wywołaj … z kontynuacją** jako jego wejścia. W tym przypadku ta kontynuacja jest

wynik rezultat nazwy parametrów: rezultat

przekazywana do skryptu o nazwie **iloczyn**. Jeśli lista wejściowa nie zawiera wartości zero, to nic nie będzie się działo z kontynuacją i ta wersja działa tak jak oryginalny **iloczyn**. Ale jeśli lista wejściowa wynosi 1,2,3,0, 4,5, to są wykonywane trzy wywołania rekurencyjne, znalezione jest zero, a **iloczyn-pomoc** *uruchamia kontynuację* z wejściem 0. Kontynuacja natychmiast zgłasza to 0 do wywołującego **iloczynu**, *bez* odwijania wszystkich wywołań rekurencyjnych i bez niepotrzebnych mnożeń.

Mógłbym napisać **iloczyn** nieco prościej używając obwiedni funkcji zamiast obwiedni komendy:

+iloczyn+ listaLiczb : +
wynik wywołaj iloczyn-pomoc listaLiczb iloczynKont nazwy parametrów: iloczynKont z kontynuacją

ale zwyczajowo używa się skryptu do reprezentowania danych wejściowych dla wywołania z kontynuacją, ponieważ bardzo często dane wejście przyjmują formę

ustaw zmiennaGlobalna na kontynuacja
wynik ...
nazwy parametrów: kontynuacja

aby kontynuacja została zapisana na stałe i mogła być wywołana z dowolnego miejsca w projekcie. Dlatego właśnie pole wejściowe w **wywołaj … z kontynuacją** ma obwiednię komendy zamiast obwiedni funkcji.

Kontynuacje pierwszej klasy są eksperymentalną funkcją Snap*!* i mają wiele znanych ograniczeń. Po pierwsze, nie działa uruchamianie pustej kontynuacji; to znaczy, jeśli blok **uruchom z kontynuacją** jest ostatni w skrypcie, kontynuacja zakończy się niepowodzeniem. Również wyświetlanie kontynuacji funkcji pokazuje tylko pojedynczy blok, w którym **wywołaj z kontynuacją** jest wejściem.

### Wyjście nielokalne

Wiele języków programowania ma polecenie przerwania (**break**), które może być użyte wewnątrz konstrukcji pętli, takiej jak **powtórz**, aby wcześniej zakończyć powtarzanie. Korzystając z kontynuacji pierwszej klasy, możemy uogólnić ten mechanizm, żeby umożliwić nielokalne wyjście nawet w obrębie bloku wywoływanego z wnętrza pętli lub poprzez kilka poziomów zagnieżdżonych pętli:[11]

Zmienna widoczna dla wywołania (upvar) **break** jest wartością kontynuacji, którą można wywołać z dowolnego miejsca w programie, aby natychmiast przejść do tego, co jest w skrypcie po bloku **złap**. Oto przykład z dwoma zagnieżdżonymi wywołaniami **złap**, ze zmienioną nazwą zmiennej upvar w zewnętrznym:

Jak widać, duszek powie 1, następnie 2, a następnie 3, a potem po wyjściu z obu zagnieżdżonych poleceń **złap** pomyśli „Hmm”. Jeśli w bloku **uruchom** zamiast **zewn** użyta zostanie zmienna **break**, to duszek powie: 1, 2, 3 i „Cześć!”, zanim pomyśli „Hmm”.

Istnieją odpowiednie bloki **złap** i **rzuć** dla funkcji. Blok funkcji **złap**, zamiast gniazda w kształcie litery C pobiera wyrażenie jako wejście. Ale blok **rzuć** jest komendą; nie przekazuje wartości do własnej kontynuacji, zamiast tego podaje wartość (którą pobiera jako dodatkowe wejście, oprócz zmiennej tag **złap**) do *kontynuacji* odpowiedniego bloku **złap**:

Bez **rzuć** wewnętrzne **wywołaj** przekazuje 5, blok + przekazuje 8, więc blok **złap** przekazuje 8, a blok × przekazuje 80. Gdy jest **rzuć** to wewnętrzne **wywołaj** w ogóle nic nie przekazuje, podobnie jak blok + . Wartość wejściowa bloku **rzuć** 20 staje się wartością przekazywaną przez blok **złap**, a blok × mnoży 10 i 20.

[11] Na teraz musi istnieć inne polecenie, które nie robi nic, aby zakończyć definicję CATCH, z powodu wspomnianego wcześniej błędu.

### Tworzenie systemu wątków

Snap*!* może uruchamiać kilka skryptów naraz, dla jednego duszka i dla wielu. Jeśli masz tylko jeden komputer, to w jaki sposób może on robić wiele rzeczy naraz? Odpowiedź jest taka, że w każdej chwili działa tylko jeden proces, ale Snap*!* często kieruje jego uwagę z jednego skryptu na inny. U dołu każdego bloku pętli (**powtórz**, **powtarzaj…aż**, **zawsze**) znajduje się domyślne polecenie „ustąp", które zapamiętuje, gdzie jest obecny skrypt, i przełącza się na każdy inny skrypt po kolei. Na końcu każdego skryptu znajduje się niejawna komenda „koniec wątku" (wątek jest terminem technicznym dla procesu uruchamiania skryptu), która przełącza na inny skrypt bez pamiętania starego.

Ponieważ wszystko to dzieje się automatycznie, generalnie nie ma potrzeby, aby użytkownik myślał o wątkach. Ale, aby pokazać, że to też nie jest magia, przedstawimy implementację prostego systemu wątków. Używa ona globalnej zmiennej o nazwie **zadania**, która początkowo zawiera pustą listę. Każde użycie bloku **wątek** w kształcie litery C dodaje do listy kontynuację (skrypt w kształcie obwiedni). Blok **ustąp** używa polecenia uruchom z kontynuacją, aby utworzyć kontynuację częściowo wykonanego wątku, dodaje go do listy zadań, a następnie uruchamia pierwsze oczekujące zadanie. Blok **koniec wątku** (który jest automatycznie dodawany na końcu skryptu każdego wątku przez blok **wątek**) uruchamia następne oczekujące zadanie.

```
wątek akcja λ
dodaj (uruchom akcja ▸
       koniec wątku) do zadania
```

```
koniec wątku
zmienne skryptu nast ▸
jeżeli nie puste? zadania to
    ustaw nast na element 1 z zadania
    usuń 1 z zadania
    uruchom nast ▸
```

```
ustąp
uruchom (zmienne skryptu nast ▸
         dodaj kontynuacja do zadania
         ustaw nast na element 1 z zadania
         usuń 1 z zadania
         uruchom nast ▸
         nazwy parametrów: kontynuacja ◂▸) z kontynuacją
```

Oto przykładowy skrypt wykorzystujący system wątków. Jeden wątek mówi liczby; drugi mówi litery. Wątek numeryczny ustępuje po każdej liczbie pierwszej, a wątek literowy po każdej samogłosce. Zatem sekwencja balonów z tekstem jest 1,2, a, 3, b, c, d, e, 4,5, f, g, h, i, 6,7, j, k, l, m, n, o, 8,9,10, 11, p, q, r, s, t, u, 12,13, v, w, x, y, z, 14,15,16,17,18, ... 30.

[*Trzeba jeszcze dodefiniować predykat* ***pierwsza?****; WtK*]

Gdybyśmy chcieli, aby ten system zachowywał się dokładnie tak samo jak wątki Snap*!*, to powinniśmy zdefiniować nowe wersje **powtórz** i tak dalej, które uruchamiają **ustąp** po każdym powtórzeniu.

```
kiedy kliknięto [zielona flaga]
ustaw zadania na lista ▸
wątek
    dla i = 1 do 30
        powiedz i przez 1 s
        jeżeli pierwsza? i to
            ustąp
wątek
    zmienne skryptu litera ▸
    dla i = 97 do 122
        ustaw litera na Unicode i jako litera
        powiedz litera przez 1 s
        jeżeli lista a e i o u y ◂▸ zawiera litera to
            ustąp
koniec wątku
```

# XI Elementy interfejsu użytkownika

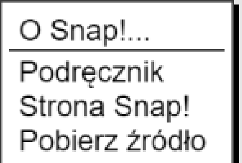
W tym rozdziale szczegółowo opisujemy różne przyciski, menu i inne klikalne elementy interfejsu użytkownika Snap*!*. Tutaj znowu jest mapa okna Snap*!*:

## A Elementy paska narzędzi

Przytrzymanie klawisza Shift podczas klikania dowolnego z przycisków menu daje dostęp do rozszerzonego menu z opcjami pokazanymi na czerwono, które są eksperymentalne lub przeznaczone dla programistów. Nie wymieniamy tu dodatkowych opcji, ponieważ często się zmieniają i nie należy się do nich przywiązywać. Ale to nie są sekrety.

### Menu Logo Snap*!*

Logo Snap*!* w lewym końcu paska narzędzi można kliknąć. Pokazuje ono menu z opcjami na temat Snap*!*:




Opcja **O Snap!...** wyświetla informacje o programie Snap*!*, w tym numery wersji dla modułów źródłowych, współpracowników, licencję (AGPL: w skrócie – można robić wszystko, oprócz tworzenia własnych wersji).

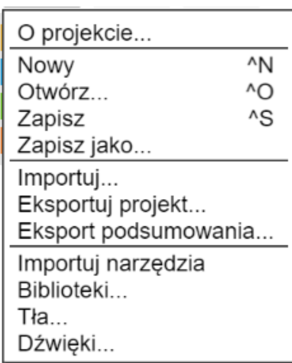
Opcja **Podręcznik** umożliwia pobranie kopii najnowszej wersji tego podręcznika w formacie PDF.

Opcja **Strona Snap!** otwiera okno przeglądarki z adresem `snap.berkeley.edu`, stroną internetową Snap*!*.

Opcja **Pobierz źródło** powoduje pobranie do zwykłego folderu pobierania przeglądarki, pliku zip zawierającego pliki źródłowe JavaScript dla Snap*!*. Możesz przeczytać kod, aby dowiedzieć się, jak jest zaimplementowany Snap*!*. Hostuj kopię na własnym komputerze (jest to jeden ze sposobów na kontynuowanie pracy np. w samolocie) lub utwórz zmodyfikowaną wersję z dostosowanymi funkcjami. (Dostęp do kont w chmurze jest jednak ograniczony do oficjalnej wersji umieszczonej na serwerze w Berkeley.) .

## Menu plik

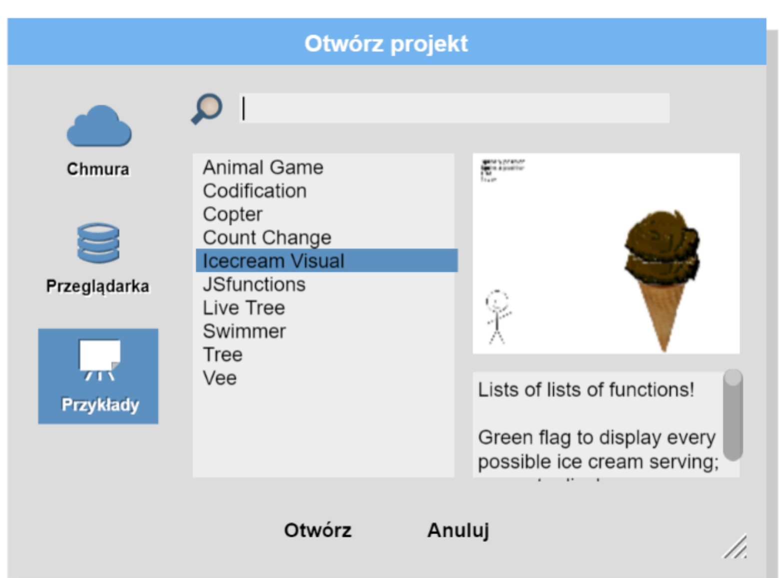
Ikona plik pokazuje menu poświęcone głównie ładowaniu i zapisowi projektów:




Opcja **O projekcie…** otwiera okno, w którym możesz wpisać uwagi na temat projektu: jak z niego korzystać, co robi, ewentualnie, który projekt został zmodyfikowany, aby go utworzyć, jakie inne źródła pomysłów zostały wykorzystane lub jakąkolwiek inną informację o projekcie. Ten tekst jest zapisywany wraz z projektem i jest przydatny, jeśli udostępniasz projekt innym użytkownikom.

Opcja **Nowy** uruchamia nowy, pusty projekt. Projekt, wcześniej wykorzystywany zniknie, więc następuje prośba o potwierdzenie, że naprawdę tego chcesz. (Projekt znika tylko z działającego aktualnie okna Snap*!*, należy więc przed użyciem **Nowy** zapisać bieżący projekt, jeśli chcesz go zachować.).

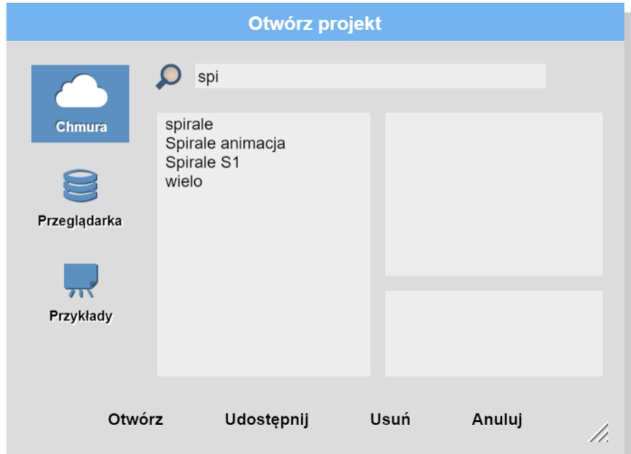
Zwróć uwagę na **^N** na końcu linii. Oznacza to, że możesz wpisać ctrl-N jako skrót dla tego elementu menu. Niestety tak nie jest w przypadku każdej przeglądarki. Niektóre przeglądarki Macintosha wymagają zamiast tego cmd N (⌘N), podczas gdy inne otwierają nowe okno przeglądarki zamiast nowego projektu. Musisz poeksperymentować. Ogólnie skróty klawiaturowe w Snap*!* są standardowe, takie jak w innych programach.




Opcja **Otwórz...** pokazuje okno otwierania projektu, w którym możesz wybrać projekt do otwarcia:

W tym oknie dialogowym trzy duże przyciski po lewej stronie umożliwiają wybór źródła projektów: **Chmura** oznacza twoje konto użytkownika Snap*!*; **Przeglądarka** oznacza pamięć twojej przeglądarki (dane dostępne tylko w tej przeglądarce, na tym komputerze,

dla tego użytkownika); **Przykłady** oznaczają zbiór przykładowych projektów, które dostarczamy. Pole tekstowe po prawej stronie tych przycisków zawiera alfabetyczną listę projektów z tego źródła; wybór projektu przez kliknięcie powoduje pokazanie po prawej stronie jego miniatury (obrazka sceny, na której został zapisany) i uwag do projektu.




Pasek wyszukiwania u góry można wykorzystać do znalezienia projektu według nazwy lub tekstu w uwagach do projektu. Oto przykład:

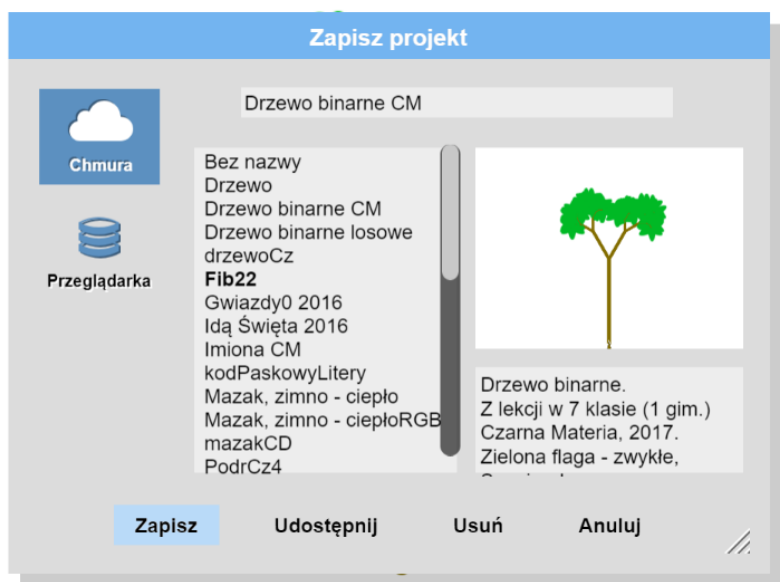
Szukałem mojego projektu z rysowaniem spiralek więc wpisałem „*spi*” w pasku wyszukiwania i zacząłem się zastanawiać dlaczego wyszukany został projekt „wielo”. Kiedy go kliknąłem zobaczyłem to:

*Moje wyszukiwanie znalazło słowo „Spiralowanie” w opisie projektu*. [*Wyszukiwanie WtK*].

Cztery przyciski na dole wybierają akcję do wykonania na wybranym projekcie. Pierwszy przycisk, **Otwórz**, ładuje projekt do Snap*!* i zamyka okno dialogowe. Drugi przycisk (tylko jeśli źródłem jest Chmura) jest widoczny jako **Udostępnij** lub **Wyłącz udostępnianie**; udostępniony projekt może zostać odczytany (ale nie zmodyfikowany) przez dowolnego użytkownika Snap!, nie tylko przez ciebie. Udostępnione projekty zaznaczono **pogrubioną** czcionką w polu listy projektów. Następny przycisk **Usuń** (jeśli źródłem jest Chmura lub Przeglądarka) po kliknięciu usuwa wybrany projekt. Na koniec przycisk **Anuluj** zamyka okno dialogowe bez otwierania projektu. (Nie cofa żadnego udostępnienia ani usunięcia.)

Opcja menu **Zapisz** zapisuje projekt w tym samym miejscu i pod tą samą nazwą, która została użyta podczas otwierania projektu. (Jeśli otwarty został udostępniony projekt innego użytkownika lub projekt przykładowy, zostanie on zapisany w pamięci przeglądarki lub na twoim koncie w chmurze, jeśli jesteś zalogowany).

Opcja menu **Zapisz jako...** otwiera okno dialogowe, w którym można określić, gdzie zapisać projekt:




To okno jest podobne do okna dialogowego **Otwórz**, z wyjątkiem poziomego pola tekstowego u góry, w którym wpisujesz nazwę projektu. Możesz także z tego miejsca udostępniać, anulować udostępnianie i usuwać projekty. Jeśli jesteś zalogowany, okno dialogowe startuje mając wybraną Chmurę; jeśli nie, wybrana zostanie **Przeglądarka**.

Opcja menu **Importuj**… służy do przeniesienia jakiegoś zewnętrznego zasobu do bieżącego projektu, zamiast ładowania całkowicie oddzielnego projektu. Możesz importować kostiumy (dowolny format obrazu obsługiwany przez twoją przeglądarkę), dźwięki (znowu, każdy format obsługiwany przez twoją przeglądarkę) i biblioteki bloków lub duszki (w formacie XML, wcześniej wyeksportowanym ze Snap*!*). Zaimportowane kostiumy i dźwięki będą należeć do aktualnie wybranego duszka; importowane bloki są globalne (dla wszystkich duszków). Użycie opcji **Import** jest równoznaczne z przeciągnięciem pliku z pulpitu do okna Snap*!*.

W zależności od przeglądarki opcja **Eksportuj projekt...** powoduje zapis bezpośrednio na dysku lub otwiera nową kartę przeglądarki zawierającą cały projekt w formacie **XML** (format zwykłego tekstu). Następnie możesz użyć funkcji Zapisz w przeglądarce, aby zapisać projekt jako plik **XML**, który powinien mieć nazwę cośtam.xml, aby Snap*!* rozpoznał go jako projekt, gdy później przeciągniesz go do okna Snap*!*. Jest to alternatywa dla zapisania projektu na koncie w chmurze, sposób na przechowywanie go na własnym komputerze.

Opcja **Eksportuj bloki...** służy do tworzenia biblioteki bloków. Przedstawia listę wszystkich (*niestandardowych; WtK*) globalnych bloków (dla wszystkich duszków) w projekcie i pozwala wybrać, które elementy eksportować. Następnie otwiera kartę przeglądarki z tymi blokami w formacie XML lub zapisuje plik bezpośrednio na dysku lokalnym, tak jak w opcji **Eksportuj projekt**. Biblioteki bloków można importować za pomocą opcji **Importuj** lub przeciągając plik do okna Snap*!*. Ta opcja jest wyświetlana tylko wtedy, gdy masz zdefiniowane niestandardowe bloki.

Opcja **Niewykorzystane bloki...** wyświetla listę wszystkich (*niestandardowych; WtK*) globalnych bloków w projekcie, które nie są nigdzie używane i oferuje ich usunięcie. Podobnie jak w przypadku **Eksportuj bloki** możesz wybrać za pomocą pól wyboru podzestaw do usunięcia. Ta opcja jest wyświetlana tylko wtedy, gdy masz zdefiniowane niestandardowe bloki.

Opcja **Eksportuj projekt...** tworzy stronę internetową, w formacie HTML, z wszystkimi informacjami o projekcie: jego nazwą, uwagami do projektu, obrazem tego, co jest na scenie, definicjami bloków globalnych, a następnie informacjami o każdym duszku: nazwa, garderoba (lista kostiumów) oraz zmienne lokalne i definicje bloków. Stronę można przekonwertować na format PDF przez przeglądarkę; ma to na celu spełnienie wymagań tworzenia zadań Advanced Placement Computer Science Principles (*zapis ma format XML, przeglądarki wyświetlają go jako kod, który można skopiować do dowolnego edytora; WtK*).

Opcja **Importuj narzędzia** importuje bibliotekę użytecznych bloków narzędzi, które możesz uważać za pierwotne bloki Snap*!*, choć faktycznie są napisane w Snap*!*. W niniejszym podręczniku założono, że biblioteka narzędzi została zaimportowana, szczególnie w objaśnieniach dotyczących list.

Opcja **Biblioteki...** przedstawia menu przydatnych, opcjonalnych bibliotek bloków:

Po kliknięciu na jednolinijkowy opis biblioteki zostaną wyświetlone bloki znajdujące się w bibliotece i dłuższe wyjaśnienie jej przeznaczenia. Możesz przeglądać biblioteki, aby znaleźć takie, które zaspokoją Twoje potrzeby.

Biblioteki i ich zawartość mogą się zmieniać, ale na ten moment **biblioteka iteracji** ma następujące bloki:

Bloki **kaskada** otrzymują wartość początkową i wielokrotnie wywołują funkcję dla tej wartości, *f (f (f (f ... (x))))*.

Blok **złożenie** bierze dwie funkcje i daje w wyniku funkcję *f(g(x))*.

Pierwsze trzy bloki **powtarzaj** są wariantami pierwotnego bloku **powtarzaj aż**, dając wszystkie cztery kombinacje tego, czy pierwszy test ma miejsce przed, czy po pierwszym powtórzeniu i czy warunek musi być prawdziwy, czy fałszywy, aby kontynuować powtarzanie. Ostatni blok **powtórz** jest podobny do pierwotnego bloku **powtórz**, ale sprawia, że liczba dotychczasowych powtórzeń (*nr powtórzenia*; *WtK*) jest dostępna dla powtarzanego skryptu. Kolejne dwa bloki są odmianami **dla**: pierwszy pozwala na określenie kroku, zamiast użycia ± 1, a drugi pozwala na dowolne wartości, a nie tylko liczby; wewnątrz skryptu, jest polecenie

uruchom pętla z parametrami następna-żądana-wartość a

w którym zastępuje się szary blok na obrazku wyrażeniem, aby nadać kolejną żądaną wartość indeksowi pętli.

Biblioteka **metody, listy** ma następujące bloki:

Blok **połącz** przyjmuje dowolną liczbę wejść listy i daje w wyniku listę wszystkich elementów wszystkich list wejściowych. **Odwróć** przekazuje listę z pozycjami listy wejściowej w odwrotnej kolejności. **Usuń powtórzenia z** przekazuje listę, w której nie ma dwóch jednakowych pozycji. Blok **sortuj** pobiera listę i predykat porównania o dwóch wejściach, taki jak < i daje w wyniku listę z elementami posortowanymi według tego porównania. Blok **asocjacja** służy do wyszukiwania klucza na *liście asocjacji*: liście list dwu-elementowych. Na każdej dwuelementowej liście pierwszy jest klucz, a druga wartość. Dane wejściowe to klucz i lista asocjacji; blok przekazuje pierwszą parę klucz-wartość, której klucz jest równy pierwszemu wejściu. **Puste?** jest wykorzystywane w pozostałych trzech blokach.

Pozostałe trzy bloki są wersjami narzędzi list, które udostępniają zmienną # zawierającą pozycję na liście wejściowej aktualnie rozpatrywanego elementu. Ta wersja **mapuj** umożliwia również wprowadzanie wielu list, w którym to przypadku funkcja **mapuj** musi przyjmować tyle wejść, ile jest list; zostanie wywołana przy wszystkich pierwszych elementach, wszystkich drugich elementach i tak dalej.

Biblioteka **strumieni** (leniwych list) zawiera bloki:

Strumienie to specjalny rodzaj list, których elementy nie są obliczane, dopóki nie są potrzebne. To sprawia, że niektóre obliczenia są bardziej wydajne, a także umożliwia tworzenie list z nieskończenie wieloma pozycjami, takimi jak lista wszystkich dodatnich liczb całkowitych. Pierwsze pięć bloków to strumieniowe wersje bloków list **wstaw…przed**, **element *1* z**, **bez pierwszego z**, **mapuj** i **zachowaj**. Kolejne dwa bloki, **mapuj** i **puste?**, są te same co w bibliotece narzędzi, uwzględnione zostały tutaj, ponieważ są używane w implementacji funkcji strumieniowych. **Pokaż strumień** pobiera strumień i liczbę *n* jako dane wejściowe i przekazuje zwykłą listę pierwszych *n* elementów strumienia. **Sito** to przykładowy blok, który przyjmuje jako strumień wejściowy liczby całkowite zaczynające się od 2 i daje w wyniku strumień wszystkich liczb pierwszych. **Strumień** jest odpowiednikiem pierwotnego bloku **lista**; tj. tworzy skończony strumień z podanych elementów.

Biblioteka funkcji **zachłannych** (wieloargumentowych) ma następujące bloki: Są to wersje operatorów asocjacyjnych +, ×, **i**, oraz **lub**, które pobierają dowolną liczbę wejść zamiast dokładnie dwóch wejść. W przypadku każdego wejścia zachłannego, możesz również upuścić listę wartości na strzałki, zamiast podawać dane wejściowe pojedynczo.

Biblioteka usług **WWW** zawiera następujące bloki:

Pierwszy blok to uogólnienie pierwotnego bloku **url**, pozwalające na większą kontrolę nad różnymi opcjami w żądaniach internetowych: GET, POST, PUT i DELETE oraz dokładną kontrolę nad treścią wiadomości wysłanej na serwer. **Moja lokalizacja** podaje Twoją szerokość i długość geograficzną. Blok **klucz:..wartość:** jest po prostu konstruktorem dla abstrakcyjnego typu danych używanego z innymi blokami.

Biblioteka **słów i zdań** ma następujące bloki:

wszystkie bez pierw. litery z
wszystkie bez pierw. słowa z
wszystkie bez ost. litery z
wszystkie bez ost. słowa z
pierw. słowo z
ost. litera z
ost. słowo z
puste słowo?
puste zdanie?

Bloki te sprawiają, że wygodniej jest myśleć o łańcuchu tekstowym albo jako słowie złożonym z liter, albo jako zdaniu złożonym ze słów, a nie z pojedynczych znaków. Każde słowo zdania to ciąg znaków bez spacji. Oznacza to, że bloki powiązane ze zdaniami dzielą łańcuch na znakach spacji. Dowolna liczba kolejnych spacji liczy się jako pojedynczy separator. Nazwy powinny być zrozumiałe.

Biblioteka **wieloargumentowych poleceń warunkowych** ma następujące bloki:

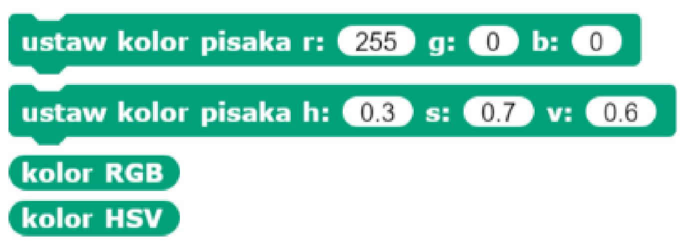
Bloki **złap**, **rzuć** i **dla każdego** to duplikaty z biblioteki narzędzi i zostały tu uwzględniane, ponieważ są używane do implementacji innych. Blok **wybierz** ustawia warunkowo wiele gałęzi, podobnie jak ***cond*** w Lispie lub ***switch*** w językach z rodziny C. Pierwsza gałąź jest wbudowana w blok **wybierz**; składa się z testu logicznego w pierwszym sześciokątnym polu i skryptu akcji w polu C-kształtnym, który zostanie uruchomiony, jeśli test ma wartość ***prawda***. Pozostałe gałęzie przechodzą do sześciokątnego wejścia wieloargumentowego (*ze strzałkami*) na końcu; każda gałąź składa się z bloku **a jeżeli**, który zawiera test logiczny i odpowiadający mu skrypt akcji, z wyjątkiem ewentualnie ostatniej gałęzi, w której może znaleźć się bezwarunkowy blok **inaczej**. Podobnie jak w innych językach, gdy gałąź odniosła sukces, żadne inne gałęzie nie są testowane.

Biblioteka **LEAP Motion** zawiera następujące bloki:

Leap number of hands
Leap x position of hand 1
Leap yaw rotation of hand 1
Leap x position of finger 1
Leap hand 1 is pinching
Leap pinch strength of hand 1
Leap

Kontroler LEAP Motion to urządzenie wykrywające położenie rąk użytkownika, modelujące kąty poszczególnych kłykci. To jedno z wielu urządzeń zewnętrznych, które Snap*!* może wspierać. Zasadniczo, aby użyć urządzenia z Snap*!*, musisz pobrać i zainstalować osobny program kontrolera. Dotyczy to również LEAP, ale ten kontroler jest instalowany w ramach konfigurowania urządzenia, więc potrzebna jest tylko biblioteka bloków. (*Tej biblioteki nie tłumaczę ponieważ nie mam dostępu do urządzenia; WtK*).

Biblioteka **kolorów pisaka** ma następujące bloki:



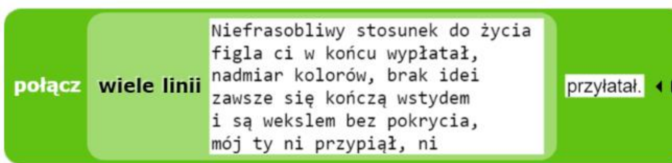

Pierwsze dwa to polecenia do ustawienia koloru pisaka; ostatnie dwa przekazują kolor pisaka. Dwie obsługiwane reprezentacje przestrzeni kolorów to RGB (czerwony-zielony-niebieski), w którym każda intensywność koloru jest liczbą od 0 do 255, oraz HSV (odcień-nasycenie-wartość), w której każdy wymiar jest ułamkiem między 0 i 1. Funkcje dają w wyniku listy trzech elementów.

Biblioteka **przechwytywania błędów** zawiera następujące bloki:

Blok **bezpiecznie próbuj** pozwala na obsługę błędów, które zdarzają się, gdy skrypt jest uruchamiany w programie, zamiast zatrzymywania skryptu z czerwonym halo i niejasnym komunikatem o błędzie. Blok uruchamia skrypt znajdujący się w pierwszym polu C-kształtnym. Jeśli kończy się bez błędu, nic się nie dzieje. Ale jeśli wystąpi błąd, uruchamia się kod znajdujący się w drugim polu C-kształtnym. Podczas gdy ten drugi skrypt jest uruchomiony, zmienna błąd zawiera tekst komunikatu o błędzie, który zostałby wyświetlony, gdyby nie było przechwytywania błędu. Blok **błąd** jest w pewnym sensie odwrotny: pozwala programowi wygenerować komunikat o błędzie, który będzie wyświetlany z czerwonym halo, chyba że zostanie przechwycony przez **bezpiecznie próbuj**. Blok **niech** służy do implementacji biblioteki; odpowiada blokowi **zmienne skryptu**, a po nim **ustaw**.

Biblioteka **wiele linii** ma tylko jeden blok:

wiele linii Jest używany do zezwalania na wprowadzanie tekstu mającego więcej niż jedną linię. Dzięki tej bibliotece możesz powiedzieć



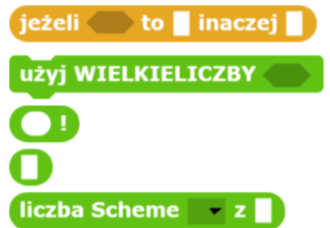

Niefrasobliwy stosunek do życia
figla ci w końcu wypłatał,
nadmiar kolorów, brak idei
zawsze się kończą wstydem
i są wekslem bez pokrycia,
mój ty ni przypiął, ni przyłatał.

Biblioteka **pobierania / ustawiania wartości systemowych** ma następujące bloki:

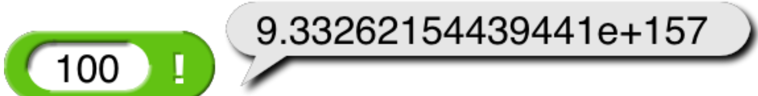
Celem tej biblioteki jest umożliwienie programowi dostępu do ustawień kontrolowanych przez elementy interfejsu użytkownika, takie jak menu ustawień. Blok **ustawienia** przekazuje ustawienie; blok **ustaw flagę na** ustawia opcje tak lub nie, które mają pola wyboru (checkboxes) w interfejsie użytkownika, podczas gdy blok **ustaw wartość na** steruje ustawieniami wartości liczbowych lub tekstowych, takich jak nazwa projektu.

Niektóre ustawienia są zazwyczaj pamiętane dla danego użytkownika, na przykład wartość „powiększenie bloków". Ale kiedy te ustawienia zostaną zmienione przez tę bibliotekę, zmiana będzie obowiązywać tylko wtedy, gdy załadowany zostanie projekt korzystający z biblioteki. Nie będą wprowadzone żadne stałe zmiany.

Biblioteka **nieskończonej precyzji liczb całkowitych** zawiera następujące bloki:

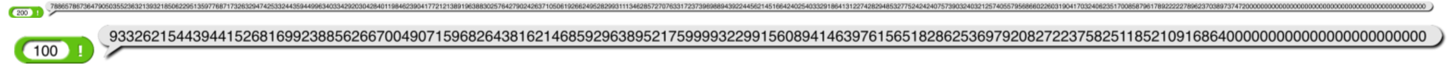



Blok **użyj WIELKIELICZBY** pobiera daną wejściową typu logicznego, aby włączyć lub wyłączyć funkcję nieskończonej precyzji. Po włączeniu wszystkie operatory arytmetyczne są na nowo definiowane, aby akceptować i dawać w wyniku liczby całkowite o dowolnej liczbie cyfr (ograniczone tylko pamięcią komputera) i w rzeczywistości, całą wieżę numeryczną języka Scheme, z dokładnymi liczbami wymiernymi i liczbami zespolonymi. Blok **liczba Scheme** zawiera listę funkcji mających zastosowanie do liczb Scheme, w tym podtyp predykatów, takich jak **wymierny?** i **nieskończony?** oraz selektory, takie jak **licznik** i **część rzeczywista**. Blok **!** oblicza funkcję silni, i można go użyć do sprawdzenia, czy pakiet WIELKIELICZBY jest włączony. Bez WIELKIELICZBY:




200 ! Infinity

Z WIELKIELICZBY:




375-cyfrowa wartość 200! nie jest możliwe do odcyfrowania na tej stronie, ale jeśli klikniesz prawym przyciskiem myszy na bloku i wybierzesz „obrazek skryptu z wynikiem", możesz otworzyć wynikowy obraz w oknie przeglądarki i przewijać go. (Liczba kończy się grupą cyfr zero, to nie błąd zaokrągleń, czynniki pierwsze 100 i 200 zawierają wiele kopii 2 i 5.). Funkcja **jeżeli-to-inaczej** jest używana wewnątrz **!**, a blok bez nazwy to sposób na wprowadzenie takich rzeczy jak ***3/4*** i ***4 + 7i*** w liczbowych polach wejściowych przez przekształcenie pola na dowolnego typu.

Biblioteka **animacji** na następujące bloki:

Pomimo nazwy nie chodzi tylko o grafikę; możesz animować wartości zmiennej lub cokolwiek innego co jest wyrażone liczbowo.

leć 50 kroków przez 1 s z wygładzaniem linear
animuj x position o 50 przez 1 s z wygładzaniem linear
jeżeli to inaczej
animuj ustal weź delta 50 na 1 s z wygładzaniem
błąd
nr
do potęgi
PI
stopnie 0.5
wygładzanie linear
--
znajdź pierw. w takie, że

Główną ideą tej biblioteki jest funkcja wygładzania, funkcja, której dziedzina i wartości są liczbami rzeczywistymi z zakresu od 0 do 1 włącznie. Funkcja reprezentuje, która część „odległości" (w cudzysłowie, ponieważ może to być dowolna wartość liczbowa, taka jak temperatura w symulacji pogody) stąd dotąd ma być ujęta w jakim ułamku czasu. Funkcja liniowego (linear) wygładzania oznacza stały wzrost. Kwadratowa funkcja wygładzania (quadratic) oznacza rozpoczynanie powoli i przyspieszanie. (Należy zauważyć, że ponieważ wymagane jest, aby f (0) = 0 i f (1) = 1, to istnieje tylko jedna funkcja liniowego wygładzania, f (x) = x i podobnie dla innych kategorii). Blok wygładzanie linear podaje niektóre z popularnych funkcji wygładzania.

Dwa bloki Ruchu w tej bibliotece animują duszka. **Leć** zawsze animuje ruch duszka. Pierwsze wejście menu rozwijanego bloku **animuj** pozwala animować ruch poziomy lub pionowy, ale także animuje kierunek lub rozmiar duszka. Blok **animuj** w palecie Kontroli pozwala animować dowolną wartość liczbową za pomocą dowolnej funkcji wygładzania. Wejścia ***ustal*** i ***weź*** najlepiej wyjaśnić na przykładzie:

animuj ustal ustaw x na 0 weź pozycja X delta 50 na 1 s z wygładzaniem wygładzanie linear

jest równoważne

animuj x position o 50 przez 1 s z wygładzaniem linear

Pozostałe osiem bloków w bibliotece pełni rolę pomocniczą dla tych czterech.

W bibliotece **pikseli** są następujące bloki:

uaktualnij ➤ na ▤
aktualny kostium
kopia z ➤
pokaż obraz ▤
piksele w ➤
niech zmienna będzie ▢

Kostiumy to w Snap*!* dane pierwszej klasy. Pola wejścia w kształcie żółwia ➤ przyjmują kostiumy jako dane wejściowe. Blok aktualny kostium przekazuje aktualny kostium duszka, a blok ja kostiumy ▾ przekazuje listę wszystkich kostiumów duszka. (Uwaga: domyślny kształt żółwia nie jest dla tej biblioteki kostiumem. Tak, to jest ironiczne, biorąc pod uwagę kształt wejścia.). Blok piksele w ➤ bierze kostium jako dane wejściowe i przekazuje listę pikseli, z których każdy jest listą czterech elementów – wartości RGBA (czerwony, zielony, niebieski, przezroczystość). Każdy element jest liczbą z przedziału 0-255. Jeśli modyfikujesz elementy listy pikseli, wykonaj:

uaktualnij aktualny kostium na zmodyfikowane piksele
zmień kostium na nr kostiumu

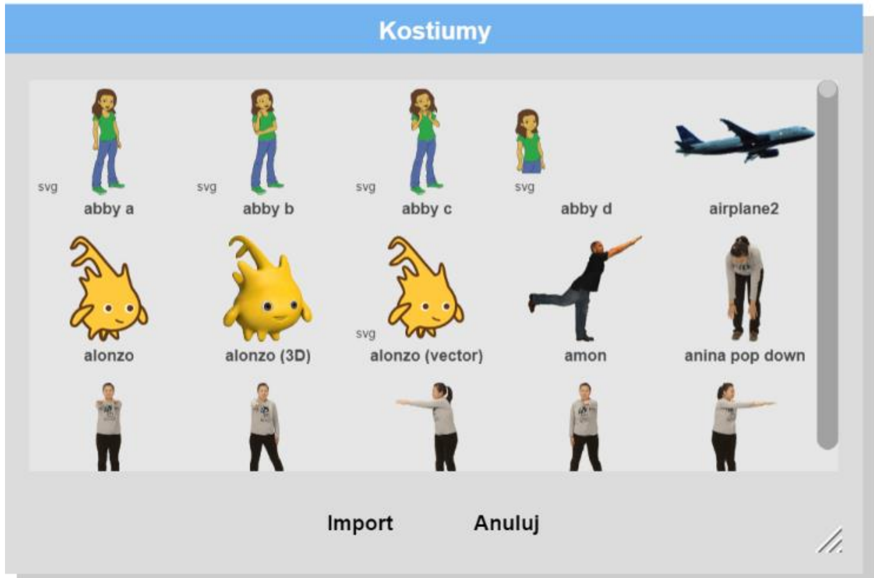
kostium w garderobie duszka pokazuje zmianę, a drugie polecenie zmienia wygląd rzeczywistego duszka. Alternatywnie można użyć bloku **pokaż obraz** z listą pikseli jako danych wejściowych, aby zmienić wygląd duszka bez zmiany jego zapamiętywanych kostiumów. Możesz też powiedzieć:

dodaj aktualny kostium do ja kostiumy ▾

aby dodać do garderoby nowy kostium nie zmieniając żadnego z istniejących kostiumów. Blok **kopia** pozwala stworzyć zmodyfikowaną wersję kostiumu, zachowując starą wersję nietkniętą. Blok **niech** jest pomocnikiem używanym do tworzenia tej biblioteki. Uwaga: Obecnie nie ma sposobu by znaleźć kształt kostiumu, więc twój kod zna piksele, ale nie to, który piksel wchodzi w skład kostiumu. Aktualizacja innego kostiumu ze zmodyfikowaną wersją macierzy pikseli kostiumu, niekoniecznie zrobi to, co masz na myśli.

---

Opcja **Kostiumy…** otwiera przeglądanie biblioteki kostiumów:




Możesz zaimportować pojedynczy kostium, klikając go, a następnie klikając przycisk Importuj. Możesz też zaimportować więcej niż jeden kostium, klikając dwukrotnie każdy, a po zakończeniu klikając przycisk Anuluj. Zauważ, że niektóre kostiumy są oznaczone na tym obrazku „svg"; są to kostiumy w formacie wektorowym, które nie są (jeszcze, wkrótce pewnie będą) edytowalne w Snap*!*.

Jeśli w zagrodzie duszków masz wybraną scenę, zamiast duszka, opcja **Kostiumy ...** zmienia się na **Tła ...** i daje możliwość wyboru tła w przeglądarce:

Biblioteki kostiumów i obrazów tła zawierają zarówno obrazy bitmapowe (nieostre po powiększeniu), jak i wektorowe (powiększanie płynne). Dzięki Scratch 2.0 za większości z tych obrazków! Niektóre przeglądarki nie importują obrazu wektorowego, ale przekształcają go w bitmapę.

Opcja **Dźwięki ...** otwiera trzeci rodzaj przeglądarki multimediów:

Przyciski Graj można wykorzystać do odsłuchania dźwięków.

## Menu Chmura

Ikona chmury otwiera menu opcji związanych z twoim kontem Snap!. Bez logowania zobaczysz takie menu:

Logowanie...
Rejestracja...
Zresetuj hasło...

Wybierz **Logowanie ...** jeśli masz konto Snap*!* i pamiętasz swoje hasło. Wybierz opcję **Rejestracja ...** jeśli nie masz konta. Wybierz **Zresetuj hasło ...** jeśli nie pamiętasz hasła lub po prostu chcesz je zmienić. Otrzymasz e-mail z nowym tymczasowym hasłem na adres podany podczas tworzenia konta. Użyj tego hasła, aby się zalogować, a następnie wybierz własne hasło, jak pokazano poniżej.

Jeśli jesteś już zalogowany, zobaczysz to menu (*trzeba poprawić; WtK*):

Logout witek
Change Password...

**Wyloguj** jest oczywiste, ale ma dodatkową możliwość – pokazuje, kto jest zalogowany. **Zresetuj hasło...** poprosi o stare hasło (tymczasowe po zresetowaniu hasła) i o nowe hasło, które wpisujesz dwukrotnie, ponieważ nie jest pokazywane.

### Menu ustawień

Ikona ustawień ⚙ pokazuje menu ustawień Snap*!* dla bieżącego projektu lub stałych, w zależności od opcji:

Język...
Powiększ bloki...
Rozmiar sceny...
☑ Wsparcie wyświetlacza Retina
☐ Suwaki wejściowe
☐ Tryb turbo
☐ Debugowanie krokowe
☑ Długa forma dialogu wejścia
☐ Prosta etykieta prototypu
☐ Dźwięk kliknięcia
☑ Prosty wygląd
☐ Omijaj bezpieczne skrypty
☐ Płaskie końce linii
☐ Kodowanie bloków
☑ Podtrzymywanie dziedziczenia

Opcja **Język..**. pozwala zobaczyć interfejs użytkownika Snap*!* (bloki i wiadomości) w języku innym niż angielski. (Uwaga: Tłumaczenia zostały dostarczone przez użytkowników Snap*!*, jeśli brakuje Twojego języka ojczystego, wyślij nam e-mail!).

Opcja **Powiększ bloki...** umożliwia zmianę rozmiaru bloków zarówno w paletach, jak i w skryptach. Standardowy rozmiar to 1.0 jednostki. Głównym celem tej opcji jest umożliwienie robienia obrazków o wysokiej rozdzielczości do wykorzystania na plakatach. Można to również wykorzystać do poprawy czytelności projekcji na ekranie podczas wykładu, ale pamiętaj, że nie powoduje to, że paleta lub obszary skryptów są szersze, więc funkcja powiększania w przeglądarce (ctrl/ cmd +) może być bardziej praktyczna. Uwaga: zoom 2 jest gigantyczny! Nawet nie próbuj 10.

Opcja **Rozmiar sceny...** umożliwia ustawienie rozmiaru pełnowymiarowej sceny w pikselach. Jeśli scena jest rozmiaru połowy wielkości lub podwójnego (tryb prezentacji), wartości rozmiaru sceny nie zmieniają się; zawsze odzwierciedlają pełnowymiarową scenę.

Pozostałe opcje pozwalają włączać i wyłączać różne funkcje. **Wsparcie wyświetlacza Retina** jest dostępna tylko wtedy, gdy komputer jest podłączony do ekranu Retina (lub „4x HD"), w którym to przypadku jest on włączony domyślnie. Obsługa Retina sprawia, że tekst i bloki są znacznie ostrzejsze niż w przypadku ekranu Retina bez włączonej obsługi Retina, ale każdy obrazek ma cztery razy więcej pikseli, co znacznie spowalnia wyświetlanie. (Ponadto, jeśli wykonasz obrazki skryptu lub sceny, będą one podczas oglądania na ekranach innych niż Retina dwa razy wyższe i dwa razy szersze, niż się spodziewasz).

**Suwaki wejściowe** dają alternatywny sposób umieszczania wartości w numerycznych polach wejściowych; jeśli klikniesz w takie pole, pojawi się suwak, który możesz ustawiać za pomocą myszy:

Zakres suwaka będzie wynosił od 25 mniej do 25 więcej niż aktualna wartość wejścia. Jeśli chcesz wprowadzić większą zmianę, możesz przesunąć suwak do końca, a następnie ponownie kliknąć pole wejściowe, aby uzyskać nowy suwak z nowym punktem środkowym. Ale nie warto użyć tej techniki do zmiany wartości wejściowej z 10 na 1000 i nie działa ona wcale dla niecałkowitych zakresów wejściowych. Ta funkcja została zaimplementowana, ponieważ wprowadzanie danych z wirtualnej klawiatury w telefonach i tabletach na początku nie działało i nadal nie działa na urządzeniach z systemem Android, a suwaki zapewniają obejście tego problemu. Od tego czasu znalazło to inne zastosowanie w zapewnieniu „żywej" reakcji na wprowadzane zmiany; jeśli **suwaki wejściowe** są zaznaczone, ponowne otwarcie menu ustawień pokaże dodatkową opcję o nazwie **Execute on slider change** (Wykonaj przy zmianie suwaka). Jeśli ta opcja jest również zaznaczona, zmiana suwaka w obszarze skryptowym automatycznie uruchamia skrypt, w którym nastąpiła zmiana. Projekt **Live tree** w kolekcji **Przykłady** (*Otwórz / Przykłady; WtK*) pokazuje, jak można z niego korzystać; zawiera własny blok drzewa fraktalnego z kilkoma polami wejściowymi i możesz zobaczyć przesuwając suwak, jak wartość w każdym z nich wpływa na obraz drzewa. Ta opcja dotyczy projektu, a nie użytkownika.

**Tryb turbo** sprawia, że większość projektów działa znacznie szybciej, kosztem nie aktualizowania wyświetlania sceny. (Snap*!* spędza zazwyczaj większość czasu, rysując duszki i aktualizując plakietki zmiennych, zamiast faktycznie wykonywać instrukcje w skryptach). Tryb turbo nie jest dobrym pomysłem dla projektu z blokami **leć** lub takimi, w których użytkownik wchodzi w interakcje z animowanymi postaciami, ale świetnie nadaje się do rysowania skomplikowanego fraktala lub obliczania pierwszego miliona cyfr π, gdy nie musisz widzieć niczego, aż do końcowego wyniku. W trybie turbo przycisk, który zwykle pokazuje zieloną flagę, staje się zieloną błyskawicą. (Ale blok kapeluszowy **kiedy kliknięto** nadal aktywuje się po kliknięciu tego przycisku).

**Debugowanie krokowe** umożliwia zwolnioną realizację skryptu opisaną w rozdziale I. Zaznaczenie tej opcji jest równoważne kliknięciu przycisku śladu stóp nad obszarem skryptu. Nie jest to potrzebne, z wyjątkiem sytuacji, gdy aktywnie debugujesz (*poprawiasz, szukasz błędów; WtK*), ponieważ nawet ustawienie suwaka na największą szybkość nadal daje znaczne spowolnienie.

Jeśli jest zaznaczona **Długa forma dialogu wejścia**, to po utworzeniu lub edytowaniu własnego bloku podczas dodawania parametru (+) od razu widać długą wersję okna dialogowego nazwy parametru, która zawiera opcje typów, domyślne ustawienia wartości itd. zamiast krótkiej formy zawierającej tylko nazwę i wybór między nazwą wejścia i tekstem tytułowym. Domyślne (niezaznaczone) ustawienie jest zdecydowanie najlepsze dla początkujących, ale bardziej doświadczeni programiści Snap*!* mogą uważać, że wygodniej jest zawsze oglądać długą formę.

**Prosta etykieta prototypu** eliminuje znaki plus między słowami w bloku prototypu w Edytorze bloków. To sprawia, że trudniej jest dodać pola wejściowe do tworzonego, niestandardowego bloku; musisz najechać myszą w miejscu, w którym znajdowałby się znak plus, aż pojawi się jeden tymczasowy znak plusa, aby można było kliknąć. Jest to dedykowane dla osób robiących obrazki skryptów z edytora bloków, do dokumentacji, takiej jak ten podręcznik. Pewnie nie będziesz tego potrzebować w innym celu.

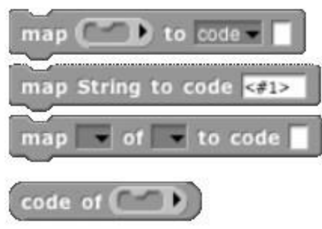
**Dźwięk kliknięcia** powoduje irytujący efekt dźwiękowy, gdy jeden blok sklei się z drugim w skrypcie. Niektóre małe dzieci i nasz kolega Dan Garcia to lubią, ale jeśli jesteś takim dzieckiem, pamiętaj, że doprowadzenie do szaleństwa rodziców lub nauczycieli spowoduje, że nie pozwolą ci używać Snap*!*. Jeśli jednak jesteś niewidomy, masz specjalne pozwolenie na jego włączenie.

**Prosty wygląd** zmienia „skórkę” okna Snap*!* na naprawdę paskudny wygląd z białym i jasnoszarym tłem, prostokątnymi zamiast zaokrąglonymi przyciskami i monochromatycznymi blokami (zamiast normalnych cieniowanych bloków, wyglądających trochę na 3D). Monochromatyczne bloki są powodem „flat” (*design*) w angielskiej nazwie tej opcji. Jedyne, co można powiedzieć o tej opcji, to że z powodu białego tła, może lepiej wtopić się w resztę strony internetowej, gdy projekt Snap*!* jest uruchamiany w ramce na większej stronie (*sorry Brian, mnie się ten wygląd bardziej podoba i systematycznie z niego korzystam; WtK*).

**Skrypty bezpieczne dla wątków** zmieniają sposób w jaki Snap! odpowiada, gdy zdarzenie (kliknięcie zielonej flagi, powiedz) uruchamia skrypt, a następnie, gdy skrypt nadal działa, to samo zdarzenie dzieje się ponownie. Zwykle uruchomiony proces zatrzymuje się tam, gdzie jest, ignorując pozostałe polecenia w skrypcie, a cały skrypt rozpoczyna się od początku. To zachowanie jest odziedziczone ze Scratcha, ponieważ zachowanie niektórych projektów konwertowanych ze Scratcha od tego zależy; właśnie dlatego jest to stan domyślny. Czasami jest to również dobre, zwłaszcza w przypadku projektów, w których muzyka jest odtwarzana w odpowiedzi na kliknięcie myszą lub naciśnięcie klawisza. Jeśli nuta jest nadal odtwarzana, ale wybierasz inną, to chcesz, aby nowa zaczęła się teraz a nie później, po zakończeniu starego procesu. Ale jeśli skrypt powoduje kilka zmian w bazie danych i jest przerwany w środku, wynikiem może być niespójność bazy danych. Gdy wybierzesz **Skrypty bezpieczne dla wątków**, to to samo zdarzenie, które dzieje się ponownie w trakcie działania skryptu, jest po prostu ignorowane. (To pewnie nadal nie jest to właściwe, wydarzenie powinno zostać zapamiętane, a skrypt powinien zostać uruchomiony, gdy tylko się ono skończy. Prawdopodobnie uda nam się kiedyś dodać ten sposób).

**Płaskie końce linii** wpływają na rysowanie grubych linii (duży rozmiar pisaka). Zazwyczaj końce są zaokrąglone, co najlepiej wygląda podczas obracania na rogach. Po wybraniu tej opcji końce są płaskie. Przydaje się to do rysowania ściany z cegły lub wypełniania prostokąta.

**Kodowanie bloków** udostępnia eksperymentalną funkcję, która może tłumaczyć projekt Snap*!* do tekstowego (a nie blokowego) języka programowania. Funkcja nie używa żadnego konkretnego języka; zamiast tego możesz przygotować tłumaczenie dla każdego pierwotnego bloku, używając tych specjalnych bloków (*widocznych w palecie **Dane** po włączeniu opcji; przyp. WtK*):




Korzystając z tych pierwotnych bloków, możesz zbudować bibliotekę bloków do tłumaczenia na dowolny język programowania. Zobacz, czy takie biblioteki zostały dodane do naszej kolekcji bibliotek (lub dodaj do niej swoją).
Aby zobaczyć kilka przykładów, otwórz projekt „**Codification**” na liście projektów **Przykłady**. Edytuj bloki **map to Smalltalk**, **map to JavaScript** itp., aby zobaczyć przykłady, jak przygotować tłumaczenie bloków.

**Podtrzymywanie dziedziczenia** umożliwia prototypowanie dziedziczenia obiektowego dla duszków. Ta opcja jest domyślnie włączona. Prawdopodobnie wkrótce zniknie jako opcja.

### Kontrola pracy krokowej

Za przyciskami menu widzisz nazwę projektu. Dalej widać przycisk pracy krokowej, który służy do włączania wizualizacji kolejnych kroków realizacji skryptu. Gdy jest on włączony pojawia się za nim suwak do kontrolowania prędkości pracy krokowej.

### Przyciski zmiany rozmiarów sceny

Dalej na pasku narzędzi, ale nad lewą krawędzią sceny, znajdują się dwa przyciski zmieniające rozmiar sceny. Pierwszym z nich jest przycisk **zmniejsz / powiększ**. Zwykle wygląda to następująco: Kliknięcie przycisku powoduje wyświetlenie sceny o połowie normalnego rozmiaru w pionie i poziomie

(tak, że zajmuje ona 1/4 jej zwykłego obszaru). Gdy scena ma połowę rozmiaru, przycisk wygląda następująco: , a kliknięcie go przywraca scenę do normalnego rozmiaru. Głównym powodem, dla którego możesz chcieć tak zmniejszyć scenę, jest proces tworzenia, budowania skryptów z rozbudowanymi polami wyrażeń wejściowych i normalny obszar skryptów nie jest wystarczająco szeroki, aby pokazać kompletny skrypt. Zwykle przełączysz się z powrotem na normalny rozmiar, aby wypróbować projekt. Następny przycisk **trybu prezentacji** zwykle wygląda następująco: . Kliknięcie przycisku powoduje podwójne powiększenie sceny w obu wymiarach i eliminuje większość innych elementów interfejsu użytkownika (paleta, obszar skryptów, zagroda duszków i większość paska narzędzi). Po otwarciu udostępnionego projektu za pomocą odsyłacza, który ktoś ci wysłał, projekt rozpoczyna się w trybie prezentacji. W trybie prezentacji przycisk wygląda następująco: . Kliknięcie go powoduje powrót do normalnego trybu (tworzenia projektu).

### Przyciski kontroli projektu

Nad prawą krawędzią sceny znajdują się trzy przyciski kontrolujące realizację projektu.

Z technicznego punktu widzenia **zielona flaga** nie bardziej kontroluje realizację projektu, niż cokolwiek innego, co może uruchomić blok kapeluszowy: naciśnięcie klawisza lub kliknięcie duszka. Ale zostało przyjęte, że kliknięcie flagi powinno rozpocząć działanie projektu od samego początku. To tylko konwencja; niektóre projekty nie mają w ogóle skryptów kontrolowanych przez flagę, zamiast tego reagują na sterowanie klawiaturą. Kliknięcie z wciśniętym klawiszem Shift włącza tryb Turbo, a przycisk wygląda wtedy jak błyskawica: , ponowne kliknięcie z wciśniętym klawiszem Shift wyłącza tryb Turbo.

Skrypty mogą symulować kliknięcie zielonej flagi przez nadanie specjalnego komunikatu **__shout__ go__** (dwa podkreślenia w każdym z trzech pokazanych miejsc).

Przycisk **pauza** zawiesza wykonywanie wszystkich skryptów. Kliknięcie przycisku podczas działania skryptów powoduje zmianę kształtu przycisku, który staje się przyciskiem **odtwarzania**: . Kliknięcie go w tej formie powoduje wznowienie zawieszonych skryptów. Jest również w palecie Kontrola blok **pauzuj wszystko**, który można wstawić do skryptu, aby zatrzymać wszystkie skrypty; zapewnia to istotę możliwości debugowania z punktem przerwania. Używanie przycisku pauzy jest nieco inne w trybie pracy krokowej opisanym w rozdziale I.

Przycisk **stop** zatrzymuje wszystkie skrypty, tak jak blok **zatrzymaj wszystko**. Nie zapobiega to ponownemu uruchamianiu skryptu w odpowiedzi na kliknięcie lub naciśnięcie klawisza; interfejs użytkownika jest zawsze aktywny. Przycisk usuwa także wszystkie tymczasowe klony.

## B Obszar palety bloków

W górnej części obszaru palety znajduje się osiem przycisków, które powodują wybór palety (kategorii bloków): Ruch, Wygląd, Dźwięk, Pisak, Kontrola, Czujnki, Wyrażenia i Dane (która obejmuje również bloki list i inne). Nie mają one kontekstowego menu.

### Przyciski w palecie

Pod ośmioma przyciskami wyboru palety znajdują się u góry aktualnej palety dwa półprzezroczyste przyciski. Pierwszy z nich to przycisk *wyszukiwania* *o*dpowiednik ctrl-F z klawiatury: zastępuje paletę paskiem wyszukiwania, w którym można wpisać część tekstu tytułowego bloku, który próbujesz znaleźć. Aby wyjść z trybu wyszukiwania, kliknij jeden z ośmiu selektorów palety lub naciśnij Escape.

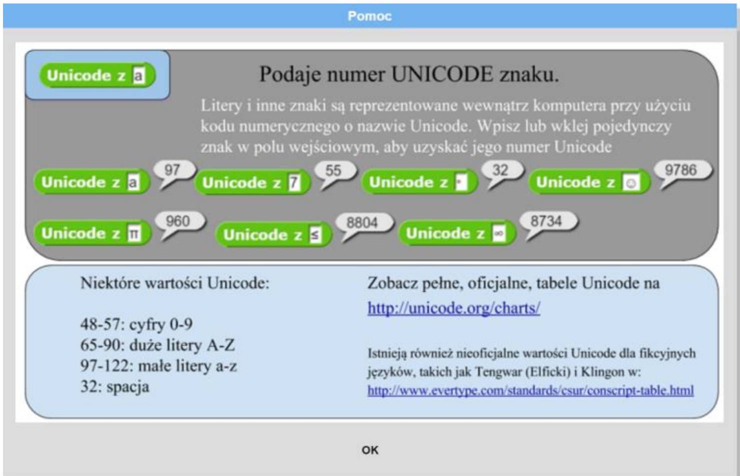
Drugi przycisk jest równoważny z przyciskiem "Nowy blok", z wyjątkiem tego, że okno dialogowe, które otwiera ma wybraną bieżącą paletę (odpowiedni kolor).

## Menu kontekstowe palety bloków

pomoc...
ukryj

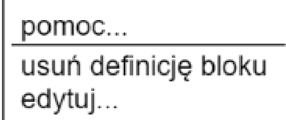
Jeśli klikniesz prawym przyciskiem myszy (lub z ctrl) pierwotny blok w palecie, zobaczysz menu:

Opcja **pomoc…** wyświetla okno z dokumentacją na temat bloku.  Oto przykład:

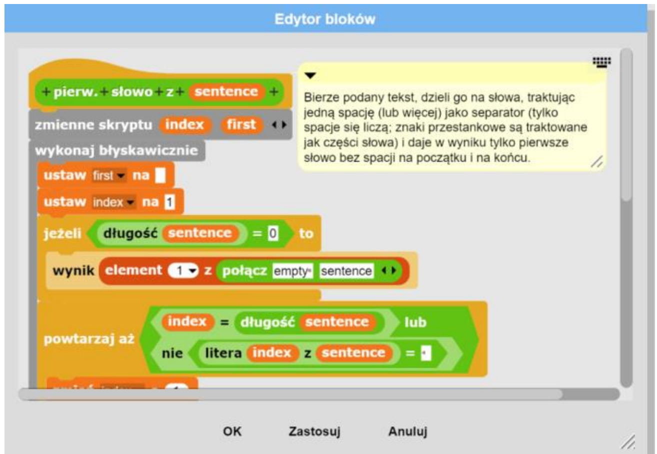


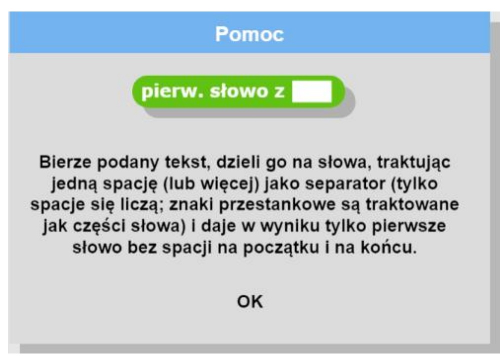

Opcja **ukryj** usuwa ten blok z palety. (Ta opcja jest dostępna tylko po kliknięciu bloku w samej palecie, a nie w skrypcie). Celem tej opcji jest umożliwienie nauczycielom prezentowania uczniom uproszczonego Snap*!* gdzie niektóre funkcje zostały skutecznie usunięte. Ukrycie pierwotnych bloków jest zapisywane w każdym projekcie, więc uczniowie mogą załadować wspólny projekt i zobaczyć tylko żądane bloki.

Jeśli klikniesz prawym klawiszem myszy w palecie na bloku niestandardowym (zdefiniowanym przez użytkownika), zobaczysz takie menu:




Opcja **pomoc…** dla niestandardowego bloku wyświetla komentarz, jeśli jest on dołączony do kapelusza bloku niestandardowego w Edytorze bloku. Oto przykład bloku z komentarzem i wyświetlaniem pomocy:




Opcja **usuń definicję bloku...** prosi o potwierdzenie, a następnie usuwa blok niestandardowy oraz usuwa go ze wszystkich skryptów, w których się on pojawia. (Wynik tego usunięcia może pozbawić skrypt sensu, najlepiej jest znaleźć i poprawić takie skrypty przed usunięciem bloku). Zauważ, że nie ma opcji, ukrycia niestandardowego bloku.

Opcja **edytuj...** otwiera Edytor bloków z definicją niestandardowego bloku.

## Menu kontekstowe tła palety

Kliknięcie prawym przyciskiem myszy (lub z ctrl) na szarym tle obszaru palety powoduje wyświetlenie tego menu:

znajdź bloki... ^F
ukryj pierwotne
pokaż pierwotne

Opcja **znajdź bloki...** działa tak samo jak przycisk lupy. Opcja **ukryj pierwotne** ukrywa wszystkie bloki pierwotne w tej palecie. Opcja **pokaż pierwotne**, która jest w menu tylko jeśli niektóre bloki pierwotne tej palety są ukryte, odkrywa je wszystkie.

## C Obszar skryptów

Obszar skryptów jest pionowym obszarem w środku okna Snap*!*, zawierającym skrypty, a także kilka elementów sterujących wyglądem i zachowaniem duszka. Zawsze istnieje aktualny duszek, którego skrypty są wyświetlane w obszarze skryptów. Ciemnoszary (*w prostym wyglądzie jasnoszary*) prostokąt w zagrodzie duszków pokazuje, który duszek (lub scena) jest wybrany. Zauważ, że tylko widoczny obszar skryptów jest „aktualny" dla duszka; wszystkie skrypty wszystkich duszków mogą być uruchomione w tym samym czasie. Kliknięcie na miniaturkę duszka w zagrodzie duszków sprawia, że staje się on aktualny (*wybrany*). Również scena może zostać wybrana jako aktualna, w którym to przypadku wygląd jest nieco inny.

### Wygląd duszka i kontrolki zachowania

W górnej części obszaru skryptów znajduje się obrazek duszka i kilka kontrolek jego zachowania:

Zauważ, że obraz duszka odzwierciedla jego kierunek, jeśli taki istnieje. Istnieją trzy rzeczy, które można tutaj kontrolować:

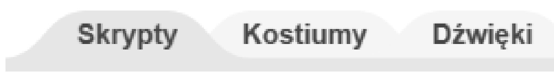
1. Trzy okrągłe przyciski (*w prostym wyglądzie - kwadratowe; WtK*) w kolumnie po lewej stronie kontrolują sposób obracania się duszka. Kostiumy duszków są zaprojektowane tak, aby w pozycji pionowej były zwrócone w prawo, gdy duszek jest skierowany w prawo (kierunek = 90). Jeśli wybrany jest najwyższy przycisk, domyślny, jak pokazano na powyższym obrazku, to kostium duszka obraca się, gdy duszek zmienia kierunek. Jeśli wybierzesz środkowy przycisk, kostium obraca się tylko w lewo lub prawo i zostanie odwrócony w lewo, gdy kierunek duszka jest mniej więcej w lewo (kierunek pomiędzy 180, a 359 lub, co jest równoważne, pomiędzy -180, a -1). Jeśli zostanie wybrany dolny przycisk, orientacja kostiumu nie zmieni się bez względu na kierunek duszka.

2. Nazwę duszka można zmieniać w polu tekstowym, które na tym obrazku zawiera nazwę „Duszek".

3. Na koniec, jeśli zaznaczone jest pole wyboru **przeciąganie**, to użytkownik może przesunąć duszka na scenie, klikając go i przeciągając. Ta funkcja jest najczęściej wykorzystywana w projektach z grami, w których niektóre duszki mają być kontrolowane przez gracza, ale inne nie.

### Zakładki pola skryptów

Tuż pod kontrolkami duszka znajdują się trzy zakładki określające, co widać w polu skryptów:




### Skrypty i bloki w skryptach

Większość z tego, co opisano w tej sekcji, dotyczy również bloków i skryptów w Edytorze bloków.

Kliknięcie skryptu (co również oznacza pojedynczy nie podłączony blok) uruchamia go. Jeśli skrypt zaczyna się od bloku kapelusza, kliknięcie skryptu uruchamia go, nawet jeśli zdarzenie w bloku kapelusza się nie zdarzyło (Jest to przydatna technika debugowania, gdy masz tuzin duszków i każdy z nich ma pięć skryptów z blokami z zieloną flagą i chcesz wiedzieć, co robi jeden z tych skryptów). Skrypt będzie otoczony przez

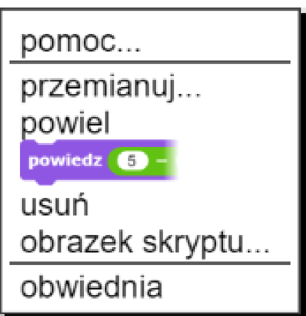
zielone „halo”, gdy jest uruchomiony. Jeśli skrypt jest dzielony z klonami, to gdy ma zieloną obwódkę, będzie również miał licznik, mówiący jak wiele instancji skryptu jest uruchomionych. Kliknięcie skryptu z taką obwódką zatrzymuje skrypt. (Jeśli skrypt zawiera blok **wykonaj błyskawicznie**, który może znajdować się w niestandardowym bloku używanym w skrypcie, Snap*!* może nie reagować natychmiast na kliknięcia.)

Jeśli skrypt jest wyświetlany z czerwoną obwódką, oznacza to, że w tym skrypcie został uchwycony błąd, na przykład użyto listy, tam gdzie potrzebna była liczba lub odwrotnie. Kliknięcie skryptu spowoduje wyłączenie halo.

Jeśli jakieś bloki zostały przeciągnięte do obszaru skryptów, to w prawym górnym rogu zobaczysz przycisk *cofnięcia* i / lub *ponowienia*, którego można użyć do cofnięcia albo przywrócenia wstawienia bloku lub skryptu. Po cofnięciu wstawienia do pola wejściowego przywracane jest wszystko, co kiedyś znajdowało się w polu. Przycisk *ponowienia* pojawia się po użyciu *cofnięcia*.

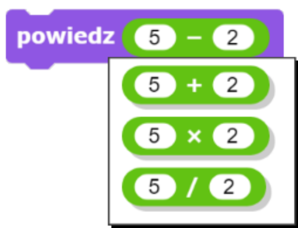
Trzeci przycisk uruchamia tryb edycji z klawiatury (rozdział D, strona 96).

Kliknięcie prawym przyciskiem myszy (lub z ctrl) na pierwotnym bloku w skrypcie powoduje wyświetlenie menu podobnego do tego:




Opcja **pomoc...** pokazuje ekran pomocy dla bloku, tak jak w palecie. Pozostałe opcje pojawiają się tylko wtedy, gdy blok jest kliknięty prawym przyciskiem myszy / kliknięty z ctrl w obszarze skryptów.

Nie każdy pierwotny blok ma opcję **przemianuj...**. Gdy jest obecna, pozwala na zastąpienie bloku innym, podobnym blokiem, zachowując wyrażenia wejściowe. Na przykład, oto, co dzieje się po wybraniu polecenia **przemianuj...** dla operatora arytmetycznego:




Zauważ, że wejścia do istniejącego bloku są również wyświetlane w menu alternatyw. Kliknij blok w menu, aby go wybrać, lub kliknij poza menu, aby zachować oryginalny blok.

Opcja **powiel** tworzy kopię całego skryptu, zaczynając od wybranego bloku. Kopia jest dołączona do myszy i możesz przeciągnąć ją do innego skryptu (lub nawet do innego okna Edytora bloków), nawet jeśli nie trzymasz przycisku myszy. Kliknij myszką, aby upuścić skrypt.

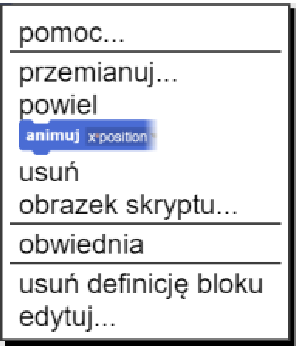
Obrazek bloku pod słowem **powiel** jest kolejną opcją powielania, ale duplikuje tylko wybrany blok, a nie wszystko pod nim w skrypcie. Zauważ, że jeśli wybrany blok jest blokiem kontrolnym w kształcie litery C, uwzględniony jest skrypt w jego wejściu w kształcie litery C. Jeśli blok znajduje się na końcu skryptu, ta opcja nie pojawia się.

Opcja **usuń** usuwa wybrany blok ze skryptu.

Opcja **obrazek skryptu...** otwiera nową kartę przeglądarki zawierającą obraz całego skryptu, a nie tylko od wybranego bloku do końca, w niektórych przeglądarkach bezpośrednio zapisuje obraz. W pierwszym przypadku możesz użyć funkcji Zapisz w przeglądarce, aby umieścić obrazek w pliku. Jest to bardzo przydatna funkcja, jeśli piszesz podręcznik Snap*!*! (Jeśli masz wyświetlacz Retina, rozważ wyłączenie wsparcia Retina przed wykonaniem obrazka skryptu, a jeśli nie, to może on być ogromny).

Jeśli skrypt nie zaczyna się od bloku kapelusza lub kliknięta została funkcja, jest jeszcze jedna opcja: **obwiednia** (i jeżeli wokół bloku lub skryptu jest już szara obwiednia: **bez obwiedni**). **Obwiednia** otacza blok (funkcję) lub cały skrypt (polecenie) szarą obwiednią, co oznacza, że blok(i) wewnątrz obwiedni są same danymi, jako dane wejściowe do procedury wyższego rzędu, a nie czymś, co należy obliczyć w ramach skryptu. Patrz Rozdział VI, Procedury jako dane.

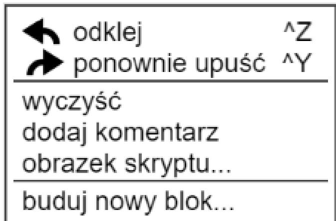
Kliknięcie *niestandardowego* bloku w skrypcie daje podobne, ale inne menu:




Opcja **przemianuj...** dla niestandardowych bloków pokazuje menu innych niestandardowych bloków o tym samym kształcie z tymi samymi wejściami. Obecnie nie można przemianować bloku niestandardowego na blok pierwotny lub odwrotnie. Dwie opcje u dołu, tylko dla bloków niestandardowych, są takie same, jak te w palecie.

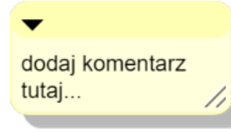
### Menu kontekstowe tła w obszarze skryptów

Kliknięcie z wciśniętym klawiszem ctrl / shift tła obszaru skryptów daje to menu:




Opcja **odklej** jest rodzajem funkcji „cofnij" dla częstego przypadku upuszczenia bloku gdzie indziej niż było zamierzone. Zapamiętuje ona wszystkie przeciągania i upuszczania, które zostały zrobione w obszarze skryptów tego duszka (to znaczy, że inne duszki mają własną oddzielną pamięć upuszczania), i cofa ostatnie, przywracając blok do poprzedniej pozycji i przywracając poprzednią wartość w odpowiednim polu wejściowym, jeśli takie istnieje. Gdy coś cofniesz, pojawi się opcja **ponownie upuść** i możesz powtórzyć operację, którą właśnie została cofnięta. Te opcje są równoważne przyciskom i opisanym wcześniej.

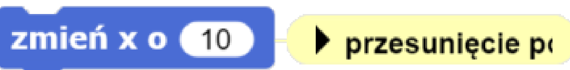
Opcja **wyczyść** przestawia skrypty tak, aby znajdowały się w kolumnie, z tym samym lewym marginesem i jednolitym odstępem między skryptami. To dobry pomysł, jeśli nie możesz przeczytać własnego projektu!




Opcja **dodaj komentarz** umieszcza pole komentarza, takie jak obrazek po prawej, w obszarze skryptów. Jest on podłączony do myszy, podobnie jak w przypadku powielania skryptów, więc ustawiasz mysz w miejscu, w którym chcesz umieścić komentarz i klikasz, aby go zwolnić. Następnie możesz edytować tekst w komentarzu według potrzeb.

Możesz ciągnąć dolny prawy róg pola komentarza, aby zmienić jego rozmiar. Kliknięcie grotu strzałki w lewym górnym rogu powoduje zmianę pola w formę kompaktową, jednoliniową, tak jak poniżej, dzięki czemu możesz mieć wiele zwiniętych komentarzy w obszarze skryptów i po prostu rozwinąć jeden z nich, gdy chcesz przeczytać go w całości.

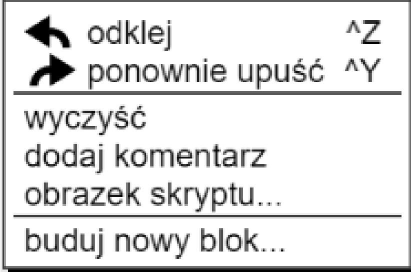
Jeśli przeciągniesz komentarz do bloku w skrypcie, komentarz zostanie dołączony do bloku żółtą linią:




Komentarze mają własne menu kontekstowe o oczywistym znaczeniu:

(***comment pic*** *– obrazek komentarza, pewnie wkrótce będzie poprawiony; WtK*)

Wróćmy do menu kontekstowego tła w obszarze skryptów:

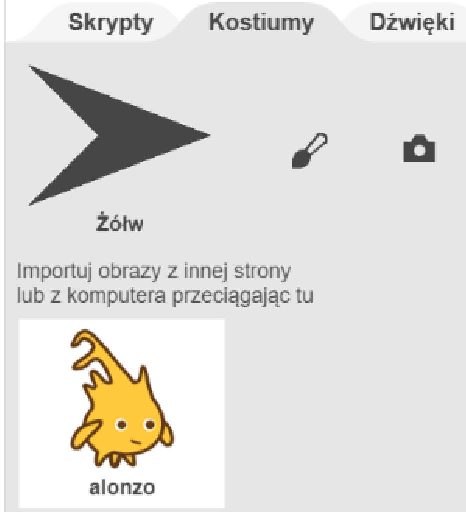



Opcja **obrazek skryptu...** otwiera nową kartę przeglądarki ze obrazkiem wszystkich skryptów w obszarze skryptów, tak jak się pojawiają, ale bez szarego tła w paski. Zauważ, że „wszystkie skrypty w obszarze skryptów" oznaczają tylko skrypty najwyższego poziomu bieżącego duszka, a nie inne skrypty lub definicje niestandardowych bloków.

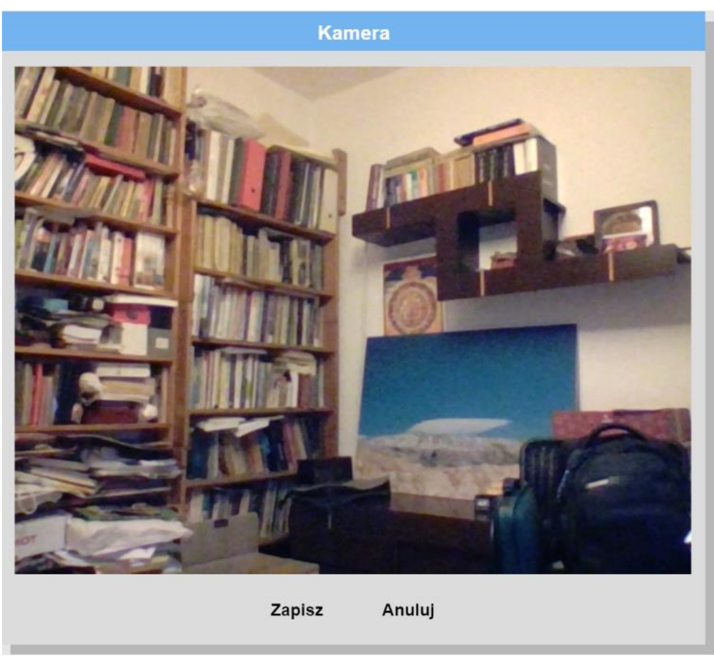
Wreszcie opcja **buduj nowy blok...** działa tak samo, jak przycisk „nowy blok" w paletach. Jest to skrót, więc nie musisz przewijać palety, jeśli tworzysz wiele bloków.

## Kontrolki w zakładce Kostiumy

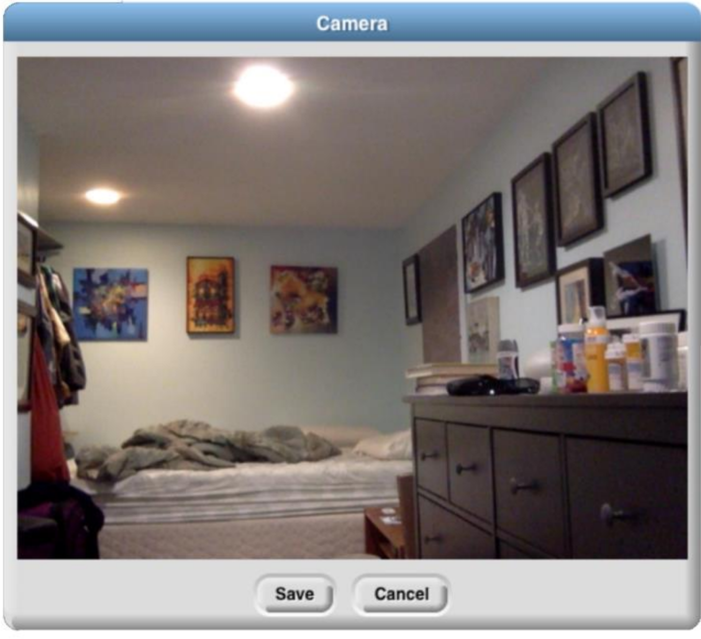
Jeśli klikniesz słowo „Kostiumy" pod kontrolkami duszka, zobaczysz coś takiego:



Kostium **Żółw** jest zawsze obecny, ma go każdy duszek; jest to kostium numer 0. Inne kostiumy można namalować w Snap*!* lub importować z plików lub innych kart przeglądarki, jeśli obsługuje je przeglądarka. Kliknięcie kostiumu wybiera go; to znaczy, duszek będzie wyglądać tak, jak wybrany kostium. Kliknięcie na ikonę pędzla otwiera edytor obrazów, w którym możesz stworzyć nowy kostium. Kliknięcie na ikonę aparatu otwiera okno, w którym widzisz, to co widzi aparat (*kamera*) twojego komputera i możesz zrobić zdjęcie (które będzie w pełnym rozmiarze sceny, chyba że pomniejszysz je w edytorze Paint). Działa to tylko jeśli dasz Snap*!* pozwolenie na korzystanie z aparatu, a może tylko wtedy, gdy otworzyłeś Snap*!* w trybie bezpiecznym (HTTPS), a następnie tylko wtedy, gdy Twoja przeglądarka Cię lubi.

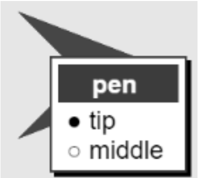



*Pokój do pracy, w którym powstało to tłumaczenie; WtK.*




*Brian's bedroom when he's staying at Paul's house.*

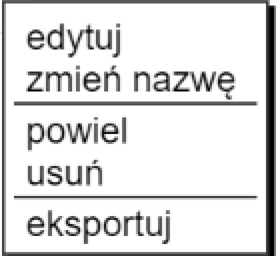
Kliknięcie prawym przyciskiem myszy na obrazie żółwia daje następujące menu:




W tym menu wybierasz punkt obrotu żółwia, który jest również punktem, przyłożenia pisaka którym żółw rysuje linie. Poniższe dwa obrazki pokazują wygląd sceny po narysowaniu kwadratu w każdym trybie; czubek (**tip** inaczej zwany „trybem Jensa") znajduje się po lewej stronie na obrazkach poniżej, środek (**middle** „tryb Briana") po prawej:

Jak widzisz, „czubek" oznacza przedni koniec grotu strzałki; „środek" nie jest środkiem zacienionego regionu, ale faktycznie środkowym z czterech wierzchołków, tym wklęsłym. (Jeśli kształt byłby prostym trójkątem równoramiennym, a nie fantazyjną strzałką, oznaczałoby to punkt środkowy tylnej krawędzi). Zaletą trybu **czubek** jest to, że jest mniej prawdopodobne, że duszek zasłoni rysunek. Zaletą trybu **środek** jest to, że punkt obrotu duszka rzadko znajduje się na czubku, a studenci mniej się mylą co do tego, co się stanie, jeśli poprosisz żółwia, aby obrócił się o 90 stopni względem pokazanej pozycji. (Jest to również tradycyjny punkt obrotu żółwia w Logo, które zapoczątkowało ten styl rysowania).

Kostiumy inne niż żółw mają inne menu kontekstowe:

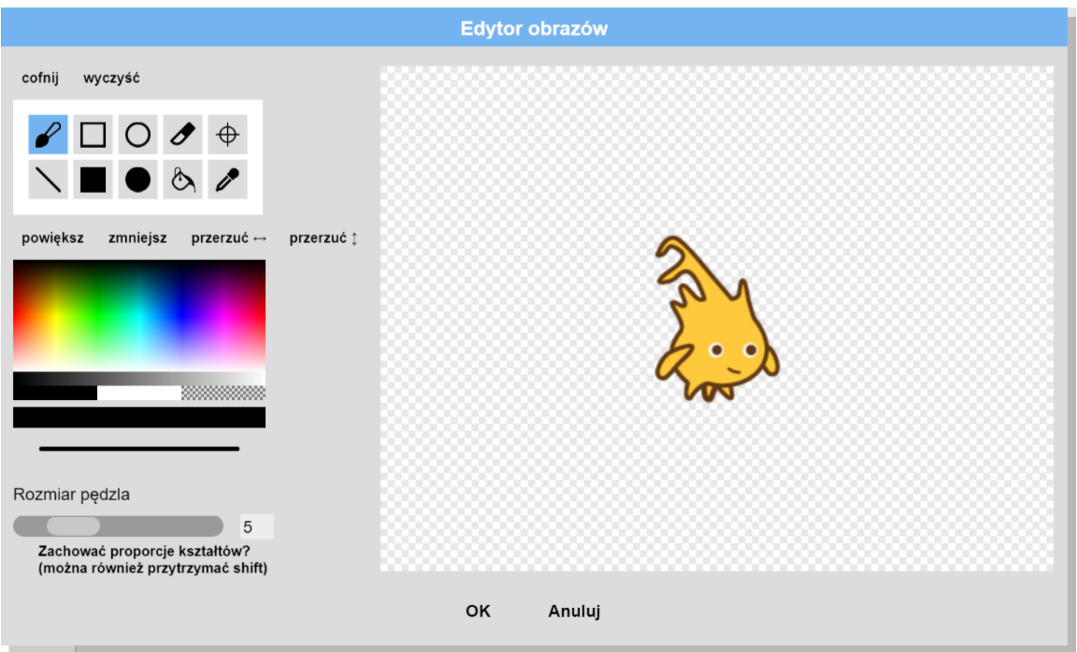



Opcja **edytuj** otwiera Edytor obrazów dla tego stroju. Opcja **zmień nazwę** otwiera okno dialogowe, w którym możesz zmienić nazwę kostiumu. (Pierwotna nazwa kostiumu pochodzi z pliku, z którego został on zaimportowany, lub jest czymś podobnym do **kostium5**). **Powiel** tworzy kopię kostiumu dla tego samego duszka. (Prawdopodobnie zrobisz to, ponieważ zamierzasz edytować jedną z kopii). **Usuń** jest oczywiste. Opcja **eksportuj** otwiera nową kartę przeglądarki ze obrazkiem kostiumu. Możesz zapisać go do pliku lub wybrać innego duszka w karcie Snap*!*, wróć do karty obrazka i przeciągnąć kostium do karty Snap*!*, aby skopiować kostium do innego duszka.

Możesz przeciągać kostiumy w górę i w dół w zakładce Kostiumy, w celu przenumerowania ich, aby blok **następny kostium** działał zgodnie z twoimi preferencjami.

## Edytor obrazów

Oto obrazek okna Edytora obrazów:




Jeśli korzystasz z programu do malowania, większość z tego będzie ci znana. Obecnie można edytować tylko kostiumy mapy bitowej (jpg, png itp.), a nie kostiumy wektorowe (svg itp.). W przeciwieństwie do Edytora bloków, na raz tylko jedno okno edytora malowania może być otwarte.

Dziesięć kwadratowych przycisków w dwóch rzędach po pięć w górnej lewej części okna to *narzędzia*. Górny rząd, od lewej do prawej, to pędzel, prostokąt – obwódka, elipsa – obrys, gumka i punkt obrotu. W dolnym rzędzie mamy narzędzie do rysowania linii, wypełniony prostokąt, wypełniona elipsa, narzędzie do zamalowania i pipeta. Poniżej narzędzi znajduje się rząd czterech przycisków, które natychmiast zmieniają obraz. Pierwsze dwa zmieniają jego ogólną wielkość; następne dwa odwracają obraz poziomo lub pionowo. Poniżej znajduje się paleta kolorów, taśma w odcieniach szarości i większe przyciski do czarnej, białej i przezroczystej farby. Niżej znajduje się pasek wyświetlający aktualnie wybrany kolor. Jeszcze niżej znajduje się obrazek linii pokazujący szerokość pędzla do malowania i rysowania, a poniżej można ustawić szerokość pędzla za pomocą suwaka lub wpisując w polu tekstowym liczbę (w pikselach). Na końcu pole wyboru ogranicza narzędzie linii do rysowania w poziomie lub w pionie, narzędzie prostokąt do rysowania kwadratów i narzędzie elipsa do rysowania okręgów. Możesz uzyskać ten sam efekt tymczasowo, przytrzymując klawisz Shift, co powoduje, że zaznaczenie pojawia się w polu tak długo, jak długo go przytrzymujesz (ale klawisz Caps Lock nie ma na to wpływu).

Możesz poprawić błędy za pomocą przycisku **cofnij**, który usuwa ostatnią narysowaną rzecz, lub przycisku **wyczyść**, który usuwa cały obraz. (Pamiętaj, że nie powrócisz do wyglądu kostiumu, przed rozpoczęciem edytowania! Jeśli tego chcesz, kliknij przycisk **Anuluj** u dołu edytora). Aby zachować zmiany, po zakończeniu edycji kliknij **OK**.

Zauważ, że narzędzia elipsy działają bardziej intuicyjnie niż te w innych programach, z których możesz korzystać. Zamiast przeciągać między przeciwległymi narożnikami prostokąta obejmującego pożądaną elipsę, tak że punkty końcowe przeciągania nie mają oczywistego połączenia z rzeczywistym kształtem, w Snap*!* zaczynasz od środka elipsy, którą chcesz uzyskać i przeciągasz do krawędzi. Po zwolnieniu przycisku kursor myszy znajdzie się na krzywej. Jeśli przeciągniesz od środka pod kątem 45 stopni do osi, wynikowa krzywa będzie kołem; jeśli przeciągniesz bardziej poziomo lub pionowo, elipsa będzie bardziej ekscentryczna.

(Oczywiście, jeśli chcesz mieć dokładny krąg, możesz trzymać wciśnięty klawisz Shift lub zaznaczyć pole wyboru). Narzędzia prostokąta działają jednak tak, jak oczekujesz: zaczynasz od rogu żądanego prostokąta i przeciągasz do przeciwległego rogu.

Za pomocą narzędzia pipeta możesz kliknąć w dowolnym miejscu w Snap*!* w oknie, nawet poza edytorem malowania, a narzędzie wybierze kolor pod kursorem myszy do użycia w edytorze obrazów. Możesz to zrobić tylko raz, ponieważ edytor obrazów automatycznie wybiera narzędzie pędzel po wybraniu koloru. (Oczywiście możesz ponownie kliknąć przycisk narzędzia pipeta).

Jedynym innym nieoczywistym narzędziem jest narzędzie ustawiania środka obrotu. Pokazuje ono, gdzie znajduje się w edytorze malowania bieżące centrum obrotu duszka (punkt, wokół którego się obraca, gdy używasz bloku **obróć**); po kliknięciu lub przeciągnięciu obrazka punkt środka obrotu zostanie przesunięty do miejsca kliknięcia. (Możesz to zrobić, na przykład, jeśli chcesz, żeby postać mogła machać ręką, więc używasz dwóch duszków połączonych ze sobą, chcesz, żeby punkt obrotu ręki duszka był na końcu, gdzie łączy się ona z ciałem, pozostanie więc przyczepiona do ciała podczas machania).

### Kontrolki w zakładce Dźwięki

W Snap*!* nie ma edytora dźwięku, a także brak aktualnego dźwięku duszka, choć ma on aktualny strój (Duszek zawsze ma wygląd, chyba że jest ukryty, ale nie śpiewa, chyba że został o to jawnie poproszony). Tak więc menu kontekstowe dla dźwięków ma (*po wczytaniu dźwięku; WtK*) tylko opcje zmiany nazwy i usuwania oraz przycisk oznaczony odpowiednio Graj lub Stop. Jeśli potrzebujesz edytora dźwięku, rozważ http://audacity.sourceforge.net.

## D Edytowanie z klawiatury

Wciąż trwają badania nad tym, jak sprawić, by wizualne języki programowania były dostępne dla ludzi z niepełnosprawnością wzrokową lub motoryczną. Jako pierwszy krok w tym kierunku zapewniamy edytor klawiatury, dzięki czemu można tworzyć i edytować skrypty bez śledzenia myszy. Do tej pory nie każdy element interfejsu użytkownika jest sterowany przez klawiaturę, a my nawet nie zaczęliśmy zapewniać obsługi wyjścia, takiej jak połączenie z syntezatorem mowy. To jest obszar, o którym wiemy, że mamy jeszcze długą drogę! Ale to jest początek. Edytor klawiatury może być przydatny również dla każdego, kto może pisać szybciej, niż przeciągać bloki.

### Rozpoczynanie i kończenie edytowania z klawiatury

Istnieją trzy sposoby na uruchomienie edytora klawiatury. **Kliknięcie z** naciśniętym klawiszem **Shift** w dowolnym miejscu obszaru skryptu spowoduje uruchomienie edytora: edytowanie istniejącego skryptu lub, jeśli klikniesz tło, edycję nowego skryptu na pozycji myszy. Alternatywnie, naciśnięcie **shift-enter** spowoduje uruchomienie edytora w istniejącym skrypcie i możesz użyć klawisza tabulacji do przejścia do innego skryptu. Możesz też kliknąć przycisk klawiatury u góry obszaru skryptów.

Gdy edytor skryptów jest uruchomiony, jego położenie jest reprezentowane przez migający biały pasek:

kiedy kliknięto
powtórz 4 razy
przesuń o coś + kroków
obróć o 90 stopni

Aby opuścić edytor klawiatury, naciśnij klawisz Escape lub po prostu kliknij tło obszaru skryptów.

## Nawigacja podczas edytowania z klawiatury

Aby przejść do innego skryptu, naciśnij klawisz **tab**. **Shift-tab**, aby poruszać się po skryptach w odwrotnej kolejności.

Skrypt to pionowy stos bloków poleceń. Blok poleceń może mieć pola wejściowe, a każde z nich może zawierać blok funkcji; funkcja może sama mieć pola wejściowe, które mogą zawierać inne funkcje. Możesz szybko poruszać się po skrypcie, używając klawiszy **strzałek w górę** i **w dół**, aby przechodzić między blokami poleceń. Po znalezieniu bloku polecenia, który chcesz edytować, klawisze **strzałek w lewo** i w **prawo** przemieszczają między elementami edytowalnymi w tym poleceniu. (Strzałka w lewo i w prawo, gdy nie ma więcej edytowalnych elementów w ramach bieżącego bloku poleceń, przesunie się odpowiednio w górę lub w dół do innego bloku poleceń). Oto sekwencja obrazów pokazująca wyniki powtarzania strzałek w prawo od pozycji pokazanej powyżej:

Możesz zmienić z klawiatury kolejność skryptów w obszarze skryptów. Wciskanie klawiszy **strzałek** i **shift** (w lewo, w prawo, w górę lub w dół) spowoduje przesunięcie bieżącego skryptu. Jeśli przeniesiesz go na inny skrypt, to nie będą się one ze sobą łączyć; ten, który poruszasz, będzie nakładał się na ten, który już tam jest. Oznacza to, że możesz przejść poprzez inny skrypt, aby uzyskać dostęp do wolnego miejsca.

## Edytowanie skryptu

Zwróć uwagę, że fokus edycji z klawiatury, punkt pokazany jako biały pasek lub halo, znajduje się pomiędzy dwoma blokami poleceń lub w polu wejściowym. Klawisze edycji robią nieco inne rzeczy w każdym z tych dwóch przypadków.

Klawisz **backspace** usuwa blok. Jeśli fokus jest pomiędzy dwoma poleceniami, to znajdujące się *przed* (nad) migającym paskiem zostanie usunięte. Jeśli fokus jest w polu wejściowym, funkcja w tym polu zostaje usunięta. (Jeśli to pole wejściowe ma wartość domyślną, pojawi się ona w polu). Jeśli fokus znajduje się na wejściu wieloparametrowym (takim, który może zmienić liczbę wejść poprzez kliknięcie strzałki), wówczas jedno wejście zostanie usunięte. (Po wejściu w strzałkę w prawo w tym wypadku, fokus najpierw zakrywa całą rzecz, w tym groty strzałek, kolejna strzałka w prawo fokusuje na pierwszym polu w tej grupie wejściowej, fokus jest „na wejściu wieloparametrowym", gdy obejmuje całą rzecz.)

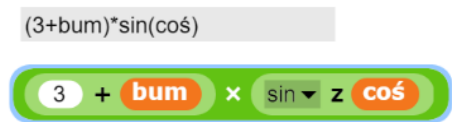
Klawisz **enter** nie robi niczego, jeśli fokus znajduje się pomiędzy poleceniami lub na funkcji. Jeśli fokus znajduje się na wejściu wieloparametrowym, klawisz enter dodaje jeszcze jedno pole wejściowe. Jeśli fokus znajduje się na białym polu wejściowym (takim, które nie zawiera funkcji), klawisz Enter wybiera to pole wejściowe do edycji; to znaczy, możesz wpisać w nie wartość, tak jak gdy pole wejściowe zostało kliknięte. (Oczywiście, jeśli fokus znajduje się na polu wejściowym zawierającym funkcję, możesz użyć klawisza backspace, aby usunąć tą funkcję, a następnie użyć klawisza enter, aby wpisać w nim wartość). Po zakończeniu wpisywania wartości ponownie naciśnij klawisz enter, aby je zaakceptować i wrócić do nawigacji, lub klawisz escape, jeśli zdecydujesz się nie zmieniać wartości znajdującej się w polu.

Klawisz **spacji** służy do wyświetlenia menu możliwości gdy fokus jest na wejściu. Nie robi nic, chyba że fokus znajduje się na pojedynczym polu wejściowym. Jeśli fokus znajduje się na polu z rozwijanym menu opcji, klawisz spacji pokazuje to menu. (Jeśli jest to pole w kolorze bloku, co oznacza, że można użyć tylko opcji w menu, klawisz enter spowoduje to samo, ale jeśli jest to białe okienko z menu, na przykład w blokach

**obróć**, wtedy enter wprowadza wartość, a spacja pokazuje menu). W przeciwnym razie klawisz spacji pokazuje menu zmiennych dostępnych w tym miejscu skryptu. W obu przypadkach użyj klawiszy strzałek w górę i w dół, aby poruszać się po menu, użyj klawisza enter, aby zaakceptować podświetloną pozycję, lub użyj klawisza escape, aby opuścić menu bez wybierania opcji.

Naciśnięcie **dowolnego innego** klawisza **znaku** (nie specjalnych klawiszy na fantazyjnych klawiaturach, które robią coś innego niż generowanie znaku) aktywuje paletę wyszukiwania bloków. Ta paleta, do której dostęp można uzyskać również, wpisując ctrl-F lub cmd-F poza edytorem klawiatury, ma u góry pole tekstowe, a następnie bloki, których tytuł zawiera wpisywany tekst. Znak wpisany w celu uruchomienia palety wyszukiwania bloków jest wprowadzany w polu tekstowym, więc zaczynasz od palety bloków zawierających ten znak. W obrębie palety bloki, których tytuły *zaczynają się* od tekstu, który wpisujesz, są na początku, następnie są bloki, w których słowo w tytule zaczyna się od wpisanego tekstu, a na końcu bloki, w których tekst pojawia się dalej w słowie tytułu. Po wpisaniu wystarczającej ilości tekstu, aby wyświetlić żądany blok, użyj klawiszy strzałek, aby przejść do tego bloku na palecie, następnie enter, aby wstawić ten blok, lub escape, aby opuścić paletę wyszukiwania bloków bez wstawiania bloku. (Kiedy nie używasz edycji z klawiatury, zamiast poruszać się za pomocą klawiszy strzałek, przeciągasz blok, którego potrzebujesz, do skryptu, jak z dowolnej innej palety).

Jeśli wpiszesz operator **arytmetyczny** (+ **-** * /) lub **operator porównania** (<=>) do pola tekstowego wyszukiwania bloku, możesz wpisać dowolnie skomplikowane wyrażenie, a zbiór arytmetycznych bloków operatora zostanie skonstruowany tak, aby pasował:




Jak pokazuje przykład, można również użyć **nawiasów** do grupowania, a operatory nie liczbowe są traktowane jako zmienne lub funkcje pierwotne. (Wprowadzona w ten sposób nazwa zmiennej może już istnieć lub nie w skrypcie, tylko **zaokrąglij** i te z rozwijanego menu bloku **pierwiastek kwadratowy** mogą być używane jako nazwy funkcji).

### Uruchamianie wybranego skryptu

Naciśnij ctrl-shift-enter, by uruchomić skrypt, na którym jest fokus, jest to jakby kliknięcie skryptu.

## E Kontrolki sceny

Scena to obszar w prawym górnym rogu ekranu Snap*!* okno, w którym poruszają się duszki.

Większość duszków można przenosić, klikając je i przeciągając. (Jeśli odznaczono pole wyboru **przeciąganie** dla duszka, przeciąganie go nie da żadnego efektu). Kliknięcie z wciśniętym klawiszem ctrl / kliknięcie prawym przyciskiem myszy powoduje wyświetlenie tego menu kontekstowego:

powiel
klonuj
usuń
porusz
oś
edytuj
eksportuj...

Opcja **powiel** tworzy innego duszka z tymi samymi skryptami, tymi samymi strojami itp., jak ten. Nowy duszek tworzy się w losowo wybranej pozycji innej niż oryginał, dzięki czemu można szybko zobaczyć, który jest który. Nowy duszek zostaje wybrany: staje się bieżącym duszkiem, pokazanym w obszarze skryptowym. Opcja **klonuj** tworzy stały klon tego duszka i wybiera go.

Opcja **usuń** usuwa duszka. Nie zostaje on po prostu ukryty; znika na dobre. Opcja **edytuj** wybiera duszka. W rzeczywistości mimo nazwy nic nie zmienia w duszku, chodzi o to, że obszar skryptowy dotyczy teraz tego duszka.

Opcja **porusz** pokazuje „uchwyt ruchu" wewnątrz duszka (kwadrat w poprzeczne paski w środku):

Zazwyczaj można po prostu chwycić i przesunąć duszka bez tej opcji, ale są dwa powody, dla których może być ona potrzebna: Po pierwsze, działa, nawet jeśli pole wyboru „przeciąganie" powyżej obszaru skryptu jest odznaczone. Po drugie, działa na duszka, który jest częścią zakotwiczoną w innym; zwykle przeciąganie części przesuwa całego zagnieżdżonego duszka.

Opcja **oś** pokazuje celownik wewnątrz ikonki:

Możesz klikać i przeciągać celownik w dowolne miejsce na scenie, aby ustawić punkt obrotu kostiumu. (Jeśli przesuniesz go poza duszka, obrócenie duszka spowoduje jego obieg jak i obrót wokół osi). Po zakończeniu kliknij scenę nie krzyżyk. Zauważ, że w przeciwieństwie do przesuwania punktu obrotu w edytorze rysunków, ta technika nie powoduje widocznego przesunięcia duszka na scenie. Zamiast tego zmienią się wartości **pozycja X** i **pozycja Y**.

Opcja **edytuj** powoduje wybranie duszka, podświetlając go w zagrodzie i wyświetlając jego obszar skryptów. Jeśli duszek był tymczasowym klonem, staje się trwały.

Opcja **eksportuj...** otwiera nową kartę przeglądarki zawierającą tekstową reprezentację duszka. (Nie tylko jego kostium, ale wszystkie jego kostiumy, skrypty, lokalne zmienne i bloki oraz inne właściwości). Możesz zapisać tę kartę w pliku na swoim komputerze, a następnie zaimportować duszka do innego projektu. (W niektórych przeglądarkach duszek jest bezpośrednio zapisywany do pliku).

Kliknięcie z ctrl / kliknięcie prawym przyciskiem myszy na tle sceny (to znaczy w dowolnym miejscu na scenie za wyjątkiem duszka) pokazuje własne menu kontekstowe:

edytuj
pokaż wszystko
obrazek...
ślady pisaka

Opcja **edytuj** sceny wybiera scenę, więc skrypty i tła scen są widoczne w obszarze skryptów. Zwróć uwagę, że po wybraniu sceny niektóre bloki, szczególnie te z palety Ruch, nie znajdują się w obszarze palety, ponieważ scena nie może się poruszać.

Opcja **pokaż wszystko** sprawia, że wszystkie duszki są widoczne, zarówno w sensie bloku **pokaż**, jak i sprowadzenia duszka na scenę, jeśli przesunął się poza krawędź sceny.

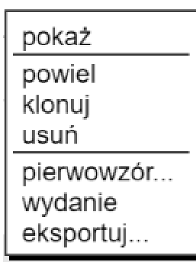
Opcja **obrazek...** otwiera zakładkę przeglądarki z obrazem wszystkiego na scenie: jego tłem, liniami narysowanymi za pomocą pisaka i wszystkimi widocznymi duszkami. To co widzisz, jest tym co dostajesz. (Jeśli chcesz uzyskać tylko obraz tła, wybierz scenę, otwórz zakładkę **Tła**, kliknij, przytrzymując klawisz ctrl lub kliknij prawym przyciskiem tło i wyeksportuj je).

Opcja **ślady pisaka** tworzy nowy kostium dla aktualnie wybranego duszka składający się ze wszystkich linii narysowanych na scenie za pomocą pisaka dowolnego duszka.

### Zagroda duszków i przyciski tworzenia duszka

Pomiędzy sceną a zagrodą duszków w prawym dolnym rogu okna Snap*!* jest ciemnoszary (*w prostym wyglądzie – biały; WtK*) pasek zawierający trzy przyciski. Służą one do stworzenia nowego duszka. Pierwszy przycisk tworzy duszka z samym strojem żółwia, z losowo wybraną pozycją, kierunkiem i kolorem pisaka. Drugi przycisk tworzy duszka i otwiera Edytor obrazów, dzięki czemu możesz zrobić dla niego własny kostium. (Oczywiście możesz kliknąć pierwszy przycisk, a następnie kliknąć przycisk malowania w zakładce **Kostiumy**, ten przycisk malowania jest skrótem do tego wszystkiego). Podobnie, trzeci przycisk wykorzystuje, jeśli to możliwe twoją kamerę, aby zrobić kostium dla nowego duszka.

W zagrodzie duszków, klikasz na obrazek „miniaturkę” duszka, aby wybrać tego duszka (aby był tym, którego skrypty, kostiumy itp. są pokazane w obszarze skryptów). Możesz przeciągać miniaturki duszków (ale nie sceny), aby zmienić ich kolejność; nie ma to specjalnego wpływu na projekt, ale pozwala na przykład umieścić obok siebie powiązane elementy. Możesz kliknąć miniaturkę duszka prawym przyciskiem myszy / z wciśniętym ctrl, aby uzyskać dostęp do tego menu kontekstowego:




Opcja **pokaż** sprawia, że duszek jest widoczny, jeśli był ukryty, a także przenosi go na scenę, jeśli wyszedł poza granicę sceny. Następne trzy opcje są takie same, jak w menu kontekstowym wybranego duszka na scenie, omówionym powyżej.

Opcja **pierwowzór...** wyświetla menu wszystkich innych duszków, pokazując, który z nich jest rodzicem duszka i pozwalając wybrać innego duszka (zastępując istniejącego rodzica). Opcja **wydanie**, która jest wyświetlana tylko wtedy, gdy ten duszek jest klonem (stałym, bo inny nie byłby w zagrodzie duszków); zmienia duszka na tymczasowy klon. (Nazwa ma oznaczać, że duszek jest zwalniany z zagrody [*to musimy poprawić tłumaczenie; WtK*]). Opcja **eksportuj...** eksportuje duszka, tak jak ta sama opcja na scenie.

Menu kontekstowe miniatury sceny ma tylko jedną opcję, **obrazek...**, która robi obrazek wszystkiego na scenie, tak jak ta sama opcja w menu kontekstowym tła sceny.

## F Załadowanie projektu gdy zaczynamy Snap!

Istnieje kilka sposobów dołączenia w adresie URL wskaźnika do projektu w celu automatycznego załadowania projektu podczas uruchamiania Snap*!*. Możesz myśleć o takim adresie URL, jako raczej o uruchomieniu projektu, a nie o uruchomieniu Snap*!*, szczególnie jeśli URL mówi, aby rozpocząć w trybie prezentacji i kliknąć zieloną flagę. Ogólna postać to:

http://snap.berkeley.edu/run#***verb***:***project***&***flag***&***flag***...

„verb” powyżej może być dowolnym z wyrazów: `open`, `run`, `cloud`, lub `present`. Dwa ostatnie są dla wspólnych projektów znajdujących się w chmurze Snap*!*; pierwsze dwa dotyczą projektów, które zostały wyeksportowane i udostępnione w dowolnym miejscu w Internecie.

Oto przykład, który ładuje projekt przechowywany na stronie internetowej Snap*!* (nie w chmurze Snap*!*):

```
http://snap.berkeley.edu/run#open:http://snap.berkeley.edu/snapsource/Examples/vee.xml
```

Plik projektu zostanie otwarty i Snap! uruchomi się w trybie edycji (z widocznym programem). Użycie #run: zamiast #open: otworzy tryb prezentacji (z widoczną tylko sceną) i „uruchomi" projekt klikając zieloną flagę. („uruchomi" jest w cudzysłowie, ponieważ nie ma gwarancji, że projekt zawiera jakiekolwiek skrypty wyzwalane przez zieloną flagę. Niektóre projekty są uruchamiane przez naciśnięcie klawisza na klawiaturze lub kliknięcie duszka).

Jeśli verb to `run`, możesz również użyć dowolnego podzestawu następujących czterech flag:

| | |
|---|---|
| `&editMode` | Zacznij w trybie edycji, nie w trybie prezentacji. |
| `&noRun` | Nie klikaj zielonej flagi. |
| `&hideControls` | Nie pokazuj rzędu przycisków nad sceną (tryb edycji, zielona flaga, pauza, stop). |
| `&noExitWarning` | Kiedy zamykasz okno lub ładujesz inny adres URL, nie pokazuj komunikatu przeglądarki: „Chcesz opuścić tę stronę?". |

Ostatnia z tych flag jest przeznaczona do użycia na stronie internetowej, w której okno Snap*!* jest osadzone.

Oto przykład, który ładuje wspólny (publiczny) projekt z chmury Snap*!*:

```
http://snap.berkeley.edu/run#present:Username=jens&ProjectName=tree%20animation
```

(Zauważ, że „`Username`" i „`ProjectName`" są pisane jak w tytule, mimo że flagi takie jak „`noRun`" są pisane w stylu wielbłąd). Należy również zauważyć, że spacja w nazwie projektu musi być reprezentowana w Unicode jako `%20`). Słowo `present` zachowuje się jak `run`: zwykle uruchamia projekt w trybie prezentacji, ale jego zachowanie można zmodyfikować za pomocą tych samych czterech flag, co w przypadku użycia `run`. Czasownik (verb) `cloud` (tak, wiemy, że nie jest to czasownik w zwykłym znaczeniu) zachowuje się jak `open`, z wyjątkiem tego, że ładuje z Chmury Snap*!* a nie ogólnie z Internetu.

## G Strony lustrzane

Jeśli strona `snap.berkeley.edu` będzie nieosiągalna, możesz załadować Snap*!* z dowolnej z następujących stron lustrzanych:

- `http://bjc.edc.org/snapsource/snap.html`
- `http://media.mit.edu/~harveyb/snap`
- `http://cs10.org/snap`

# Indeks


**!** blok · 80
# 1 · 47
# zmienna · 78
( **i** ), blok · 78
( **lub** ), blok · 78
<zmienna> **z** <duszek> · 52
➤-kształtne gniazda wejściowe · 82

A
**a jeżeli**, blok · 79
Abelson, Hal · 3
Advanced Placement Computer Science Principles · 76
AGPL · 73
aktualna data lub czas · 64
aktualny duszek · 89
Alonzo · 8
animacja · 11
animacji biblioteka · 81
**animuj**, blok · 81
anonimowa lista · 28
Arduino · 64
arytmetyka · 10
**assoc**, blok · 78
atrybut · 54
atrybut rodzic · 55
atrybuty, lista · 56

B
Ball Michael · 3
**bez obwiedni** · 44, 58
**bez pierwszego z** strumień · 78
**bez pierwszego z**, blok · 31
Bezpieczne skrypty wątków, opcja · 86
**bezpiecznie próbuj**, blok · 80
biblioteka bloków · 27
biblioteka bloków · 27, 76
biblioteka funkcji zachłannych · 78
biblioteka iteracji · 77
biblioteka kolorów pisaka · 80
Biblioteka LEAP Motion · 79
biblioteka liczb całkowitych nieskończonej precyzji · 80
biblioteka metody, listy · 78
biblioteka narzędzi · 4, 12, 19, 31
biblioteka pikseli · 82
biblioteka pobierania/ ustawiania wartości · 80
biblioteka pobierania/ustawiania wartości systemowych · 80
biblioteka przechwytywania błędów · 80
biblioteka słowa i zdania · 79
biblioteka słów i zdań · 79
biblioteka strumieni (leniwych list) · 78
biblioteka usług internetowych · 78
biblioteka wiele linii · 80
biblioteka wieloargumentowych poleceń warunkowych · 79
biblioteka wieloargumentowych poleceń warunkowych · 79
**Biblioteki ...**, opcja · 77
bitmapa · 83
bjc.edc.org · 101
blok · 5; polecenia (komenda) · 5;  C-kształtny · 6;
blok adresów URL · 63, 78
blok bez nazwy · 19
blok bez nazwy · 19, 81
Blok funkcji · 10
blok kapeluszowy · 5, 24
blok kapeluszowy · 5; predykat · 11; funkcja (reporter) · 10;
blok poleceń (komend) · 5
blok w kształcie litery C · 6, 45
Bloki predykatów · 11
bloki, kolor · 23
**błąd**, blok · 80
Boole, George · 11
Boolean · 11
brązowa kropka · 9
BYOB · 23

C
Chandra, Kartik · 3
Church, Alonzo · 8
Computer Science Principles · 76
**cond** w Lisp · 79
control-shift-enter (edytor klawiatury) · 98
CORS · 64
CPS · 68
cs10.org · 101
CSV · 36
czas · 64
czerwone halo · 46, 47, 90
części (w bloku **ja**) · 56
części (zagnieżdżonego duszka) · 9
Czujniki · 63

D
dane pierwszej klasy · 28
data · 64
Dave, Achal · 3
debugowanie · 16, 87
**Debugowanie krokowe**, opcja · 85
delegowanie · 59
Dinsmore, Nathan · 3
**dla**, blok · 12, 19, 41, 43, 77
**Długa forma dialogu wejścia**, opcja · 85
długie okno dialogowe nazwy wejścia · 37
dodaj komentarz, opcja · 91
domyślna wartość · 41
dowolny (nieobliczany) typ · 50
dowolny typ · 38
drzewo binarne · 29
duszek · 5
duszki · 51
duszki pierwszej klasy · 51
dziedziczenie · 51, 59
**dźwięki** (w bloku **ja**) · 56
**Dźwięki...**, opcja · 83

E
Edytor bloków · 24, 25, 37
Edytor obrazów · 93
edytor z klawiatury · 96
**edytuj**, opcja · 94, 99
**edytuj...**, opcja · 88
eksport projektu · 21

Eksport XML · 21
**eksport...**, opcja · 100
**Eksportuj bloki...**, opcja · 76
**Eksportuj projekt...**, opcja · 76
**eksportuj**, opcja · 94, 99
**element 1 z** strumień, blok · 78
**element 1 z**, blok · 31
elementy interfejsu użytkownika · 73
**etykieta**, blok · 19

F
**fałsz**, blok · 18
Finch · 64
flaga, zielona · 5
fokus (edytor z klawiatury) · 97
forma specjalna · 50
formanty w zakładce Dźwięki · 96
funkcja asocjacyjna · 32
funkcja asocjacyjna · 32
funkcja tożsamościowa · 50
funkcja wygładzania · 81
funkcja wyższego rzędu · 48
funkcja, **jeżeli**, blok · 11
funkcja, **jeżeli…to…inaczej**, blok · 19
funkcje paska narzędzi · 73
funkcje rekurencyjne · 26

G
garderoba · 8
getter (pobieracz) · 54
groty strzałek · 28, 41

H
halo · 10, 90; czerwone · 47
Hotchkiss. Kyle · 3
**HSV**, blok · 80
HTML (HyperText Markup Language) · 63
HTTP · 64
HTTPS · 64, 93
Hudson, Connor · 3
Hummingbird · 64

I
**id**, blok · 50
**Ignoruj**, blok · 19
ikona aparatu · 93
ikona Chmury · 83
ikona pędzla · 93
ikona ustawień · 84
ikony w tekście tytułu · 42
**iloczyn**, blok · 78
**iloList**, blok budowany · 48
imperatywny styl programowania · 30
import narzędzi · 11
**Importuj narzędzia**, opcja · 76
**Importuj...**, opcja · 76
**inaczej**, blok · 79
Ingalls, Dan · 3
**inne duszki** (w bloku **ja**) · 56
**inne klony** (w bloku **ja**) · 56
instancja · 57
interakcja · 14
**ja**, blok · 51, 54

J
JavaScript · 18
**jest _ typu?**, blok · 18
**jeżeli … w przeciwnym razie**, blok · 49
**jeżeli … w przeciwnym razie**, blok funkcji · 19
**jeżeli**, blok · 11
**Język ...**, opcja · 84
Język programowania Java · 46
język tekstowy · 86

K
**kaskada**, bloki · 77
kawałki układanki · 23, 38
Kay, Alan · 3
klasa / instancja · 55
klasa · 57
Klasa licznik · 57
klasa potomek · 59
klasa rodzic · 59
klawiatura fortepianu · 18
klawiatura fortepianu · 18
klawisz Backspace (edytor klawiatury) · 97
klawisz Enter (edytor z klawiatury) · 97
klawisz Escape (edytor z klawiatury) · 96
klawisz spacji (edytor z klawiatury) · 97
klawisz tabulatora (edytor z klawiatury) · 97
klawisze strzałek z shift (edytor z klawiatury) · 97
kliknięcie na skrypcie · 89
kliknięcie opcji dźwięku · 85
kliknięcie z shift · 73
**klon z**, blok · 61
klony (w bloku **ja**) · 56
klony: stałe · 52; tymczasowe · 52
**klucz…wartość**, blok · 78
kodowanie bloków, opcja · 86
kolor bloków · 23
Kolorowanie w zebrę · 11
kontrola wyglądu i zachowania duszka · 89
Kontrola, paleta · 6
kontrolki na scenie · 98
kontrolki w zakładce Kostiumy · 92
kontynuacja · 65
kontynuacja styl przekazywania · 66
konwersja między ciągami tekstowymi i listami · 19
**kopia z**, blok · 82
kostium · 5, 7
kostiumy (w bloku **ja**) · 56
Kostiumy ... opcja · 82
kotwica (w bloku **ja**) · 56
kotwica · 9
kształty bloków · 23
kształty bloków · 23, 38
kształty pól wejścia · 37

L
Laboratorium Sztucznej Inteligencji MIT · 3
lambda · 45
Leap Motion · 64
**leć**, blok · 85
Lego NXT · 64

licencja · 73
**liczba Scheme**, blok · 80
**liczby od…do**, blok · 19
Lifelong Kindergarten Group · 3
linkowana lista · 31
lista asocjacyjna · 60
lista dwóch elementów (x, y) · 18
lista procedur · 49
**lista**, blok · 28
lista, linkowana · 31
listy list · 29
lody · 75
**Logout** (Wyloguj), opcja · 84
**Logowanie...**, opcja · 84
lokalna zmienna duszka · 13
lokalny blok duszka · 53
ładowanie zapisanych projektów · 22

M
magazyn lokalny · 20
Maloney, John · 3
małe ludziki · 26, 68
**mapuj na strumień**, blok · 78
**mapuj**, blok · 32, 43
Massachusetts Institute of Technology · 3
McCarthy, John · 3
Media Lab · 3
media.mit.edu · 101
menu ikony pliku · 74
menu kontekstowe dla bloków palety · 87
menu kontekstowe tła obszaru skryptów · 91
menu kontekstowe tła palety · 88
metod tabela · 60
metoda · 51, 53, 58
miniaturka (duszka) · 89
MIT Media Lab · 3
**moja lokalizacja**, blok · 78
Morphic · 3
Motyashov, Ivan · 3
Myślenie rekurencyjne · 27

N
**nadaj do wszystkich i czekaj**, blok · 8, 53
**nadaj do wszystkich**, blok · 51
narzędzie do rysowania linii · 95
narzędzie do wypełniania, · 95
narzędzie elipsa · 95
narzędzie elipsa- obrys· 95
narzędzie gumka · 95
narzędzie pędzel · 95
narzędzie prostokąt · 95
narzędzie prostokąt- obwódka· 95
narzędzie wypełnionego prostokąta · 95
narzędzie wypełnionej elipsy · 95
**nazwa** (w bloku **ja**) · 56
nazwa parametru, dialog · 25
nazwa użytkownika · 21
nazwa wejścia · 47
nazwa, pole· 89
nazwy parametrów · 47
**niech**, blok · 80
nielokalne wyjście · 71
nieobliczane typy procedur · 39
niestandardowy blok w skrypcie · 91
**Niewykorzystane bloki...**, opcja · 76
Nintendo · 64
**nowy blok ...**, opcja · 92
Nowy blok · 23
**nowy blok**, przycisk · 87
**nowy klon**, blok · 55
nowy przycisk duszka · 7
**Nowy**, opcja · 74

O
**o Snap!,** opcja · 73
**obecnie**, blok · 64
obiekty, budowanie jawnie · 57
obiekty, duszki · 51
**obrazek ...** opcja · 99, 100
obrazek bloku, opcja · 90
**obrazek skryptu** ..., opcja · 91
**obrazek skryptu...** (obszar skryptów), opcja · 92
obrót · 89
obrót części (duszka) · 9
**obrót x** (w bloku **ja**) · 56
**obrót y** (w bloku **ja**) · 56
obszar palety · 87
obszar skryptów · 5, 89
obwiednia · 44, 46
**obwiednia**, opcja · 91
**odklej**, opcja · 91
**odwróć**, blok · 78
ogólny blok kapeluszowy · 5
okno dialogowe nazwy wejścia · 25, 37
Okno edytora obrazów · 95
opcja anulowania · 91
Opcja Uwagi do projektu · 74
operator rekurencyjny · 49
**ostatni**, bloki · 79
**oś**, opcja · 99
**Otwórz...**, opcja · 74
owalne bloki · 23, 38

P
paleta · 5
paleta kolorów · 95
pamięć · 15
para klucz-wartość · 60
Parallax · S2 · 64
parametry formalne · 47
pasek narzędzi · 5
pasek wyszukiwania · 75
**pierw. słowo z**, blok · 79
pierwotny blok w skrypcie · 90
piksele w bloku · 82
pipeta, narzędzie · 95, 96
plakietka podglądu zmiennej · 14
pliki źródłowe Snap! · 74
**Płaskie końce linii**, opcja · 86
**Pobierz źródło**, opcja · 74
podgląd zmiennej · 14
podmenu · 40
podpowiedź · 41
**Podręcznik**, opcja · 74

**Podtrzymywanie dziedziczenia**, opcja · 86
**podziel**, blok · 63
Pojedyncze wejście · 41
pojedynczy krok · 17
**pokaż obraz**, blok · 82
**pokaż pierwotne**, opcja · 89
**pokaż strumień**, blok · 78
pokaż wewnętrzną zmienną · 41
**pokaż wszystko**, opcja · 99
**pokaż zmienną**, blok · 16
**pokaż**, opcja · 100
pole komentarza · 91
pole wejścia zmiennej · 46
polecenie przerwania · 71
polimorfizm · 53
**połącz słowa**, blok · 19
**połącz**, blok · 78
pomarańczowy owal · 12
**pomoc...** opcja · 88, 90
**pomoc...** opcja dla niestandardowego bloku · 88
**ponownie upuść**, opcja · 91
**porusz**, opcja · 99
potomstwo (w bloku **ja**) · 56
potraktuj tym, blok · 32
**powiedz**, blok · 53
**powiel**, opcja · 90, 94, 98
**Powiększ bloki…**, opcja · 84
**powtarzaj aż**, blok · 11
**powtórz**, blok · 6, 45, 77
**pozycja X**, blok · 10
**pozycja Y**, blok · 10
praca krokowa · 86
**prawda**, blok · 18
procedura · 11, 44
procedura wysyłająca · 57, 58, 60
procedura wyższego rzędu · 44
programowanie obiektowe · 51, 57
**Prosty opis prototypu**, opcja · 85
**Prosty wygląd**, opcja · 85
prototyp · 24
prototyp, pomoc · 41
prototypowanie · 54, 55, 60
przechowywanie w chmurze · 21
**przeciąganie**, pole wyboru · 89, 98
przekazywanie wiadomości · 51, 58
**przemianuj...**, opcja · 90, 91
przycisk Anuluj · 95
przycisk Chmura · 21, 74
przycisk cofania · 90, 95
przycisk edycji z klawiatury · 90
przycisk pauzy · 16, 87
przycisk pliku · 11
przycisk ponów · 90
przycisk pracy krokowej · 17
przycisk pracy krokowej · 86
przycisk przeglądarki · 74
przycisk **stop** · 87
przycisk trybu prezentacji · 87
przycisk wyszukiwania · 87
przycisk zmniejszania / powiększania · 86
przycisk: **pauza** · 16; **praca krokowa** · 17
przyciski obracania · 89
przyciski sterowania projektem · 87
przyciski tworzenia duszków · 100
Przyciski zmiany rozmiaru sceny · 86
**Przykłady**, przycisk · 75
przypadek podstawowy · 26
punkt obrotu żółwia · 94
punkt obrotu, narzędzie · 95, 96
punkt przerwania · 16, 87
puste pola wejściowe, wypełnianie · 44, 46, 48
**puste?**, blok · 78

R
**Rejestracja...**, opcja · 84
rekurencja · 26, 106
Reynolds, Ian · 3
**RGB**, blok · 80
Roberts, Eric · 27
roboty · 63, 64
rodzic (w bloku **ja**) · 56
**rodzic ...** opcja · 100
rotacja synchroniczna · 9
**Rozmiar sceny...**, opcja · 84
Równoległość (procesów) · 7

S
**sam** (w bloku **ja**) · 56
**sąsiedzi** (w bloku **ja**) · 56
scena (w bloku **ja** · 56
scena · 5, 51
Scheme · 3
Scratch · 4, 8, 23, 28, 29, 30, 37
Scratch Team · 3
separator: menu · 40
seter (ustalacz) · 54
shift-enter (edytor z klawiatury) · 96
Shift-tab (edytor z klawiatury) · 97
silnia · 26, 49, 80
**sito**, blok · 78
skrót · 92, 100
skróty klawiaturowe · 74
skróty: klawiatura · 74
skrypt · 4
**słowo**, bloki · 19
Snap*!* menu logo · 73
Snap*!* podręcznik · 91
Snap*!* program · 4
Snap*!* **strona internetowa**, opcja · 74
snap.berkeley.edu · 74
**sortuj**, blok · 78
squiral  (kwadrala) · 12
stałe funkcje · 50
stan lokalny · 51
Stanford Artificial Intelligence Lab · 3
Steele, Guy · 3
stos bloków · 6
struktura danych · 29
Struktura i interpretacja programów komputerowych · 3
**strumień**, blok · 78
strzałka skierowana ku górze · 41
strzałka w dół (edytor z klawiatury) · 97
strzałka w górę (edytor z klawiatury) · 97

strzałka w lewo (edytor z klawiatury) · 97
strzałka w prawo (edytor z klawiatury) · 97
strzałka, wskazanie w górę · 41
styl programowania funkcyjnego · 30
**suma**, blok · 78
Super-Awesome Sylvia · 64
Sussman, Gerald J. · 3
Sussman, Julie · 3
suwak: szybkość kroku · 17
**Suwaki wejściowe**, opcja · 85
switch w C · 79
symbol pinezki · 53
symbole w tekście tytułu · 42
symulacja · 51
szafa grająca · 8
szczelina w kształcie litery C · 50
sześciokątne bloki · 11, 23, 38
ślady pisaka, blok · 18
ślady pisaka, opcja · 100
**środek**, opcja · 94

T

tabela danych · 60
tablica dynamiczna · 31
tablica, dynamiczna · 31
tajemnice · 73
tekst tytułu · 25
**tip** (czubek), opcja · 94
Tła..., opcja · 83
tło palety · 88
tłumaczenie · 84
trwały klon · 52, 100
**Tryb Turbo**, opcja · 85
tymczasowy klon · 52, 99
typ · 18
Typ boolowski (nieaktualizowany) · 50
typ danych · 18, 37
Typ liczbowy · 38
typ listy · 38
typ nieobliczany · 50
Typ obiektu · 38
Typ procedury · 50
Typ tekstu · 38

U

**uaktualnij**, blok · 82
**Udostępnij** · 75
układ okien · 4
ukryj i pokaż pierwotne · 17
**ukryj pierwotne**, opcja · 89
**ukryj zmienną**, blok · 16
**ukryj**, opcja · 88
ukrywanie danych · 51
Uniform Resource Locator · 63
upvar · 41
**uruchom z kontynuacją**, blok · 70
**uruchom**, blok · 43, 46
urządzenia · 63, 64
urządzenia fizyczne · 63
**ustaw flagę**, blok · 80
**ustaw wartość**, blok · 80
**ustaw**, blok · 14
**ustawienie**, blok · 80
**ustąp**, blok · 72
**usuń definicję bloku...**, opcja · 88
usuń duplikaty z bloku · 78
**usuń zmienną**, przycisk · 14
**usuń**, opcja · 90, 94, 99
Utwórz listę · 28
Utwórz zmienną · 13
**użyj WIELKIELICZBY**, blok · 80

W

wartości rozdzielone przecinkami · 36
Warunki świadczenia usług · 22
Water Color Bot· 64
wątek · 72
**wątek**, blok · 72
wejście · 6
wejście dla listy · 46
wejście rozwijane · 39
wejście rozwijane tylko do odczytu · 39
wektor · 83
wiadomość · 51
widok listy · 33
widok tabeli · 33
**wiele linii**, blok · 80
Wiele wejść · 41
**WIELKIELICZBY**, Blok · 80
Wiimote · 64
wiszący? (w bloku **ja**) · 56
witryny lustrzane · 101
World Wide Web · 63
wprowadzenie tekstu · 9
**Wsparcie wyświetlacza Retina**, opcja · 84
**wstaw … przed strumień**, blok · 78
**wstaw … przed**, blok · 31
wstępne ładowanie projektu · 100
**wszystkie bez**…, bloki · 79
**wybierz**, blok · 79
wyczyść przycisk · 95
wyczyść, opcja · 91
**wydanie** (uwolnij), opcja · 100
wygląd, okno · 4
**wygładzanie**, blok · 81
**wykonaj błyskawicznie**, blok · 18, 90
**Wykonaj przy zmianie suwaka**, opcja · 85
**Wyłącz udostępnianie** · 75
**wynik**, blok · 26
wyrafinowanie · 50
wyrażenie · 11
**wywołaj … z kontynuacją**, blok · 69
**wywołaj**, blok · 43, 46
wywołania zagnieżdżone · 48
wywołanie rekurencyjne · 46

X

x środka (w bloku **ja**) · 56
Xerox PARC · 3

Y

y środka (w bloku **ja**) · 56
Yuan, Yuan · 3

Z
**z parametrami** · 44
**z…**, blok · 52, 53
**zachowaj … ze strumienia**, blok · 78
**zachowaj**, blok · 31
**zacznij**, blok · 53
zaczynając Snap*!* · 100
zagnieżdżanie duszków · 9
Zagnieżdżone duszki · 9
Zagroda duszków · 7, 100
zakładka Kostiumy · 8, 92
zakres leksykalny · 57
zakres leksykalny · 57
zapisywalne wejścia rozwijane · 39
**Zapisz jako...**, opcja · 75
**Zapisz…**, opcja · 75
**zapytaj**, blok · 53, 58
zasada projektowania · 28, 55
**zatrzymaj** blok · 16, 87
**zatrzymaj blok**, blok · 27
**zatrzymaj wszystko**, blok · 87
**zdanie** bloki · 19
zestaw bloków koloru pisaka · 80
zielona flaga · 5, 87
zielone halo · 89
**złap**. blok · 71
**złożenie**, blok · 77
zmiana nazw zmiennych · 14
zmienna · 12, 54; globalna · 13; zmiana nazwy · 14; lokalna skryptu · 14; lokalna duszka · 13; chwilowa · 15
zmienna globalna · 13, 14
zmienna tymczasowa · 15
zmienna wewnętrzna · 41
**zmienne skryptu**, blok · 14, 58
zmienne w polach obwiedni · 44
Zmień hasło ... opcja · 84
**zmień nazwę**, opcja · 94
**znajdź bloki...**, opcja · 89
**Zresetuj hasło ...**, opcja · 84
**żółw**, kostium · 93