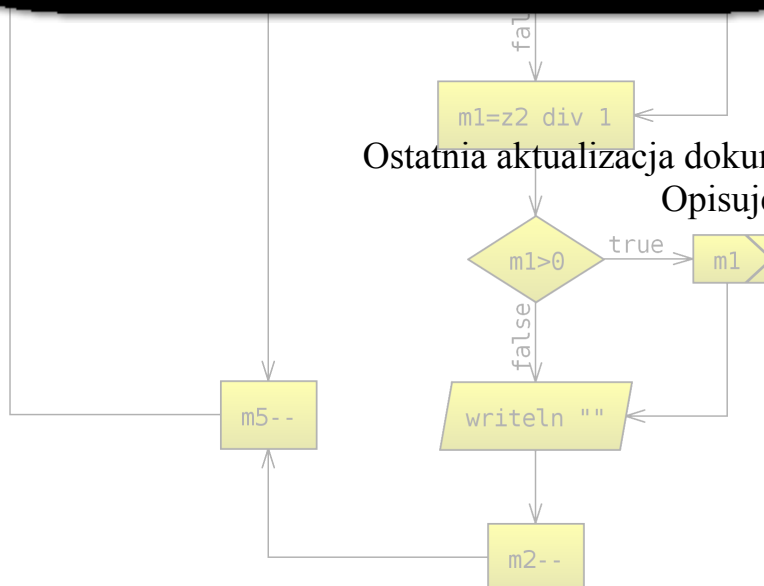


Schematy blokowe z

JavaBlock

Ostatnia aktualizacja dokumentu: 14.05.2011
Opisuje wersję: river 0.6



Spis Treści

1. Wstęp.....	4
1.1. O autorze.....	4
1.2. Dlaczego więc schemat blokowy miałby być lepszy?.....	4
1.3. Co dają schematy blokowe?.....	4
2. O programie.....	5
2.1. Do czego służy?.....	5
2.2. Czym więc wyróżnia się JavaBlock?.....	5
2.3. Założenia programu.....	6
2.4. Wymagania.....	6
2.5. Instalacja.....	7
3. Schemat blokowy.....	8
3.1. Słowniczek.....	8
3.2. Budowa.....	8
3.3. Typy bloków.....	9
3.4. Silniki symulatorów.....	10
3.5. Tryby edycji.....	11
3.6. Kilka zasad tworzenia czytelnych schematów.....	12
4. Obsługa programu.....	14
4.1. Interfejs.....	14
4.2. Poruszanie się po przestrzeni roboczej.....	15
4.3. Zaznaczanie.....	15
4.4. Tworzenie bloków.....	15
4.5. Łączenie bloków i usuwanie połączeń.....	15
4.6. Kolorowanie.....	15
4.7. Wyrównywanie.....	16
4.8. Grupowanie.....	16
4.9. Tryb pascala.....	16
5. Składnia, czyli tworzymy schemat.....	17
5.1. Tworzenie zmiennych.....	17
5.2. Przypisywanie wartości zmiennych.....	17
5.3. Operatory matematyczne.....	17
5.4. Dzielenie całkowite.....	18
5.5. Funkcje matematyczne.....	18
5.6. Operatory przypisania.....	20
5.7. Tworzenie i działania na tablicach.....	20
5.8. Wejście/wyjście.....	21
5.9. Warunki.....	22
6. Proste przykłady schematów.....	24
6.1. Zatem tworzymy. Przykład bez rozgałęzień.....	24
6.2. Tworzenie stringów.....	25
6.3. Pierwszy blok decyzyjny.....	26
6.4. Miejsca zerowe funkcji kwadratowej.....	28
6.5. Pętle.....	30
6.6. Klamra – jak działa.....	32
7. Symulacja schematu i szukanie błędów.....	33
7.1. Symulacja.....	33
7.2. Śledzenie wartości zmiennych.....	33
7.3. Interwał.....	34
7.4. Zakładka „Wejście”.....	34
8. Bloki deklaracyjne i Struktury.....	35
8.1. Struktury.....	35

8.2. Skrypt osadzony.....	36
9. Funkcje w JavaBlock.....	37
9.1. Co to jest Funkcja.....	37
9.2. Jak wywołać schemat jako funkcję?.....	37
9.3. Zasady używania funkcji.....	37
10. Konfiguracja.....	38
10.1. Ogólne.....	38
10.2. Kolory.....	39
10.3. Pastebin.com.....	39
11. Dzielenie się algorytmem.....	40
12. Dodatki.....	41
12.1. canvas2d.....	41
12.2. logo.....	42
13. Sędzia.....	43
14. Rozwiązywanie problemów.....	44
14.1. Jak uruchomić program?.....	44
14.2. Program nie uruchamia się.....	44
14.3. Program nie eksportuje schematów do pastebin.....	44
14.4. Błąd wykonania przy interwale równym 0.....	44
14.5. Błąd wykonywania schematu z wykorzystaniem innego schematu jako funkcji.....	44
14.6. Stary schemat nie uruchamia się.....	44
14.7. Błąd wykonania schematu mimo poprawności na innym komputerze.....	45
14.8. Wersja apletowa jest przestarzała.....	45
14.9. Program niemiłosiernie się tnie.....	45
15. Dla programistów.....	46
15.1. Dziennik zmian.....	46
15.2. Plany.....	48
15.3. JavaScript, a Python.....	48
15.4. Źródła.....	48
15.5. Pluginy – nowe obiekty.....	48
15.6. Skrypt inicjujący silnik JavaScript.....	48
15.7. Składnia pliku .jbf.....	49
16. Kontakt.....	50

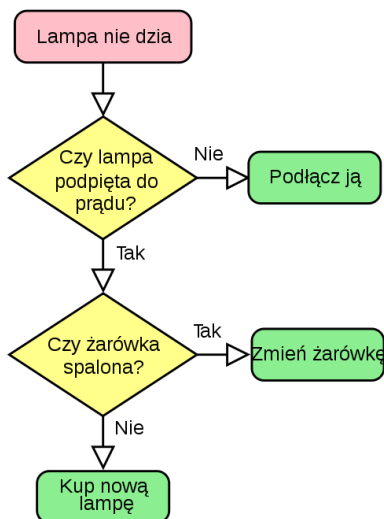
1. Wstęp

1.1. O autorze

Programowaniem zajmuję się od 5 lat. Najpierw był język skryptowy PHP. Dopiero 3 lata temu zacząłem się interesować programowaniem na poważnie. w szkole zaczynałem właśnie od schematów blokowych. Później był Pascal, C/C++, po drodze schematy blokowe, teraz Java... Ale uważam że jednak warto zaczynać od schematów blokowych. Dlaczego?

Oglądając różne tematy z prośbą o pomoc w programowaniu większość z nich (pomijając tematy typu „zróbcie za mnie”) to głównie problemy z samym językiem: biblioteki, źle zapisane klamry, źle ustawione warunki, średnik w niewłaściwym miejscu itd.

1.2. Dlaczego więc schemat blokowy miałby być lepszy?



Przykładowy schemat blokowy postępowania

W schemacie blokowym wszystko jest przedstawione graficznie. Nie da się po prostu popełnić niektórych błędów np. źle ustawić warunku: widać od razu co jest z czym połączone, co jest wywoływane. Programista skupia się wyłącznie na algorytmie, a nie bibliotekach, kompilatorach itd.

Kurs, podręcznik ten, ma na celu nauczyć tworzenia algorytmów z użyciem schematów blokowych w programie *JavaBlock*. Jako że jest to kurs dla potencjalnych przyszłych programistów, nie będę omawiał kompletnych podstaw obsługi komputera takich jak instalacja programu.

Niektórzy preferują tworzenie schematów na kartce i ręczną analizę. Jednak w przypadku pomyłki, lub chęci zmiany struktury trzeba dużo kreślić, albo przepisywać, a przy samej analizie można się łatwo pomylić. Oczywiście umiejętność ręcznej analizy również się przydaje, ale można stracić cierpliwość po 10 poprawce.

1.3. Co dają schematy blokowe?

Graficzne przedstawienie działania algorytmu. Łatwiej się analizuje schematy blokowe, które można przedstawić w przestrzeni dwuwymiarowej, niż sam kod. Początkujący programista widzi od razu co jest wykonywane przez co i co jest wykonywane później.

2. O programie

2.1. Do czego służy?

JavaBlock to program do tworzenia i symulowania (interpretowania, wykonywania) schematów blokowych mojego autorstwa. Nie jest to oczywiście jedyne takie narzędzie. Innymi popularnymi narzędziami są:

- **ELI** - Stary, ale dobry edytor schematów blokowych. Datuje się go na 1994, więc jest bardzo stary. Nie używałem go zbyt wiele. Oferuje oprócz tworzenia i symulowania schematów także obsługę urządzeń typu czujnik temperatury, natężenia światła itd., które mogą posłużyć za źródła danych do przetwarzania w algorytmie. Wygląda jednak niezbyt estetycznie.
- **Magiczne Bloczki** - mały, wydajny, o sporych możliwościach. Tworzy również pseudokod, który można nazwać tekstową wersją schematu blokowego. Używałem go do nauki schematów blokowych. Szczerze przyznam się że wzorowałem się na nim nieco, ale o tym później. Język interpretowany przypomina Basic.

Podsumowując: ELI - niewygodny, Magiczne Bloczki - wygodniejszy, ale nieładny efekt, oba - płatne.

Dlaczego *JavaBlock* jest darmowy? Jest to mój pierwszy poważny projekt w Javie, którą zacząłem się uczyć na początku Sierpnia. Poza tym jakby to napisać... program taki trudny do napisania nie jest.

Program możesz ściągnąć z:

<https://javablock.sf.net/install.jnlp> - Instalator online

<https://sourceforge.net/projects/javablock/> - główne wydania

<http://javablock.sf.net/> - strona główna

<http://javablock.sf.net/applet.php> - aplet (pełny obszar)

<http://javablock.sf.net/JavaBlock.jar> - najnowsza wersja, niekoniecznie stabilna

2.2. Czym więc wyróżnia się *JavaBlock*?

Przede wszystkim jest jeszcze rozwijany. Jest napisany w Javie, co zapewnia niezależność od platformy (tzn. działa tak samo jednocześnie na Windowsach, Linuksach, Macach itd.). Wymaga jedynie zainstalowanej Wirtualnej Maszyny Java.

Sam program udostępniony będzie na licencji GPL, dzięki czemu każdy doświadczony programista będzie mógł w łatwy sposób rozszerzyć możliwości programu i przysłużyć się jego rozwojowi. Na razie kod jest zamknięty i intensywnie rozwijany.

Poza standardowym tworzeniem schematów przewiduję także:

- tworzenie schematów z gotowych kodów językowych
- tworzenie pseudokodu
- struktury i obiektowość
- generowanie skryptów

Program nie ma nic wspólnego z UML! UML moim zdaniem jest formatem „zbyt ciężkim” do tak trywialnych zadań. W założeniu *JavaBlock* ma być lekki (w

założeniu sam program bez pluginów do 2MB) i być możliwie prosty w obsłudze.

2.3. Założenia programu

Program ma być z założenia nie tylko lekki, ale i prosty w obsłudze i pozwalać na programowanie zarówno bardziej doświadczonym programistom jak i tym zupełnie zielonym. Schematy blokowe pozwalają na kompletną swawolę w projektowaniu algorytmów i nie ma tradycyjnych pęteli. Bloki niekodowe zwalniają niemal całkowicie początkującemu programisty z konieczności poznania składni przed rozpoczęciem programowania.

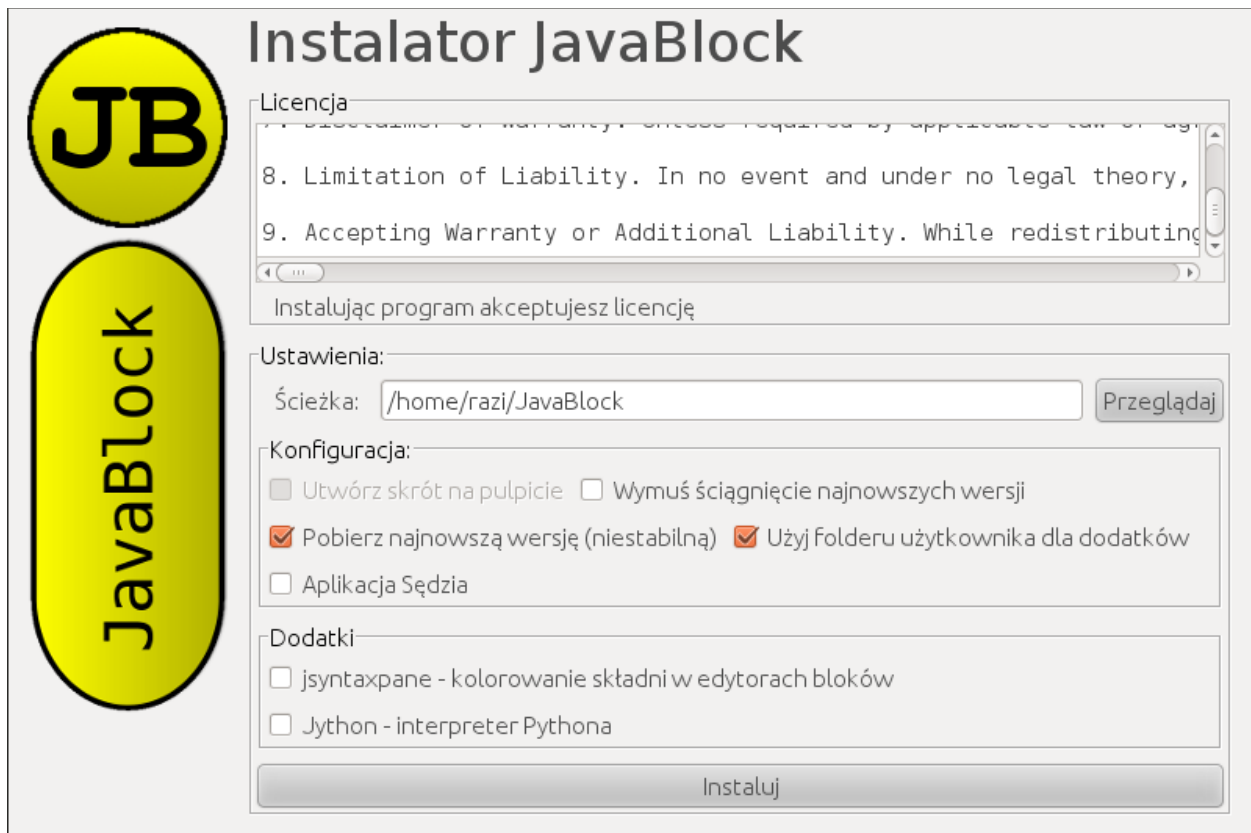
Sam program nie musi być używany tylko do programowania, można w nim tworzyć również schematy blokowe z programowaniem nie mające nic wspólnego.

2.4. Wymagania

- Procesor 1GHz
- Rozdzielczość 1024x600
- 256 MB RAM, zalecane 1GB
- 50 MB na HDD
- System dowolny z Javą SE 1.6
- Połączenie z internetem (sprawdzanie wersji, wysyłanie/ściągnięcie schematów z internetu, instalacja modułów dodatkowych)

2.5. Instalacja

Najprostszą metodą instalacji jest użycie instalatora online dostępnego pod [tym adresem](#). Musisz potwierdzić certyfikaty.



Okno instalatora

a) Licencja

Program JavaBlock jest na licencji Freeware, jednak moduły jsyntaxpane oraz jython są na licencjach Apache i GPL. Instalując program oraz te dodatki akceptujesz licencje.

b) Konfiguracja

- Wymuś ściągnięcie najnowszych wersji - jeśli są dostępne zasoby offline, ignoruje je pobierając program i dodatki z sieci
- Pobierz najnowszą wersję (niestabilną) - pobiera najnowszą wersję niekoniecznie stabilną, tzw. „nighty build”

c) Dodatki

- jsyntaxpane - poszerza możliwości edytora kodu, koloruje składnię adekwatnie do wybranego silnika skryptu
- jython - silnik skryptu (interpreter) używany do symulacji, jest znacznie szybszy od standardowego (JavaScript), ale za to waży 10MiB i nie działa w wersji apletowej.

3. Schemat blokowy

3.1. Słowniczek

- **Flowchart, Diagram Przepływu** – różne nazwy schematów blokowych. Przepływ, bo podczas wykonywania schematu jakby przepływamy od jednego bloku do drugiego.
- **Blok** – Podstawowy element schematów blokowych zawierających kod lub informację o swojej funkcji.
- **Połączenie** – strzałka łącząca bloki, wskazuje kierunek przepływu
- **„Blok a łączy się z blokiem b”** - po wykonaniu bloku *a* przechodzi do bloku *b*, czyli strzałka wychodząca z *a*.
- **„Blok a jest podłączony do bloku b”** - blok *a* jest wskazywany jako następny po *b*.
- **Blok oparty na kodzie**¹ – blok do którego użytkownik wpisuje kod
- **Blok nieoparty na kodzie** – blok którego kod jest generowany dzięki specjalnemu edytorowi, do którego użytkownik wprowadza dane, bloki te generują kod do wszystkich obsługiwanych silników skryptowych bez konieczności przerabiania kodu
- **Blok definiujący** – blok, który nie bierze bezpośredniego udziału w przepływie, ale są w nim wykorzystywane, np. zdefiniowane funkcje, struktury, klasy

3.2. Budowa

Schematy blokowe zbudowane są z tzw. bloków: figur geometrycznych będącymi „ramkami” dla tekstu (kodu lub komentarza). Każdy schemat musi mieć początek: blok startowy – owal z zazwyczaj tekstem „Start”.

Algorytm może się kończyć w wielu miejscach. Wynika to z tego, że bloki decyzyjne mają zawsze dwa wyjścia: w przypadku spełnionego warunku i niespełnionego warunku. Wtedy powstaje rozgałęzienie, które niekoniecznie kiedyś się jeszcze „spotyka”. Początek natomiast zawsze jest jeden.

Bloki są połączone strzałkami. Strzałka wskazuje, który blok będzie następny. Wszystkie standardowe bloki mogą mieć dowolną ilość strzałek wchodzących. Ze strzałkami wychodzącymi jest inaczej: większość musi mieć tylko jedną, blok zakończenia nie może mieć w ogóle, a blok decyzyjny musi mieć dwie.

Bloki mają różne kształty w zależności od ich funkcji.

¹ Nazwa robocza, jak coś wymyślę bardziej przystępną nazwę, zmienię

3.3. Typy bloków

a) Standardowe:



Rozpoczęcie algorytmu. Od niego cały algorytm się zaczyna, jeden schemat może posiadać tylko jeden taki blok. Deklarujemy w nim argumenty funkcji i typ zwracanej wartości. Również początek metody klasy (w przyszłości)



Wszelkie przetwarzanie danych: operacje matematyczne, tworzenie obiektów, wywoływanie metod/funkcji.



Wczytywanie i wypisywanie danych, oparte na kodzie. Posiada tryb nieoparty na kodzie (dwa bloki poniżej).



Wczytywanie danych, nieoparte na kodzie. w specjalnym edytorze wpisuje się jedynie nazwę zmiennej oraz komunikat wyświetlany przy wczytywaniu.



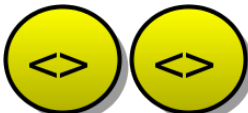
Wypisywanie zmiennej, w specjalnym edytorze wpisuje się tylko nazwę zmiennej i dodatkowy komunikat.



Sprawdza warunek i wywołuje odpowiedni blok w zależności od prawdziwości warunku. Posiada 2 połączenia wychodzące: true i false. Służy m.in. do tworzenia instrukcji warunkowych, oraz do tworzenia pętli.



Przeskok, Węzeł pomocniczy - służy jedynie do tworzenia łamanych połączeń, oraz do łączenia kilku połączeń w jedno. Nie wykonuje żadnego kodu, jest ignorowany przy symulacji. Jeżeli ten blok ma tylko jedno połączenie wchodzące, nie jest rysowane dzięki czemu można robić łamane połączenia.



Łącznik - służy do połączenia dwóch fragmentów algorytmu. Jeden wywołuje blok podłączony do drugiego, więc działa obustronnie. Dzięki niemu można uniknąć łamanych połączeń.

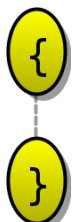


Nie może być podłączony do żadnego bloku, gdyż służy tylko do wyświetlania komentarzy.



Terminator - kończy wykonywanie algorytmu.

b) Niestandardowe:



Klamry - graficzne przedstawienie klamer obecnych w językach programowania, ułatwia wizualizację (dokładniejszy opis później).



Obsługa dodatków, tylko edytory niekodowe.

c) Definiujące:

Bloki definiujące to bloki które nie biorą bezpośrednio udziału w *przepływie*, lecz są definicjami używanymi w programie. Nie koniecznie należą do „standardu”².

Skrypt osadzony

Skrypt osadzony – zawiera gotowy napisany skrypt dla danego silnika skryptowego. Treść nie jest w żaden sposób zmieniana. Nie może być podłączony, ani łączyć się z innym blokiem (na chwilę obecną). Może również przechowywać globalne zmienne/obiekty

Struktura:
pole :Number

Struktura – definicja struktury – obiektu mogącego przechowywać kilka danych jednocześnie (w tym i inne struktury). Może generować także interfejs akcesorów (tzw. *getter* i *setter*) formatujących typ danych. W przyszłości przewiduję dodawanie metod.

3.4. Silniki symulatorów

JavaBlock oferuje 2 silniki skryptów, które są używane do interpretacji kodu i symulacji schematów. Są to JavaScript i Python.

Silniki te nie są w pełni ze sobą w pełni kompatybilne, różnią się nieco składnią kodu (nawet z poziomu bloku), przez co niektóre schematy (zwłaszcza te używające tablic) mogą nie działać w obu silnikach jednocześnie bez poprawek.

Pracuję nad stworzeniem uniwersalnego interfejsu, który ma ujednoczyć tworzenie kodu działającego pod oboma silnikami jednocześnie. Będą to specjalnie utworzone funkcje oraz generatory kodów w postaci edytorów bloków nieopartych na kodzie.

a) JavaScript

Domyślny interpreter skryptów w Javie. Prosty w obsłudze, lecz wykonywanie funkcji może sprawiać problemy (problem nadpisywania wartości), przez co wszystkie zmienne w funkcjach powinno się deklarować.

Wybierz go, jeśli jesteś początkującym i nie zależy ci na razie na szybkości obliczeń

b) Python (Jython)

Dodatkowy interpreter skryptów, opcjonalny. Uczy większej dbałości o kod, oraz jest znacznie szybszy. Wykonywane funkcje nie nadpisują wartości zmiennych. Jest też zdecydowanie szybszy od JavaScriptu.

Jest on opcjonalny, więc żeby go używać, należy ściągnąć *JavaBlock* w wersji z Jythonem. Aktualnie nie działa w aplecie (z racji jego rozmiaru).

Wybierz go, jeśli zależy ci na szybkości wykonywania skomplikowanych algorytmów.

JavaBlock w celu ujednoczenia interfejsu i zgodności między tymi silnikami wprowadza wiele ułatwień takich jak inkrementacja, funkcje pośredniczące, synonimy funkcji, których normalnie w danym języku nie ma.

Python nie posiada żadnych klamer. „Przynależność” jest opisywana wcięciami. Jeśli chcesz podzielić swoje działanie na kilka linii, możesz to zrobić dodając 2 spacje na koniec linii.

```
d=b*b#koniec linii
+4*a*c #niepoprawny zapis, zakończy się błędem
```

² Żadnej ustawy normalizującej schematy blokowe nie ma. Po prostu takie kształty i funkcje się przyjęły i tak zostało. Łamiąc ograniczenia tych „Standardów” *JavaBlock* zyskuje nowe możliwości.

```
d=b*b #koniec linii  
+4*a*c #poprawny zapis
```

3.5. Tryby edycji

JavaBlock oferuje dwa tryby tworzenia schematów blokowych. Aby zmienić tryb, przed stworzeniem bloku należy włączyć, lub wyłączyć opcję **Bazowane na kodzie** w menu kontekstowym (lub na lewym panelu, jeśli włączony). W widoku standardowym na lewym pasku narzędziowym bloki wejścia i wyjścia są traktowane osobno jako bloki niekodowe.

a) Kodowe (oparte na kodzie)

Jest to tryb w którym użytkownik bezpośrednio pisze kod. Może to być nieco kłopotliwe, gdy nie zna się składni wejścia/wyjścia.

b) Niekodowe (nie oparte na kodzie)

Nie wymaga znajomości interfejsu wejścia/wyjścia, gdyż polecenie pobierające, lub wczytujące jest generowane przez program. Użytkownik musi tylko podać nazwę zmiennej, wybrać jej typ i ew. wpisać stosowny komunikat.

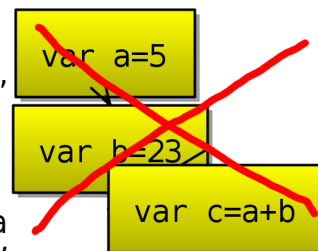
Ułatwia również obsługę podstawowych dodatków `canvas2d` i logo. (przewidywane)

Jeżeli blok jest w trybie edycji niekodowej, zamiast zwykłego edytora kodu i komentarza wyświetla się edytor przeznaczony dla danego typu bloku.

Kod generowany przez takie bloki będzie działać na wszystkich obsługiwanych silnikach skryptowych.

3.6. Kilka zasad tworzenia czytelnych schematów

W tworzeniu schematów blokowych obowiązuje kilka zasad, zarówno logicznych, jak i estetycznych:



a) Pionowość

W programowaniu ważna jest estetyka. Podczas kodzenia (pisania kodu programu) trzeba używać wcięć, żeby się nie pogubić. W schematach blokowych powinno bloki ustawiać się w miarę pionowo. Dlaczego? Wtedy bardziej przypominają „liniowość” kodu. W celu ułatwienia tej zasady *JavaBlock* oferuje grupę bloków *Klamry*

b) Jeden blok – jedna instrukcja

Oczywiście nie dosłownie. Bloki mają ułatwić wizualizację, a nie być tylko „kontenerem” na kod. w jednym bloku staraj się umieszczać jak najmniej kodu. Lepiej wyglądają dwa mniejsze bloki, niż jeden długi.

c) Schemat w każdym przypadku kończy się terminatorem

Terminator (blok końca, „End”) kończy analizowanie schematu. Nie powinno być w schemacie bloku, który nie jest połączony do żadnego dalszego bloku.

d) Kolory

JavaBlock pozwala na kolorowanie bloków. Należy jednak pamiętać, żeby treść bloków była czytelna. Kolory zostały wprowadzone w celu ułatwienia grupowania bloków. Nie przesadzaj jednak zbyt z kolorowaniem.

e) Nazywaj zmienne w miarę logicznie i prawidłowo

Podstawa programowania. Nie nazywaj zmiennych wg alfabetu (styl „na matematyka”): *a*, *b*, *c*, *d*, *e* itd., bo się łatwo można pogubić. Ogólnie jest przyjęte, że zmienne *i*, *j*, *k*... to tzw. zmienne iteracyjne (używane w pętlach do np. odliczania), ale dla pozostałych wartości przypisz sensowne nazwy zmiennych (np. *dlugosc*).

Nazwy zmiennych nie powinny zawierać polskich znaków, spacji, ani żadnych innych specjalnych znaków. Tylko litery a-z, A-Z, oraz cyfry 0-9 i podkreślenie *_*, przy czym nazwa nie może zaczynać się od cyfry, gdyż może wtedy być uznana za liczbę, za którą stoi jakiś nierozumiany ciąg znaków, co zostanie uznane za błąd i algorytm zakończy się błędem.

UWAGA! Trzeba pamiętać, że nie wszystkie nazwy są dozwolone. Przykładowe niedozwolone nazwy:

```
function, for, while, if, true, false, do, new, Array, Out, foreach,
var, in, is, outputStream, InputReader, JOptionPane, addons,
*_block, from, import
```

f) Zachowaj przerwy między blokami

Nie nakładaj bloków na siebie. Są wtedy nieczytelne i łatwo się pomylić przy łączeniu. Między blokiem decyzyjnym, a połączonymi z nim blokami również zostaw miejsce na wartość Prawda/Fałsz.

g) Nie rób strzałek pod kątem

Utrzymuj strzałki zawsze pionowo, lub poziomo. Wyglądają wtedy estetyczniej.

Użyj bloku Przeskok, by zbudować linie łamane.

h) Strzałki nie mogą się krzyżować

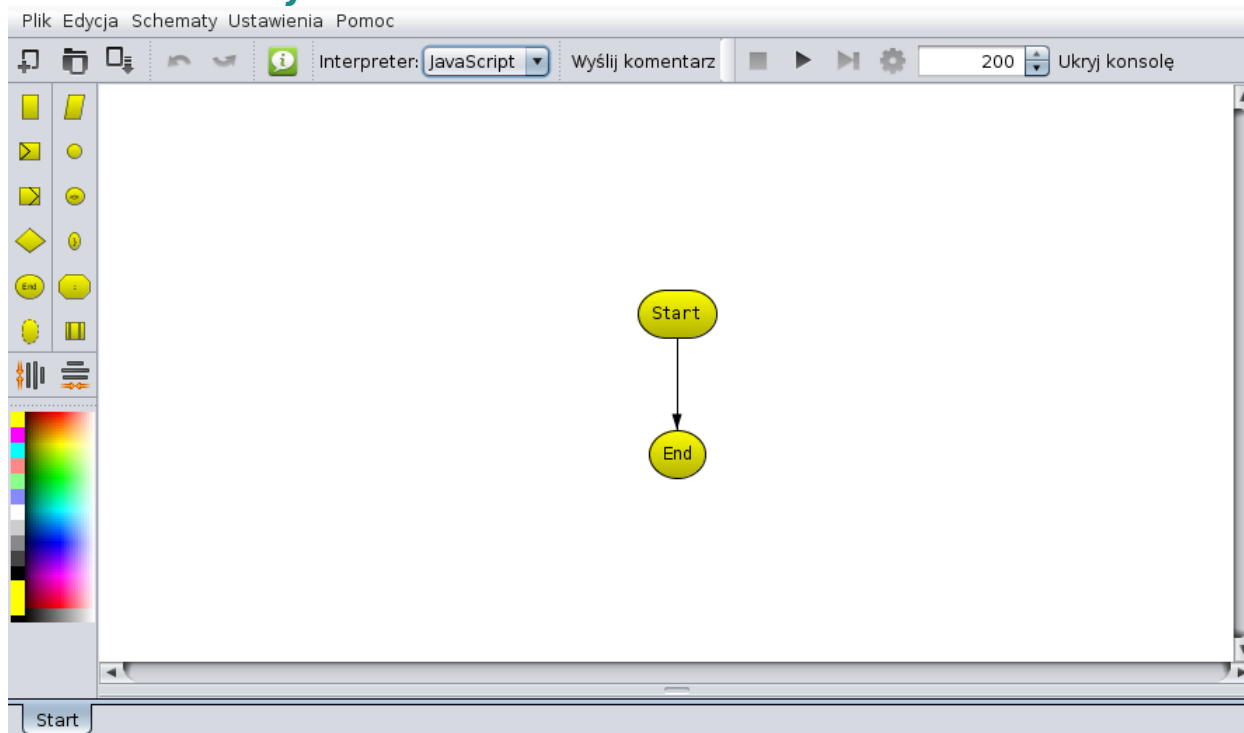
Unikaj sytuacji, w której dwie strzałki się krzyżują. Gdy sytuacja tego wymaga, użyj łącznika jako mostu nad istniejącą już strzałką.

i) Kolejność rysowania

Klawiszami Page Up/Page Down zmieniasz kolejność rysowania. Szczególnie przydatne przy grupowaniu, gdzie się tworzy podgrupy.

4. Obsługa programu

4.1. Interfejs



Domyślne okno programu

Interfejs został zoptymalizowany do pracy przy małych rozdzielczościach tak, aby obszar roboczy był jak największy

Po lewej stronie znajduje się pasek narzędziowy w którym znajdują się wszystkie podstawowe bloki, można wyrównać zaznaczone bloki oraz ustawić ich kolory. Można włączyć szeroki lewy panel w ustawieniach programu

„Konsola” jest standardowo ukryta w dolnej części nad zakładkami. Aby ją pokazać/schować, należy nacisnąć przycisk „Ukryj konsolę” w górnym pasku narzędziowym. W ustawieniach programu można wyłączyć konsolę na spodzie, będzie wtedy po prawej razem z przyciskami do kontroli symulacji.

Na górnym pasku narzędziowym znajdują się standardowe przyciski: *Nowy*, *Otwórz*, *Zapisz*. Poza tym, przycisk *info* do wyświetlenia informacji o wersji programu, wybór interpretera kodu, przycisk do wysyłania komentarzy/ uwag/ sugestii, oraz kontrola symulacji (jeśli włączona konsola u dołu).

4.2. Poruszanie się po przestrzeni roboczej

Aby przesunąć widok, należy nacisnąć i przytrzymać środkowy lub prawy przycisk myszy i przesuwać mysz.

Aby przybliżyć/oddalić widok poruszaj rolką. Jeżeli nie masz rolki, użyj klawiszy +/-.

4.3. Zaznaczanie

Bloki zaznacza się lewym przyciskiem myszy. Aby zaznaczyć więcej, należy trzymać lewy przycisk i przeciągać, lub zaznaczać/odznaczać kolejne trzymając klawisz shift.

4.4. Tworzenie bloków

Aby dodać blok do schematu należy wybrać z lewego panelu blok, który chcemy dodać, albo z menu kontekstowego, które otwieramy klikając prawym przyciskiem myszy w pustej przestrzeni schematu.

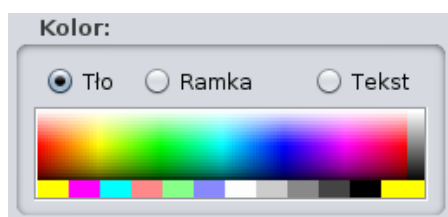
4.5. Łączenie bloków i usuwanie połączeń

Aby połączyć bloki należy zaznaczyć blok, z którego strzałka ma wychodzić, a następnie trzymając klawisz **CTRL** nacisnąć blok do którego strzałka ma być skierowana. w przypadku zwykłych bloków poprzednie połączenie wyjściowe zostanie usunięte, jeśli istniało.

W przypadku bloku decyzyjnego, muszą być dwa wyjściowe połączenia. Tworzy je się tak samo: trzymając CTRL. Zostanie usunięte najstarsze stworzone połączenie. Aby zamienić połączenia wartościami (*Prawda* na *Falsz* i na odwrót), zaznacz blok decyzyjny i naciśnij klawisz *r* (*reverse*).

Aby usunąć połączenia, zaznacz blok/i i z menu kontekstowego wybierz opcję usunięcia połączeń wejściowych, wyjściowych lub wszystkich.

4.6. Kolorowanie



Paleta kolorów przy włączonym szerokim lewym panelu

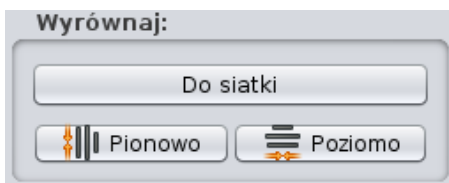
Na lewym pasku narzędziowym znajduje się paleta kolorów. Zaznacz blok którego kolor chcesz zmienić i wymierz z palety kolor. Trzymając Shift zmienisz kolor czcionki, a CTRL kolor ramki.

Aby pokolorować bloki, zaznacz je i naciśnij na kolor w panelu edycji bloku na palecie kolorów. Nad paletą wybierz wcześniej kolor czego chcesz zmienić. Zmiana koloru tła nie mając zaznaczonego bloku

zmieni kolor tła widoku schematu.

Klikając prawym przyciskiem, otworzy się bardziej zaawansowana paleta.

4.7. Wyrównywanie



Panel wyrównywania bloków przy włączonym szerokim lewym panelu

Standardowo program ma włączone wyrównywanie do siatki, dzięki czemu łatwo wyrównywać bloki w linii. Jednak czasem się zdarza, że trzeba wyrównać kilka. Na lewym pasku narzędziowym znajdują się przyciski do wyrównywania.

Możesz też użyć klawisza Home (w pionie), lub End (w poziomie).

4.8. Grupowanie

JavaBlock pozwala grupować bloki tzn. rysować pod grupą bloków prostokąt. Pozwala to wydzielać sekcje np. pętlę.

Aby zgrupować bloki, zaznacz co najmniej dwa, kliknij na jeden z zaznaczonych prawym przyciskiem i wybierz „Grupuj”. Powstały blok można zaznaczyć klikając na niego dwukrotnie. Aby dodać, lub usunąć bloki z tej grupy, należy zaznaczyć tą grupę i trzymając CTRL naciskać na bloki, które mają być dodane lub usunięte. Kolor grupy można zmienić tak samo jak kolor bloku.

5. Składnia, czyli tworzymy schemat

Zanim zaczniemy robić pierwszy schemat, trzeba się nauczyć podstaw programowania, a więc tworzenia zmiennych, operacji na nich, pobieranie i wypisywanie danych.

Program ujednocila składnię między obsługiwanymi silnikami skryptowymi, lecz nie zawsze są one ze sobą zgodne.

5.1. Tworzenie zmiennych

W *JavaScript* nie trzeba deklarować zmiennych, ani ich typów. Jest to tzw. *Dynamiczne typowanie*. Typ zmiennej jest automatycznie przypisywany w zależności od jej treści:

```
var liczba=25  
var tekst="Tekst"
```

Pogrubiłem słowa kluczowe **var**. Nie są one wymagane, ale warto je stosować przy pierwszym przypisaniu wartości. Program przy uruchomieniu symulatora wyszukuje wszystkie zadeklarowane w ten sposób zmienne i dodaje je do śledzonych wartości zmiennych.

Słowo **var** jest wymagane przy tworzeniu funkcji używając silnika JavaScript (domyślnie). Powoduje ono, że jest tworzona nowa zmienna, a nie korzysta z tej „wyżej”, tzn. schematu który wywołuje inny schemat jako funkcję. Dlatego warto na początku zadeklarować sobie wszystkie używane zmienne.

W silniku Python słowo kluczowe `var` jest usuwane, ale nadal pełni funkcję deklaracji zmiennej w celu dodania jej do śledzonych. Trzeba jednak pamiętać, żeby podczas używania słowa `var` w Pythonie od razu przypisać do zmiennej jakąś wartość.

5.2. Przypisywanie wartości zmiennych

Przypisywanie wygląda w sumie podobnie do „deklaracji”. Tu jednak już nie piszemy słowa kluczowego **var**.

```
liczba=6  
tekst="Inny tekst"
```

5.3. Operatory matematyczne

W JavaScript i Python dostępne są operatory matematyczne: + (suma), - (różnica), * (iloczyn), /(iloraz), % (reszta z dzielenia). Uwzględniana jest kolejność wykonywania działań, oraz nawiasy:

```
liczba1 = 2+2*2 //liczba1=6  
liczba2 = (2+2)*2 //liczba2=8  
reszta = 121%3 //reszta=1
```

5.4. Dzielenie całkowite

a) JavaScript

Problemem w JavaScript jest dynamiczne typowanie, które uniemożliwia w prosty sposób podzielić liczbę całkowitą tak, aby pozostała liczbą całkowitą. Można użyć operatora dzielenia całkowitego `div` nawet bez trybu pascala.

Trzeba jednak pamiętać, że zarówno tryb pascala jak i inne „ułatwiacze” mogą nie działać w niektórych warunkach, gdyż opierają się na zamianie kodu. Przykładowo `div` w wyrażeniu: `a div sqrt(3)` nie zadziała (pracuję nad tym). Aby być pewnym bezpieczeństwa, należy użyć funkcji `floor(x)`:

```
d=floor(5/2.5)
```

b) Python

Mimo dynamicznego typowania w Pythonie, zmienne utrzymują swoje typy, przez co dzielenie liczb całkowitych da liczbę całkowitą, czyli bez ułamka. Jeżeli natomiast jedna z nich jest liczbą rzeczywistą, dzielenie nie jest już całkowite.

Zamiast floor, można użyć funkcji rzutującej int rzucającej argument do liczby całkowitej. Zatem:

```
5/2 = 2 #obie są całkowite
5.0/2 = 2.5 #jedna z nich jest rzeczywistą mimo braku części
ułamkowej
int(5.0)/int(2.0) = 2 #obie są zrzucone do liczb całkowitych
int(5.0/2.0) = 2 #sam wynik jest zrzucony do liczby całkowitej
floor(5.0/2.0) = 2.0 #Wynikiem jest liczba rzeczywista, tylko część
ułamkowa została usunięta
```

5.5. Funkcje matematyczne

Dostępne są wszystkie funkcje matematyczne z JavaScript i Python. Skrócone (tzn. nie trzeba pisać *Math.*) zostały te najważniejsze:

Zwraca	Sygnatura	Opis
Liczba	sqr (liczba)	Zwraca kwadrat podanej liczby
Liczba	sqrt (liczba)	Zwraca pierwiastek kwadratowy podanej liczby
Liczba	pow (k, n)	Zwraca <i>n</i> -tą potęgę liczby <i>k</i>
Liczba	toRadians (stopnie)	Zwraca kąt w radianach z podanych stopni
Liczba	toDegrees (radiany)	Zwraca kąt w stopniach z podanych radianów
Liczba	sin (kąt) cos (kąt) tg (kąt)	Funkcje trygonometryczne. UWAGA!! Podaje się kąty w radianach! Trzeba więc najpierw rzucić je do do radianów funkcją toRadians
Liczba	rand (a, b)	Zwraca losową liczbę całkowitą od <i>a</i> do <i>b</i> .
Liczba	floor (a)	Zwraca liczbę bez części ułamkowej (zaokrąglenie w dół)
Liczba	ceil (a)	Zwraca liczbę zaokrągloną w górę (ceil(2.1)=3.0)
Liczba	round (a, p)	Zwraca zaokrągloną liczbę z precyzją <i>p</i> (0.1 oznacza jedno miejsce po przecinku)

Przykład wykorzystania:

```
var a=4
var b=sqr(a) //b=16
var c=sin(toRadians(90)) //c=1
var ran=rand(1, 10) //liczba losowa od 1 do 10 włącznie
```

5.6. Tryb pascala

Jest to tryb w którym kod pascalopodobny zamieniany jest na kod javascriptowy. Pozwala to pisać w stylu pascala, tzn np.:

```
a:=5 mod 2
```

Natomiast znak „=” jest porównaniem:

```
a=5
```

Sprzeczność to <>, zamiast !=

```
a<>5
```

5.7. Operatory przypisania

Dozwolone są także inne operatory przypisania oprócz zwykłego (=). Są to:

- += dodaje do zmiennej to co po prawej
- -= odejmuje od zmiennej to co po prawej
- *= mnoży przez to, co po prawej
- /= dzieli przez to, co po prawej
- %= reszta z dzielenia tego co po prawej

Są one równoznaczne z:

```
a=a+5
a+=5

b=a
b-=a

c*=a+b
c=c*(a+b)
```

I analogicznie reszta. Zwróć uwagę na pogrubiony fragment. - najpierw wykonywane jest całe działanie za operatorem przypisania, a potem dopiero jest przypisanie z daną operacją.

5.8. Tworzenie i działania na tablicach

a) JavaScript

Aby stworzyć tablicę, można wykorzystać jeden ze sposobów:

```
var tab=new Array(10) //10-elementowa tablica
var tab2=new Array(1, 2, 3) //3-elementowa tablica z podanymi
wartościami
```

Nic nie stoi na przeszkodzie aby stworzyć tablicę o większym rozmiarze (zarezerwować więcej miejsca).

Pojedyncza komórka tablicy jest traktowana jak normalna zmienna. Numer komórki to tzw. indeks, podaje się w nawiasach kwadratowych [], :

```
tab[0]=1
tab[1]=2
```

Najczęściej jednak tablice używane są w pętlach, o tym później.

Wielkość tablicy pobieramy polem **length**:

```
tab=new Array(1,2,3)
l=tab.length //bez nawiasów na końcu!
```

b) Python

W pythonie używa się list. Są one bardziej zaawansowane od zwykłych tablic. Skrypt inicjujący udostępnia funkcję **newArray(n)** („kompatybilna” z JavaScript lub samo **Array(n)**), która tworzy n-elementową tablicę. Jest zatem kilka możliwości tworzenia tablic:

```
tab=newArray(10) # tworzy tablicę (listę) 10-elementową
tab=[1,2,3] # tworzy tablicę (listę) 3-elementową z danymi
wartościami
```

Do komórek odwołujemy się dokładnie tak samo jak w JavaScript:

```
tab=newArray(5)
```

```
tab[0]=1
tab[1]=1
tab[2]=2
```

Listy oferują jeszcze kilka przydatnych metod takich jak **extend** i **append**. **Append** dodaje na koniec tablicy element podany w argumencie (nawet jeśli to tablica), natomiast **extend** dołącza do tablicy komórki podanej w argumencie tablicy:

```
tab=[1,2,3]
tab.append(4) # tab==[1,2,3,4]
tab.extend([5,6]) # tab==[1,2,3,4,5,6]
tab.append([7,8]) # tab==[1,2,3,4,5,6,[7,8]]
```

Wielkość tablicy pobieramy funkcją **len(tab)**:

```
tab=Array(5)
l=len(tab)
```

5.9. Wejście/wyjście

Jak wcześniej było wspomniane, schematy można tworzyć nawet nie znając specjalnie składni dzięki blokom nie kodowym. po odznaczeniu przełącznika można tworzyć wczytywać i wypisywać dane nie znając składni korzystając z prostego edytora.

a) Edytor nieoparty na kodzie

Jest to edytor służący do wczytywania i wyświetlania danych bez konieczności znania składni.

Edytor nie posiada kontroli błędów, więc wszelkie błędy mogą wyniknąć dopiero przy wykonywaniu.

Edytor bloku nieopartego o kod

- **Wczytaj**

- Tablica: Pozwala na wczytanie całej tablicy. W ramach bezpieczeństwa maksymalny limit to 20. Ustawienie ilości elementów na 0 jest równoznaczne z wyłączeniem tablicy
- **Typ:** liczba rzeczywista, całkowita, tekst, tablica znaków (nie działa w JavaScript), logiczna (*Prawda/Falsz*). Wpisanie przez użytkownika błędnego formatu spowoduje ponowne wyświetlenie formularza wejścia
- **Wypisz** w przypadku wypisania nie jest konieczne wpisanie nazwy zmiennej. Można wpisać tekst (w cudzysłowie), liczbę, lub całe poprawne wyrażenie.
 - **Sufix wiadomości** czyli dalsza część wiadomości już po wypisaniu wartości zmiennej (np. „, ” przy wypisywaniu tablicy).
 - **Wstaw nową linię** wstawia nową linię po wypisaniu.

W przypadku wyjścia co najmniej jedno z tych 3 pól musi być wypełnione. Blok taki nie musi wypisywać żadnych wartości zmiennych.

Warto stosować ten rodzaj bloku, gdyż jest on kompatybilny z silnikiem skryptów Python (o ile nie kombinujemy w polu „Wiadomość”)

b) „Komendy schematowe”

Drugim sposobem jest wejście/wyjście oparte na kodzie gdyż w ten sposób można samemu sformułować całe wyrażenie za pomocą tzw. Stringów. O nich później. Poza standardowymi funkcjami *JavaBlock* oferuje również tzw. „komendy schematowe”. Nie da się przy nich wyświetlać komentarzy, ale ładniej wyglądają na schemacie.

c) Funkcje

tekst *Read(wiadomość)*

Wczytuje i zwraca tekst. *wiadomość* to wiadomość wyświetlana w okienku (może być pozostawiona pusta)

```
a=Read("Wpisz tekst")
read a
```

liczba *ReadNumber(wiadomość)*

Wczytuje i zwraca podaną liczbę.

```
a=ReadNumber("Podaj liczbę")
readNumber a
```

Write(wiadomosc)

Wypisuje tekst do pola „Wyjście”. *wiadomosc* to tekst, jaki ma wyświetlać.

```
a=5
Write(a)
write a
```

Print(wiadomość) jest synonimem³

Writeln(wiadomosc)

jw., tylko przechodzi do następnej linii

```
a=4
Writeln(a)
writeln a
```

Println(wiadomosc) jest synonimem

5.10. Warunki

Warunki zawsze są albo spełnione (wartość *true*), albo nie spełnione (wartość *false*). Najczęściej sprawdza się równość dwóch wartości tzw. operatorami porównania.

a) Operatory porównania

Wartości porównuje się jak w matematyce z taką różnicą że znak „=” nie oznacza „jest równy”, tylko jest przypisaniem. Aby porównać dwie zmienne trzeba użyć „==” (dwa znaki „równa się”). Mała tabelka:

Znaki	Jak czytać	Kiedy <i>true</i>
== (=)	„jest równy”	wartości są równe (w nawiasie podany znak w trybie Pascala)
!= (<>)	„nie jest równy”	wartości nie są równe (w nawiasie podany znak w trybie Pascala)

³ Synonim – w programowaniu funkcja, która znaczy dokładnie to samo, lub po prostu wykonuje to samo co funkcja, której jest synonimem.

Znaki	Jak czytać	Kiedy <i>true</i>
<	„mniejszy od”	wartość po lewej jest mniejsza
>	„większy od”	wartość po lewej jest większa
<=	„mniejszy lub równy”	wartość po lewej jest mniejsza lub równa prawej
>=	„większy lub równy”	wartość po lewej jest większa lub równa prawej

UWAGA!! W JavaScript porównywać można zawsze tylko dwie wartości! Jeżeli chce się porównać 3 wartości, np. $1 < a < b$, trzeba sprawdzić oba warunki osobno i połączyć je odpowiednim operatorem logicznym. Wyrażenie $1 < a < b$ jest traktowane podobnie jak $(1 < a) < b$, a $(1 < a)$ zwraca wartość *true/false*.

Podwójne porównania są dostępne dopiero w silniku Python.

b) Operatory logiczne

Warunki można ze sobą łączyć tzw. operatorami bitowymi, logicznymi. Wyróżniamy 4 podstawowe operatory logiczne:

Znak	Angielski (Pascal)	Polski	Nazwa fachowa	Kiedy sumarycznie <i>true</i>
&&	AND	I	Iloczyn bitowy (Koniunkcja)	wszystkie porównania są <i>true</i>
	OR	Lub	Suma bitowa (Alternatywa)	co najmniej jedno jest <i>true</i>
^^	XOR	„Albo”	Alternatywa wykluczająca	kiedy <u>TYLKO</u> jedno jest <i>true</i>
!	NOT	Nie	Negacja bitowa	jest <i>false</i>

Wracając do przykładu z poprzedniego podpunktu, warunek ten będzie wyglądał tak:

```
1 < a && a < b
```

W przypadku silnika Python używamy małych słów angielskich. Brak XOR. Program zamienia automatycznie formę znakową i angielską z dużymi literami na angielskie z małymi literami.

6. Proste przykłady schematów

6.1. Zatem tworzymy. Przykład bez rozgałęzień

Pierwszym programem jaki się zazwyczaj tworzy jest program „Hello World”⁴, ale uważam to za zbyt trywialny przykład – ograniczałby się do bloku „Start” > Wejście/Wyjście > „End”. Może na początek małe ćwiczenie bez bloków decyzyjnych: prosty kalkulator.

a) Bloki niekodowe

Tworzymy blok wejścia z lewego panelu (lub Wejścia/Wyjścia z menu kontekstowego, lub szerokiego lewego panelu mając odznaczone „Bazowane na kodzie” i wybieramy „Wejście”). W polu „Zmienna” wpisujemy „a, b”, a we „Wiadomość”: „Podaj liczbę:”. Oznacza to że wczytujemy wartości do zmiennych a , a potem b . Można to również zrobić osobno w dwóch blokach, ale tak jest mniej bloków. Zaznaczamy w „Typ”: „Liczba” i upewniamy się że „Tablica” jest wyłączona.

```
a, b
Podaj liczbę:
```

Mamy już zmienne a i b . Teraz tworzymy blok Przetwarzanie, w którym tworzymy kod obliczający sumę, różnicę, iloczyn i iloraz:

```
suma=a+b
roznica=a-b
iloczyn=a*b
iloraz=a/b
```

Nie trzeba pisać var, gdy nie chcemy śledzić ich wartości. Teraz musimy tylko wypisać wszystkie dane. Tworzymy kolejny blok Wejścia/Wyjścia i tym razem zaznaczamy „Wyjście”. W polu „Zmienna” wpisujemy kolejno: „suma, roznica, iloczyn, iloraz”. Zaznaczamy „Wstaw nową linię”.

I łączymy kolejno bloki od Start przez pierwszy blok Wejścia/Wyjścia, Przetwarzania, drugi Wejścia/Wyjścia i Końca.

b) Bloki kodowe

Najlepiej w tym samym otwartym pliku utwórz nowy schemat w nowej karcie („Schematy”>„Dodaj” i wpisz nową nazwę). Przede wszystkim zaznaczamy „Bazowane na kodzie” w menu kontekstowym lub/i panelu po lewej. Tworzymy blok Wejścia/Wyjścia, w którym wczytamy liczby a i b . Liczby wczytujemy funkcją *ReadNumber*:

```
a=ReadNumber("Podaj a");
b=ReadNumber("Podaj b");
```

Jak łatwo zauważyć, w jednym bloku można pobrać kilka zmiennych używając różnych wiadomości.

Mamy już liczby a i b . Teraz przygotujemy sobie zmienne do wyświetlenia:

```
suma=a+b
roznica=a-b
iloczyn=a*b
iloraz=a/b
```

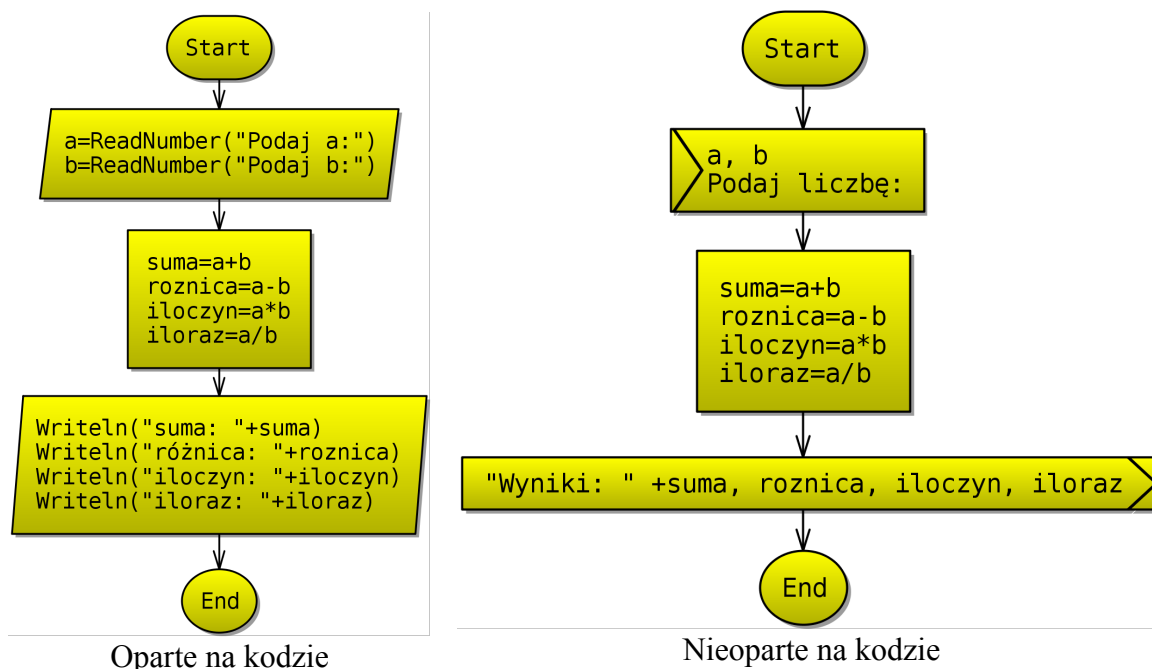
⁴ Programy typu „Hello World” mają na celu jedynie wyświetlić tenże napis powitalny. Ma one pokazać podstawową składnię języka programowania, bądź biblioteki

I wypisujemy wszystkie te zmienne:

```
Writeln("suma: "+suma)
Writeln("różnica: "+roznica)
Writeln("iloczyn: "+iloczyn)
Writeln("iloraz: "+iloraz)
```

I połączyć teraz w kolejności: „Start” > pierwszy blok Wejścia/Wyjścia > blok przetwarzania danych > drugi blok Wejścia/Wyjścia > „End”.

Schemat gotowy. Powinien wyglądać mniej więcej tak:



Jak widać używanie bloków nieopartych na kodzie jest łatwiejsze i nie wymaga znajomości nazw funkcji wczytywania i wypisywania danych, jednak ręczne stukanie kodu daje ładniejszy efekt

c) Symulacja

Teraz na prawym panelu naciśnij i . Program wyświetli okienko i poprosi o wpisanie kolejnych liczb, a następnie na prawym panelu w zakładce pojawią się wyniki.

6.2. Tworzenie stringów

W poprzednim schemacie trochę wyprzedziłem, ponieważ już łączyłem stringi. Wyłumaczę o co chodzi:

Jeżeli chcemy wyświetlić jakąś zmienną wraz z opisem, trzeba zrobić nowego stringa. Można zrobić to bezpośrednio w nawiasach w `Write/WriteLn`, lub też wcześniej i zapisać do zmiennej. Najprościej robi się to „sumując” tekst z wartością liczbową:

```
tekst="suma: "+suma;
Writeln(tekst);
Writeln("różnica: "+roznica);
```

W stringach (między cudzysłowami) możemy już używać polskich znaków. to co jest między cudzysłowami traktowane jest jako tekst i nie jest w żaden sposób interpretowane tzn. nie zamieni nazwy zmiennej na jej wartość.

Można oczywiście nie używać nawet zmiennej pomocniczej, tylko od razu

policzyć i wypisać sumę:

```
Writeln("suma: "+(a+b));
```

Jest to zapis jak najbardziej poprawny. Trzeba jednak pamiętać o nawiasach. bez nich wyrażenie $a+b$ nie będzie potraktowane jako suma dwóch liczb, tylko doda do stringa obie wartości po kolei.

Dlatego tak ważne jest podczas pobierania danych użycie *ReadNumber* zamiast *Read*. w tym drugim przypadku możemy być niemal pewni, że suma nie będzie liczbą, tylko tekstem (" $2+2=22$ ").

W przypadku bloku nieopartego na kodzie można zastosować utworzony string w polu „Zmienna”.

W przypadku Pythona, każdą zmienną trzeba rzutować do tekstu. Robi się to funkcją `str(var)`:

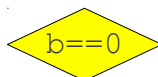
```
Writeln("suma: "+str(a+b));
```

6.3. Pierwszy blok decyzyjny

W poprzednim algorytmie jest pewna luka: Liczba b może być równa 0, wtedy nie można dzielić. Trzeba sprawdzić jej wartość i w zależności od tego, czy jest to liczba równa 0, czy nie, wykonać różne bloki.

Jeżeli wyrażenie wewnątrz bloku decyzyjnego jest prawdziwe, następnym wykonywanym blokiem będzie ten wskazywany przez strzałkę oznaczoną „true” (lub linią ciągłą). w przeciwnym razie, wykonany zostanie blok wskazywany strzałką z oznaczeniem „false” (lub linia przerywana).

Otwórz poprzedni schemat i z ostatniego bloku wejścia/wyjścia usuń ostatnią linijkę i z bloku przetwarzania danych tak samo. Musimy teraz zrobić warunek sprawdzający, czy b jest równe zeru:



I dwa bloki wejścia/wyjścia informujące o ilorazie i niemożliwości podzielenia:

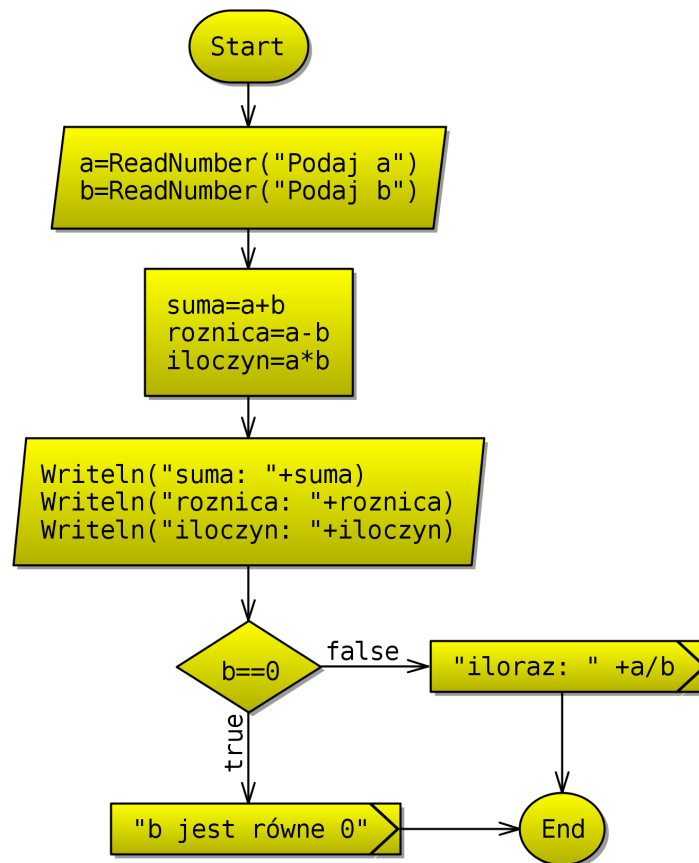


dla "true"



dla "false"

I poprowadź z bloku decyzyjnego dwie wychodzące strzałki do tych bloków, a te bloki połącz do bloku „End”. Powinno to wyglądać mniej więcej w ten sposób:



Bez tego zabezpieczenia wypisze: „iloraz: *Infinity*”. Dzielenie przez zero daje teoretycznie liczbę nieskończoną (im bliżej 0, tym większy wynik dzielenia, ale 0 jest asymptotą).

6.4. Miejsca zerowe funkcji kwadratowej

Przykład z dwoma warunkami: miejsca zerowe funkcji kwadratowej. Funkcja kwadratowa ma postać: $y = ax^2 + bx + c$. Aby obliczyć miejsca zerowe należy najpierw obliczyć Deltę:

$$d = b^2 - 4ac$$

Jeżeli $d > 0$, funkcja ma 2 miejsca zerowe, jeśli $d = 0$, jest tylko jedno, a gdy $d < 0$, brak miejsc zerowych. Mając deltę można obliczyć miejsca zerowe:

$$x_1 = \frac{-b + \sqrt{d}}{2a}$$

$$x_2 = \frac{-b - \sqrt{d}}{2a}$$

I analogicznie, jeśli delta równa zero:

$$x = \frac{-b}{2a}$$

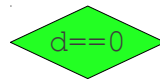
Mając te wzory można łatwo napisać program, który wypisze te miejsca. Zacząć trzeba najpierw od pobrania a , b , oraz c . Następnie obliczyć deltę:

$$d = b * b - 4 * a * c;$$

I sprawdzić pierwszy warunek: czy d jest mniejsze od zera:



Jeśli tak, wypisuje że brak miejsc zerowych i kończy, Jeśli jednak nie, sprawdza, czy jest równy zero:



Jeśli tak, oblicza x, wypisuje i kończy:

```
var x = (-b) / (2*a);
```

```
Writeln("x="+x);
```

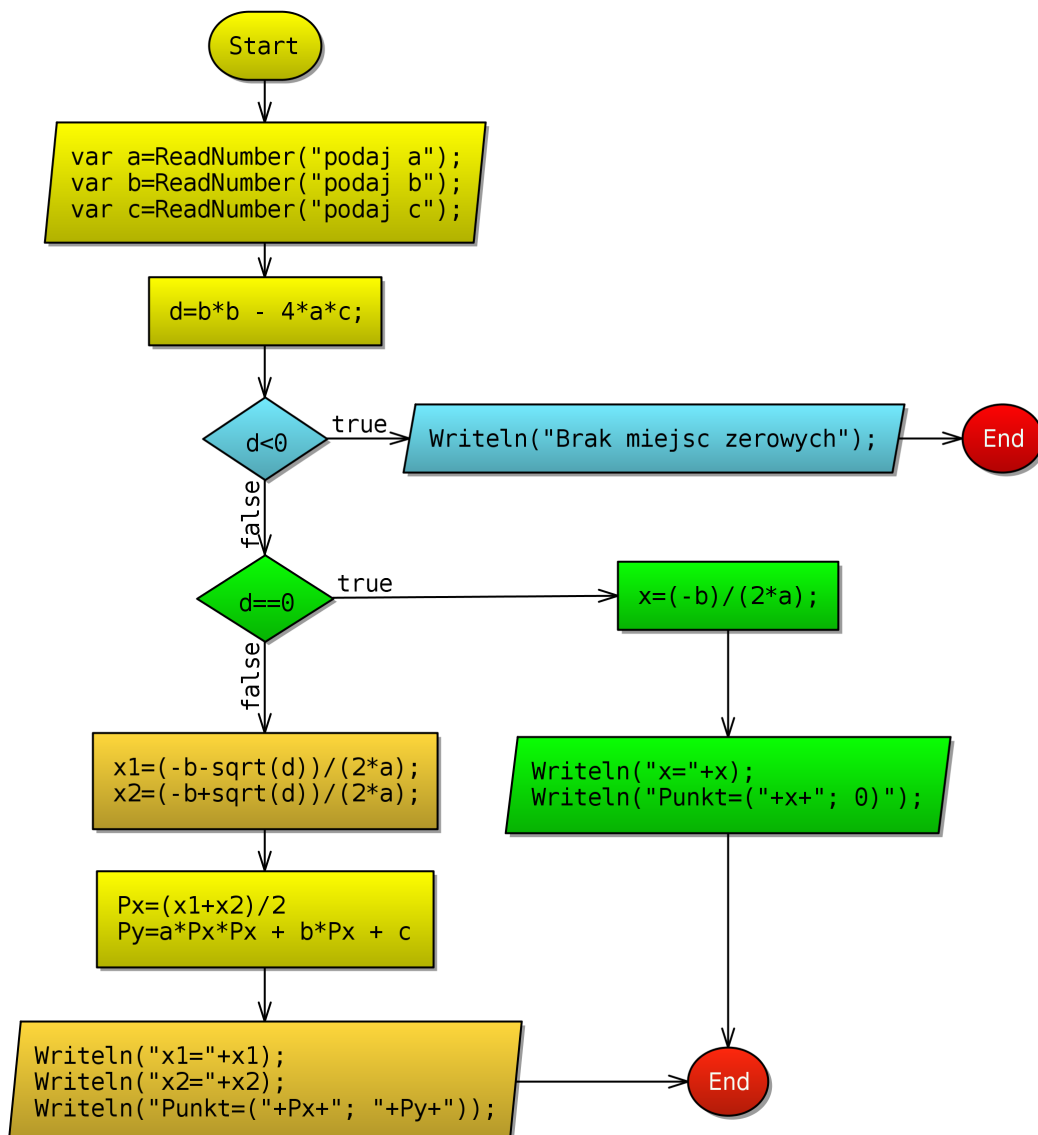
A jeśli nie, pozostaje jedyna opcja, której nie trzeba nawet sprawdzać: d jest większe od zera:

```
var x1 = (-b - sqrt(d)) / (2*a);
var x2 = (-b + sqrt(d)) / (2*a);
```

```
Writeln("x1="+x1);
```

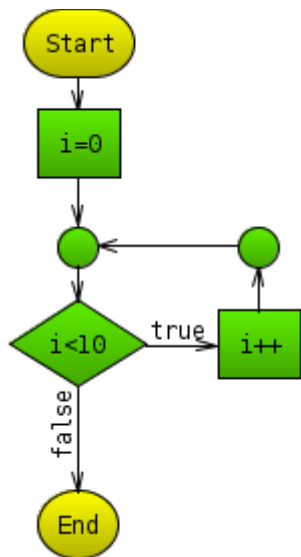
```
Writeln("x2="+x2);
```

I oczywiście bloki wypisujące wynik prowadzą do „End”. Wynik:



Jak widać są dwa bloki „End”. Kolory zastosowałem by był czytelniejszy: niebieski to przypadek gdy brak miejsc zerowych, zielony gdy jedno, a pomarańczowy gdy dwa. Czerwone zaś są terminatory.

6.5. Pętle



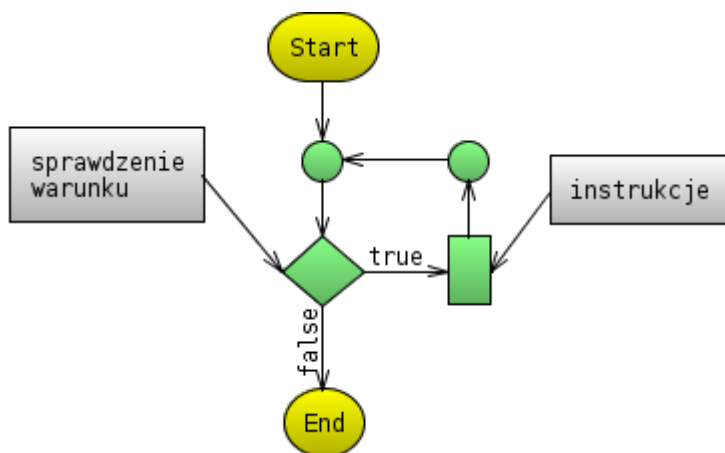
pętla for

Warunki to nie wszystko. w programowaniu często trzeba stosować pętle tzn. kilkakrotne wykonanie pewnego fragmentu algorytmu. Schematy blokowe ładnie obrazują działanie pętli. Pętle polegają na tym, że po warunku gałąź prowadzi najczęściej przed ten warunek. w schematach blokowych trudno mówić o rodzajach pętli, ale warto już znać te pętle występujące normalnie w programowaniu. Standardowa pętla wygląda następująco:

Jak widać, jeśli warunek ($i < 10$) będzie spełniony, zwiększy wartość i o jeden i ponownie sprawdzi warunek. Pętla będzie się tak długo powtarzać aż zmienna i osiągnie wartość 10.

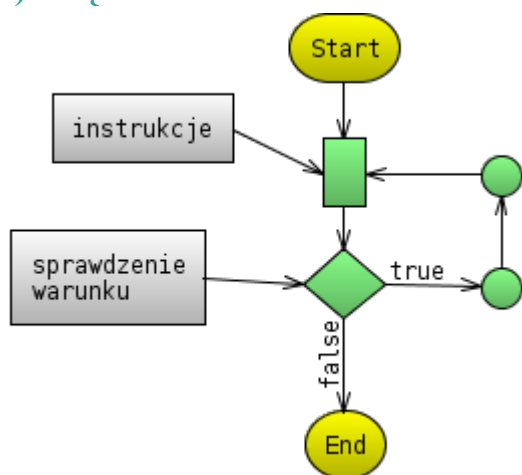
a) Pętla WHILE

Pętla WHILE to podstawowa pętla w programowaniu. Warunek jest sprawdzany PRZED instrukcjami zawartymi w pętli. jest to najprostsza pętla.



while

b) Pętla DO WHILE



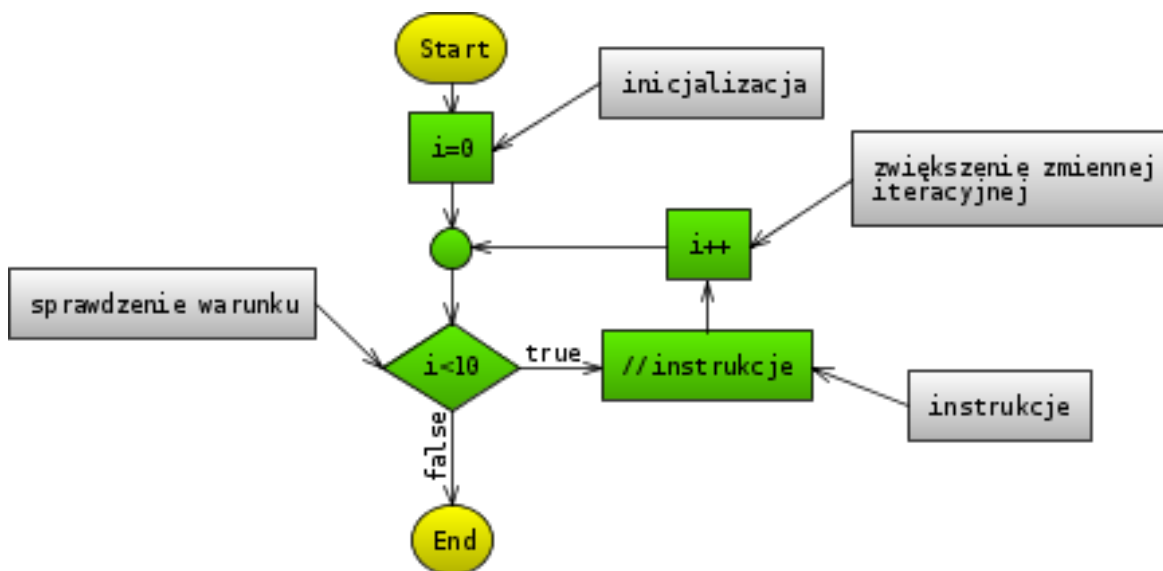
do while

Zwana też pętlą REPEAT UNTIL (Pascal). Różni się od pętli WHILE tym, że warunek jest sprawdzany na końcu, więc pętla zostanie wykonana co najmniej raz:

c) Pętla FOR

Pętla FOR opiera się na pętli WHILE. Używa się tu jednak tzw. zmiennej iteracyjnej. Składa się z czterech kroków:

1. inicjalizacja zmiennej iteracyjnej (przypisanie wartości początkowej)
2. sprawdzenie warunku
3. wykonanie instrukcji
4. wykonanie operacji (najczęściej inkrementacji zmiennej iteracyjnej użytej w warunku)



for

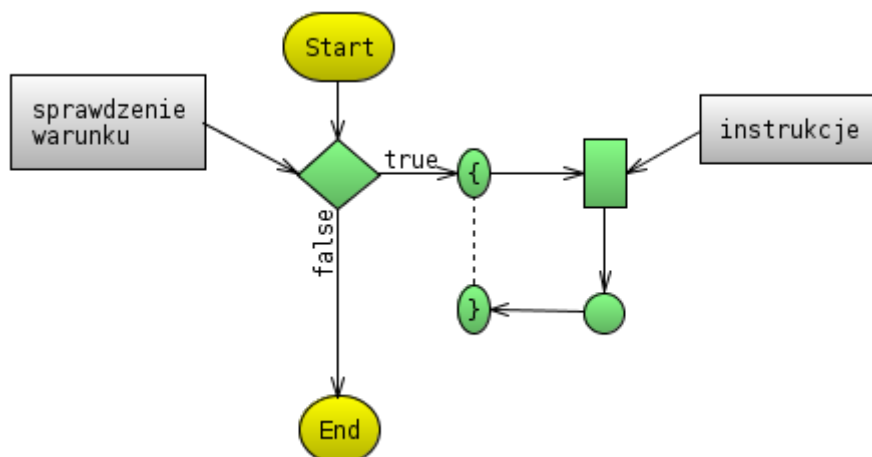
6.6. Klamra - jak działa



Klamra jest blokiem (grupą bloków) nienależącą do standardu schematów blokowych (o ile można tu mówić o jakiś „standardach”), lecz moim zdaniem ułatwia przesiadkę ze schematów blokowych na kodzenie.

Blok otwarcia działa podobnie jak Przeskok. jest normalnym blokiem, który wskazuje na następny.

Blok zamknięcia może również działać jak przeskok, jeśli połączy się go z innym blokiem. Jeśli jednak nie będzie miał połączeń wychodzących, wskazywać będzie na blok łączący się z klamrą otwierającą. Powinien to być blok decyzyjny, a klamra taka elementem pętli. Przykład pętli WHILE z wykorzystaniem klamer:



while z klamrami

Jak łatwo zauważyć na danym przykładzie, klamra zamykająca nie ma strzałek wychodzących. Następnym wykonanym po nim bloku będzie blok łączący się z klamrą otwierającą, a więc w tym przypadku z blokiem decyzyjnym.

7. Symulacja schematu i szukanie błędów






7.1. Symulacja

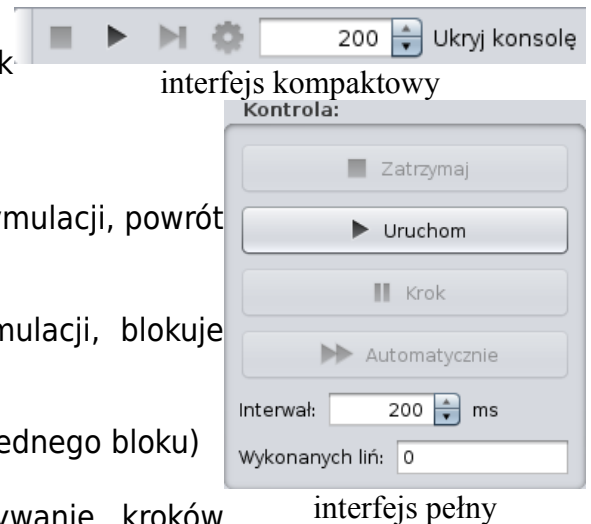
JavaBlock do symulacji używa silników skryptów (interpretatorów) JavaScript lub Python. Symulacja polega na wykonaniu kodu poszczególnych bloków. Podczas symulacji nie jest możliwa edycja bloków.

a) Sterowanie symulacją

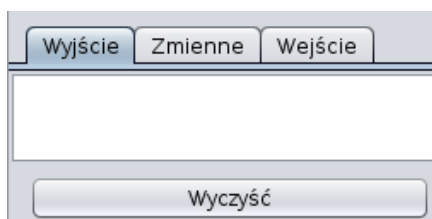
Do sterowania symulacją służy pasek narzędzi, lub panel:

Kolejne elementy oznaczają:


-  **Zatrzymaj/reset** - zatrzymanie symulacji, powrót do trybu edycji
-  **Uruchom** - uruchamia tryb symulacji, blokuje możliwość edycji
-  **Krok** - wykonanie jednego kroku (jednego bloku)
-  **Automatycznie** - wykonywanie kroków automatycznie co określony czas
-  **Interwał między krokami** - co ile milisekund ma wykonać blok
- [Interfejs kompaktowy] **Ukryj konsolę** - ukrywa konsolę pod schematem
- [Interfejs pełny] **Wykonanych linii** - ilość wykonanych linii kodu (nie licząc predefiniowanych funkcji i wywołanych schematów jako funkcje)



b) Konsola⁵



sterowania symulacji.

Konsola służy do wypisywania danych oraz do wprowadzania predefiniowanego wejścia, oraz argumentów, a także do śledzenia wartości zmiennych. W standardowym interfejsie kompaktowym konsola znajduje się pod schematem (może być zwijana, aby rozwinąć trzeba nacisnąć na rozdzielacz ). W interfejsie pełnym konsola znajduje się pod panelem

7.2. Śledzenie wartości zmiennych

JavaBlock poza zwykłym wykonaniem kodu umożliwia również dynamiczny podgląd wartości zmiennych. Służy do tego zakładka **Zmienne** na panelu konsoli

Na dole znajduje się pole tekstowe **Śledź:** . Wpisujemy w nim nazwy zmiennych, które chcemy śledzić, oddzielając je przecinkami.

Można również wpisać całe wyrażenia, np. $a*b$. Przykładowo linia ta może wyglądać następująco:

⁵ Nie jest to dosłownie konsola, raczej po prostu wyjście (to, co jest wypisywane) i stan zmiennych. Określenie „konsola” jest tu raczej określeniem potocznym

```
a, b, a*b
```

Jeżeli opcja Szukaj zmiennych (var) jest zaznaczona, program automatycznie doda do śledzonych zmiennych wszystkie zadeklarowane zmienne (poprzedzone słowem kluczowym var).

Wartości nie są aktualizowane gdy symulacja zachodzi w trybie automatycznym o interwale 0.

7.3. Interwał

Interwał to odstęp pomiędzy wykonaniami poszczególnych bloków w trybie automatycznym. Im większy, tym więcej czasu mamy na przeanalizowanie tabelki wartości.

W przypadku dodatkowego interwału można wyłączyć automatyczne kroczenie poprzez ponowne naciśnięcie „Automatycznie”. Nie można jednak mieszać trybu automatycznego, czy też ręcznego z interwałem równym 0, czy też automatycznym z dodatnim interwałem.

Dla wartości bliskich 0 nie można liczyć na rzeczywisty tak krótki interwał. Bloki wykonują się wtedy po prostu tak szybko, jak tylko mogą.

7.4. Zakładka „Wejście”

Zakładka **Wejście** w panelu symulacji ułatwia szybkie poprawianie algorytmów, które wczytują (predefiniowane) dane. Podczas każdej kolejnej symulacji nie trzeba ręcznie wpisywać danych, lecz są one pobierane z tego pola tekstowego.

Każdą daną należy wpisać w osobnej liniжке, tak więc jeżeli program ma pobrać dwie liczby, obie trzeba umieścić w osobnych liniжkach.

Jeżeli **Predefiniowane wejście** się skończy, program zacznie pytać o dane użytkownika.

Jeżeli nie chcesz, aby wszystkie dane były wczytywane automatycznie, użyj znaku ?, np.:

```
1  
2  
3  
?  
5
```

Wtedy kiedy napotka znak ?, poprosi użytkownika o dane.

8. Bloki deklarycyjne i Struktury

JavaBlock nie jest zwykłym edytorem i symulatorem schematów blokowych. Umożliwia także tworzenie prostych programów (skryptów Python lub JavaScript) oraz używanie struktur i w przyszłości (wersja 1.0, *Wodospad*⁶) obiektowość.

Bloków tej grupy nie dodaje się bezpośrednio do przepływu jednego schematu. Są one jakby globalne.

8.1. Struktury

edytor niekodowy struktury

```
Punkt:
x :Number
y :Number
```

Przykładowa struktura z dwoma polami

Struktura to w programowaniu obiekt, który przechowuje kilka danych różnych rodzajów jednocześnie (zwanym *polami*). Ułatwia to programowanie bardziej skomplikowanych algorytmów.

a) Pola struktury

Edytor pozwala na łatwe modyfikowanie pól i ich typów. Typy pól nie są sztywne: mogą przyjmować dane dowolnego typu. Jest to informacja bardziej dla *interfejsu*.

Aby dodać pole, naciśnij przycisk „Dodaj”. Aby usunąć: przycisk „X” obok nazwy pola, lub zostaw nazwę pola puste

b) Używanie struktur

Strukturę tworzy się następująco:

```
zmienna=NazwaStruktury()
```

Do pól bezpośrednio odnosimy się stawiając kropkę za nazwą zmiennej, a za nią nazwę pola:

```
zmienna.nazwa_pola=4
```

c) Interfejs akcesorów

Interfejs zapewnia bezpieczeństwo typów danych. Jeżeli typ pola to „Całkowita”, a przypiszemy np. 3.5, to liczba ta będzie zrzutowana do liczby całkowitej. Metody interfejsu wyglądają następująco:

```
struktura.getNazwa_pola() //pobieranie wartości pola
struktura.setNazwa_pola(wartosc) //ustawianie wartości pola
```

Zauważ, że w tym przypadku pierwsza litera całej nazwy pola jest wielka, a pozostałe małe. Jeśli pole będzie się nazywało np. *POLE*, to akcesory będą miały sygnatury *setPole()* i *getPole(val)*.

⁶ *Nazwa Kodowa* – wyraża możliwości programu. Aktualna nazwa to *strumyk*. Kolejna (0.6) to *rzeka*. Wersja 0.8 oznaczona będzie nazwą *delta*, a „finalna” 1.0 – *wodospad*. Motyw płynącej wody ma związek z określeniem „Diagram przepływu”.

8.2. Skrypt osadzony

Nie trzeba pisać całego programu schematami blokowymi. Można użyć bloku Skrypt Osadzony, by wpisać fragment (funkcję, klasę) gotowego kodu. Wymagana jest znajomość danego języka skryptowego, ale dzięki temu można korzystać z dobrodziejstw tychże języków.

9. Funkcje w JavaBlock

JavaBlock od wersji 0.4.6 oferuje obsługę schematów jako funkcji. Na razie możliwość tworzenia funkcji tylko działa i jest w fazie testowej.

Wykonywane schematy jako funkcje są wykonywane w trybie natychmiastowym z przydzieloną nową pamięcią, co sprawia, że wykonują się dość długo dla dużej ilości wywołań i rekurencji. Niemożliwe jest też śledzenie tych algorytmów (trzeba uruchomić je ręcznie z podanymi argumentami).

9.1. Co to jest Funkcja

Funkcja jest to „podprogram”. Fragment programu, który wykonuje pewne instrukcje, może przyjąć argumenty i zwrócić jakąś liczbę. Przykładem funkcji jest np. `sqrt(a)`, która przyjmuje za argument (parametr) liczbę i zwraca jej pierwiastek.

Dzięki funkcjom program wygląda przejrzysiej, niż gdybyśmy mieli wszystko w jednej linii pisać. Poza tym, tą samą funkcję można wywołać z kilku miejsc w programie, przez co jest to też wygodniejsze.

Przykładowo program może składać się z głównej funkcji generującej jakąś tablicę, oraz drugiej, która ją posortuje i zwróci.

9.2. Jak wywołać schemat jako funkcję?

Na początku ustaw nazwę schematu (z menu: „Schemat” > „Zmień nazwę”) na docelową nazwę funkcji.

Argumenty dodajemy w edytorze bloku startowego podobnie jak pola w strukturze.

Jeżeli funkcja ma coś zwracać, trzeba wpisać w Kodzie bloku końca (terminatorze) wartość, jaką ma zwracać. Musi być ona zgodna z zadeklarowanym typem w bloku Start.

1. **UWAGA!!** Nie są wspierane przeładowania! Tzn. nie może być dwóch funkcji o takich samych nazwach mimo różnych sygnatur!
2. **UWAGA!!** Nazwa podana w bloku Start nie jest nazwą funkcji! Nazwą funkcji jest nazwa schematu!

Teraz schemat jest dostępny jako funkcja pod sygnaturą np.

```
Start (arg1)
```

To znaczy że jest schemat o nazwie *Start* i posiada jeden argument.

Dozwolona jest rekurencja.

9.3. Zasady używania funkcji

- Brak zmiennych globalnych, wszystko przekazujemy w argumentach.
- [JavaScript] Wszystkie zmienne w funkcji należy zadeklarować (słowem kluczowym **var**), w przeciwnym razie mogą nadpisać wartości funkcji z algorytmu, który ją wywołał (gdy takie same nazwy, np. *i*).
- Nie „nadpisuj” nazwy funkcji nazwą zmiennej.
- Brak wsparcia dla przeładowań, tzn. funkcji o takich samych nazwach

10. Konfiguracja

10.1. Ogólne

a) Rysowanie:

Rysowanie

- Prerenderowane grafiki
- Gradienty w trybie edycji
- Czcionka TlwgMono
- Krzywe Beziera

są pod ukosem.

- Prerenderowane grafiki - po zmianie treści bloku przerysowuje go od nowa, później wyświetla tylko obrazek. Przyspiesza rysowanie kosztem RAMu
- Gradienty w trybie edycji - rysuje przyciemniane bloki
- Krzywe Beziera - rysowanie krzywych połączeń, gdy

b) Interpreter skryptów

Interpreter skryptów

Silnik: Zamieniaj "schematowe" polecenia

- Pojedyncze wywołanie skryptu w trybie natychmiastowym (interwał=0)
- Podświetlaj bloki nie wykonujące kodu (przeskoki, łączniki, klamry)
- Zaznacz ostatnie zmiany w tabeli

- Silnik: domyślny silnik nowych schematów
- Zamieniaj „schematowe” polecenia - zamiana poleceń w bloku wejścia/wyjścia

- Pojedyncze wywołanie skryptu w trybie natychmiastowym - zamienia schemat na skrypt JS i wywołuje go całego naraz. Przyspiesza wykonanie, ale wymaga dużo RAMu przy dużej ilości iteracji
- Podświetlaj bloki nie wykonujące kodu - podświetla podczas symulacji bloki pomocnicze, niezawierające kodu
- Zaznacz ostatnie zmiany w tabeli - podświetla ostatnie aktualizowane pozycje w tabeli śledzenia wartości zmiennych

c) Uruchamianie

Uruchamianie

- Pokazuj obraz tytułowy
- Wczytaj ostatni schemat

- Pokazuj obraz tytułowy - wyświetla tzw. splash zanim uruchomi się programach
- Wczytaj ostatni schemat - przy uruchomieniu wczytuje ostatni edytowany schemat (nawet niezapisany)

d) Edytowanie

Edytowanie

- Twórz Przeskoki podczas próby "połączenia" z pustą przestrzenią

Efekt wyróżnienia:

- Twórz przeskoki (...) - podczas próby połączenia z pustą przestrzenią, tworzy przeskoc. Ułatwia tworzenie łamanych linii.
- Efekt wyróżnienia: efekt po najechnaniu myszką na blok (auto - w zależności od przybliżenia)

e) Interfejsu

Interfejs

- Pokaż pasek narzędzi
- Podpisy w pasku narzędziowym

Look and Feel:

- Pokaż pasek narzędzi - wyłącz, jeśli masz niską rozdzielczość pionową
- Look and Feel - motyw, zmień, jeśli długo rysuje interfejs, lub jest zbyt duży. Zalecany Nimbus,

lub systemowy (Windows, Gtk+)

10.2. Kolory

a) Domyślne kolory

Domyślne kolory:

Tło bloków:

Obramowanie bloków:

Kolor czcionki:

Tło schematu:

Przejrzyste tło dla PNG

- Domyślne ustawienia dla nowostworzonych bloków oraz ikon bloków
- Rysowanie przezroczystego tła przy eksportowaniu do PNG

b) Paleta statyczna

Paleta kolorów

Paleta statyczna:

Poziomy rozmiar piksela:

Pionowy rozmiar piksela:

- Paleta statyczna - paleta stałych kolorów pod paletą HSL na lewym panelu
- Poziomy/Pionowy rozmiar piksela - rozmiar pojedynczej próbki koloru na paletce (im większa, tym szybciej się rysuje i łatwiej jest znaleźć ponownie ten sam odcień)

10.3. Pastebin.com

e-mail:

Wygasa:

Wystaw:

Pytaj o nazwę

dokumentu

- E-mail - twój e-mail na który będą wysyłane linki
- Wygasa - czas po którym dokument zostanie usunięty z serwera
- Wystaw - dostępność na serwisie pastebin.com
- Pytaj o nazwę - przy wysyłaniu pyta o nazwę

11. Dzielenie się algorytmem

Jeśli chcesz podzielić się swoim algorytmem ze znajomym, są dwie możliwości:

- Tradycyjna - wysłać plik (na maila, przez jakiś hosting), ale nie każdemu się uśmiecha trzymać dziesiątki plików
- Za pośrednictwem usługi pastebin.com - Plik>Eksportuj do pastebin.com. Program może zapytać o nazwę schematu (jeśli zaznaczone w ustawieniach). po potwierdzeniu po chwili pokaże się okienko z trzema tekstami:
 - **URL** - bezpośredni adres HTTP do serwisu pastebin.com z wysłanym dokumentem
 - **Pastebin ID** - numer identyfikacyjny dokumentu, używany przy Plik>Importuj z pastebin.com
 - **Aplet** - adres HTTP do apletu, który automatycznie wczyta wysłany schemat (wystarczy wkleić link w przeglądarce).



Okno z adresami pastebin i apletem

e-mail:

Wygasa: 10M 10 Minutes ▼

Wystaw: Public ▼

Pytaj o nazwę

Konfiguracja wysyłania do pastebin.com

Dokument będzie można otworzyć do 10 minut, 1 dnia, 1 tygodnia, 1 miesiąca lub zawsze (bez limitu czasowego) w zależności od ustawień wysyłania. Ustawienia wysyłania do serwisu pastebin.com można zmienić w ustawieniach (zakładka [pastebin.com](#)).

12. Dodatki

12.1. canvas2d

canvas2d umożliwia proste rysowanie piksel po pikselu. Umożliwia zapisanie wygenerowanego obrazu.

Do jego obsługi można wykorzystać blok canvas2d (z menu kontekstowego), lub za pomocą kodu:

Na początek tworzymy okno do rysowania:

```
var canvas=addons.canvas2d(500, 400);
```

Gdzie 500 i 400 to szerokość i wysokość okna. Później możemy operować na stworzonym obiekcie:

GUI ⁷	Zwraca/ Typ	Sygnatura/ Nazwa	Opis
	---	setColor (r,g,b)	Ustawia kolor rysowania na kolor wyrażony w 3 liczbach (r, g i b)
X	---	setColor (color)	Ustawia kolor rysowania na wyrażony liczbą szesnastkową (np. 0xff0000)
X	---	drawPixel (x, y)	Rysuje piksel w na podanych współrzędnych
X	---	drawLine (x1, y1, x2, y2)	Rysuje linię od (x1;y1) do (x2;y2)
X	---	lineFrom (x, y)	Ustawia początek linii
X	---	lineTo (x, y)	Rysuje linię od punktu początkowego do podanego u ustawia nowy początek linii
	---	setAA (true/false)	Ustawia wygładzanie krawędzi (widoczne przy rysowaniu linii)
X	---	update ()	Aktualizacja widoku (gdy wyłączona automatyczna)
	boolean	autoUpdate	Automatyczna aktualizacja widoku, przypisuje się true, lub false
	Graphics2D	G	Obiekt Graphics2D, daje bezpośredni dostęp do Javowego obiektu Graphics2D
	---	addProgress (max)	Tworzy pasek postępu (max - ilość „kroków”)
	---	setProgress (i)	Ustawia wartość postępu (i - numer „kroku”)

Przykład:

```
canvas=addons.canvas2d(500, 400);
canvas.setAA(true);
canvas.moveLine(250,200);
canvas.lineTo(300,250);
canvas.drawLine();
```

⁷ GUI – kolumna informuje, czy dana funkcja jest dostępna z poziomu edytora bloku dla danego dodatku

12.2. logo

Logo służy do rysowania za pomocą pisaka obsługiwanego prostymi komendami.

Również posiada prosty edytor z gotowymi funkcjami

Najpierw należy stworzyć obiekt:

```
var logo= addons.logo(500, 400);
```

Gdzie 500 i 400 to szerokość i wysokość okna. Komendy wykonuje się jak metody stworzonego obiektu. Do dyspozycji są następujące metody (zarówno skróty jak i całość):

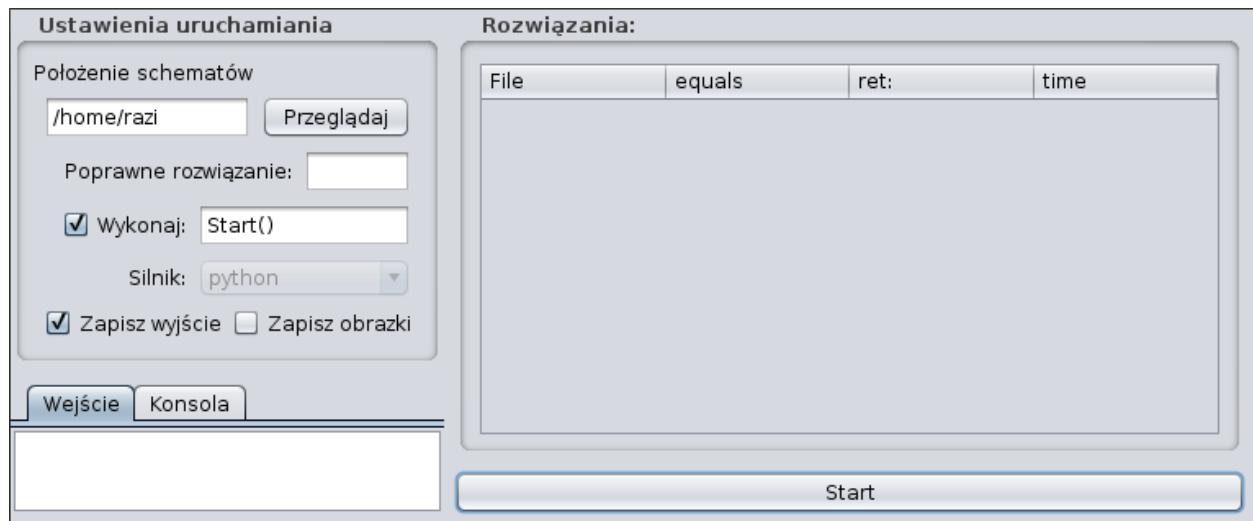
GUI	Sygnatura długa	Sygnatura skrócona	Opis
X	forward(x)	fw(x)	Do przodu o x punktów
X	backward(x)	bw(x)	W tył o x punktów
X	turnLeft(α)	tl(α)	Obróć w lewo o α stopni
X	turnRight(α)	tr(α)	Obróć w prawo o α stopni
X	dropPen()	dp()	Upuszcza pędzel, zaczyna rysować
X	pickPen()	pp()	Podnosi pędzel, przestaje rysować
X	hideTurtle()	hp()	Chowa tzw. żółwia
X	showTurtle()	sp()	Pokazuje żółwia
	setColor(r,g,b)	---	Ustawia kolor wyrażony trzema liczbami
X	setColor(color)	---	Ustawia kolor wyrażony jedną liczbą szesnastkową (np. 0xff0000)
X	fill()	---	Wypełnia aktualnym kolorem obszar, w którym znajduje się żółw
	eval(String)	e(String)	Wykonuje polecenie LOGO ze stringa (np. logo.eval("forward 100");

Przykład:

```
var logo= addons.logo(500, 400);
logo.setColor(155,10,40);
logo.fw(100);
logo.tl(90);
logo.forward(100); //dla przykładu - cała nazwa
logo.tl(90);
logo.fw(100);
logo.turnLeft(90);
logo.fw(100);
logo.tl(90);
```

Dostępny jest także edytor nieoparty o kod, dzięki któremu nie trzeba znać składni poleceń. Blok dla LOGO tworzy się z menu kontekstowego.

13. Sędzia



Okno sędziego

Sędzia to osobna aplikacja (musi być jednak w folderze z JavaBlock.jar i folderem *lib* z Pythonem) ułatwiająca sprawdzanie wielu schematów blokowych jednocześnie i porównywanie ich ze wzorcem. Konfiguracja:

- **Położenie schematów** - wklej ścieżkę do folderu z plikami *.jbf*.
- **Poprawne rozwiązanie** - nazwa pliku ze wzorowym rozwiązaniem
- **Wykonaj** - funkcja uruchamiająca schemat (razem z wartościami argumentów), aby schematy zostały wykonane, pole **Wykonaj** musi być zaznaczone
- **Zapisz wyjście** - zapisuje to, co wypisują wykonywane schematy
- **Zapisz obrazki** - zapisuje schematy jako obrazki

W zakładce **Wejście** wpisz (w każdej linijce osobno) dane wejściowe (wczytywane w bloku wejścia/wyjścia)

Zakładka **Konsola** wyświetla informacje na temat działania aplikacji. Jeżeli któryś ze schematów nie pasuje do formy (np. źle nazwana funkcja, lub ze złymi argumentami), wyświetli się informacja o tym.

Tabela **Rozwiązania** prezentuje wyniki dla każdego z rozwiązań:

- **File** - nazwa pliku
- **equals** - czy wyjście (przez blok wejścia/wyjścia) zgadza się ze wzorcowym rozwiązaniem
- **ret:** - zwrócona wartość (jeśli funkcje zwracają jakieś)
- **time** - czas wykonania, nie jest on dokładnym wyznacznikiem jakości algorytmu, gdyż dla każdego wykonania czas może się bardzo różnić.

Silnik JavaScript nie jest aktualnie wspierany w aplikacji Sędziego. Same przygotowanie skryptów do sprawdzenia może potrwać dość długo.

14. Rozwiązywanie problemów

14.1. Jak uruchomić program?

JavaBlock napisany został w Javie, przez co wymaga zainstalowanego Środowiska Uruchomieniowego Java (JRE) które można pobrać ze strony <http://www.java.com/pl/download/>. w przypadku systemów z menedżerami oprogramowania, wystarczy zainstalować z tego menedżera pakiet np. sun-java6-bin.

Jego plik wykonywalny ma rozszerzenie .jar. Wydania główne mają przygotowywane pliki .exe dla Windowsa. System powinien jednak domyślnie uruchamiać pliki .jar w Javie.

14.2. Program nie uruchamia się

Przed wszystkim upewnij się, czy jest zainstalowana Java i czy system uruchamia .jar w Javie.

Jeżeli program wcześniej działał, prawdopodobnie problem jest ze wcześniejszą konfiguracją. Wejdź do folderu `~/JavaBlock` (Unix), lub folder `_użytkownika/JavaBlock` (Windows) i usuń pliki *last.jbf* i *config.jbc*. Jeśli to nie pomoże, usuń pozostałe pliki w tym folderze.

14.3. Program nie eksportuje schematów do pastebin

Spróbuj ponownie za chwilę. Upewnij się, że masz połączenie z internetem, oraz program ma dostęp do sieci w Firewallu.

14.4. Błąd wykonania przy interwale równym 0

Symulacja schematów przy interwale równym 0, a dodatnim jest całkiem inna i może powodować pewne nieprawidłowości. Algorytm jest wtedy zamieniany w całości na kod JavaScript i uruchamiany, co uniemożliwia znalezienie konkretnego miejsca w którym znajduje się błąd/nieprzewidziana niezgodność.

Jeżeli przy interwale dodatnim algorytm wykonuje się bez problemu, wyślij plik na mój adres e-mail (rozdział Kontakt)

14.5. Błąd wykonywania schematu z wykorzystaniem innego schematu jako funkcji

Problem może leżeć po stronie generatora skryptu. Sprawdź (używając symulatora krok-po-kroku) z jakimi argumentami wywołuje daną funkcję, a następnie ręcznie uruchom ten schemat z takimi samymi argumentami (program zapyta o nie). Jeżeli problem nadal występuje, wyślij plik .jbf na moją pocztę (punkt Kontakt)

14.6. Stary schemat nie uruchamia się

JavaBlock jest intensywnie rozwijany, wprowadzane jest wiele zmian, co może powodować niezgodność z plikami stworzonymi w starszych wersjach (choć staram się zabezpieczać przed takimi niezgodnościami).

Można spróbować uruchomić plik starszą wersją, lub skopiować w dowolnym edytorze tekstu gałęzie `<flowchart>` ze starszego pliku do pliku utworzonego nowszą wersją.

Jeżeli otwiera się, a nie działa (a wcześniej działał), to otwórz edytory wszystkich bloków po kolei tak, aby zaktualizowały kod.

14.7. Błąd wykonania schematu mimo poprawności na innym komputerze

Spróbuj na innym silniku skryptu. Być może schemat ten jest przystosowany do Pythona, a twój *JavaBlock* ma tylko JavaScript.

14.8. Wersja apletowa jest przestarzała

Aplet na stronie głównej zawsze jest w najnowszej wersji, nie ma prawa być starsza. Jeśli jednak wydaje się starsza, to znaczy że twoja przeglądarka korzysta ze starszej wersji, którą ma w pamięci podręcznej. Wyczyść ją (pamięć podręczna, pamięć cache).

14.9. Program niemiłosiernie się tnie

Wyłącz antialiasing z menu: Ustawienia>Rysowanie>Wygładzanie krawędzi, upewnij się że jest włączona opcja „Prerenderowane grafiki”, możesz wyłączyć również rysowanie gradientów. Dodatkowo możesz zmienić motyw na „lżejszy”, np. systemowy (Linux: Gtk+, Windows: windows), lżejszy Metal, lub najlżejszy CDE/Motif. Włącz też (lub wyłącz) przyspieszenie sprzętowe.

15. Dla programistów

15.1. Dziennik zmian

a) 0.6

- Optymalizacje
- Poprawiono masę błędów

b) 0.5.5

- Nowy interfejs (zamieniony lewy panel na pasek narzędzi)
- Naprawiono kilka błędów
- Kilka optymalizacji obszaru roboczego

c) 0.5.4

- Przybliżanie do punktu
- Nowe wyróżnienie wykonywanego bloku
- Paski przewijania
- Wczytywanie ostatniego pliku
- Przeciąganie do siatki
- „Schematowe znaczkii” - zamiana np. znaku przypisania na strzałkę
- Zmodyfikowany edytor wejścia/wyjścia; blok może być rysowany ze standardowym kształtem
- Dzielenie połączenia podczas próby połączenia z nim
- Naprawione kilka błędów ze wczytywaniem schematów i przyspieszenie wczytywania
- Typ CharArray działa z Pythonem, z JS nie

d) 0.5.3

- 10-punktowy margines zaznaczania
- Poprawienie kilku błędów z Pythonem
- Implementacja JSyntaxKit (kolorowanie składni); opcjonalne (jako plugin)
- Zamiana znaków z programowania na matematyczne i bardziej „schematowe” (np. „←” zamiast „=”)

e) 0.5.2

- Poprawienie kilku starych błędów
- Przywrócenie do życia bloku skryptów osadzonych (niekompatybilne wstecz!)
- Blok struktury - generator struktur
- Poprawiony edytor bloku startowego

f) 0.5.1

- Łączenie bloków gdy to możliwe podczas usuwania
- Wstawianie bloku między połączone bloki (klawisz *p*)
- Interfejs kompaktowy, przystosowany do wąskiej przestrzeni (jak na stronach www między panelami)
- Poprawa obsługi argumentów
- Generowanie skryptów Python do aplikacji „Sędziego”
- Poprawienie generatora kodu Python; Python w ogóle zaczyna działać poprawnie :)
- Częściowa likwidacja błędu z UTF-8 i polskimi znakami

g) 0.5

- Poprawiono wiele błędów, ustabilizowanie działania programu
- Ustabilizowano działanie
- Wyświetlanie ustawień pastebin przy eksportowaniu
- Krzywe Beziera

h) 0.4.6 (0.5b)

- Funkcje
- Zmodyfikowano edytor bloku startowego.
- Przebudowano sposób generowania skryptów, wykonują się do 5 razy szybciej i nie tworzą stosu wywołanych funkcji (zajmują tyle samo RAMu niezależnie od ilości iteracji)
- Poprawiono wiele błędów
- Testowa obsługa Pythona (potrzebny plik jython.jar w folderze /lib)

i) 0.4.5 (0.5a)

- Zmodyfikowanie interfejsu rysowania: efekty podświetlania wskazanego bloku, odizolowanie rysowania bloków od połączeń
- Ukrywanie bloku Przeskok, gdy ma tylko jedno połączenie wchodzące; możliwość tworzenia za ich pomocą łamanych linii
- Edytor Logo i Canvas2d
- „Wstaw nowy blok tego typu” w edytorze bloków. Wstawia nowy blok tego samego typu co edytowany nieco niżej i wciśnięty w połączenia
- Tryb pełnoekranowy
- Zmieniony format .jbf (program wczytuje starsze pliki, ale starsze wersje programu będą miały problem ze wczytaniem nowszych plików)
- Kolorowanie zmian w tabeli
- Osobne okienko dla „konsoli” interpretera

j) 0.4.4

- Wybór motywów (domyślny motyw: Nimbus)
- Czyszczenie pamięci po stworzeniu nowego/otwarcia schematu
- Edytor bloku Wejścia/Wyjścia nieopartego na kodzie
- „Cienie”

k) 0.4.3

- Renderowanie ikon bloków
- Przygotowania do bloków nieopartych na kodzie
- Poprawa zachowania grup
- Poprawa zapisu PNG

l) 0.4.2

- Gruntowna przebudowa mechanizmu symulacji:
 - Wielowątkowość dla symulowania schematów
 - Rozdzielenie paneli kontroli symulacji dla każdego schematu osobno
- „Wejście” i jego zapis do pliku
- Zapis do pliku czasu interwału
- Edytor kodu w menu kontekstowym

m) 0.4.1

- Poprawiony konfigurator
- Loader klas (`addons.load(„nazwa”)`)
- Poprawione kilka błędów z odczytem
- Schowek - kopiowanie, wycinanie i wklejanie bloków (nieukończone)

15.2. Plany:

- Edytor połączeń, warstwy edycji (warstwa bloków, grup i połączeń), rozwinięcie możliwości przestrzeni roboczej
- Wysyłanie obrazków bezpośrednio do ImageShack.us, lub inny serwis
- Menu zakładkowe

15.3. JavaScript, a Python

Argumenty za wprowadzeniem Pythona:

- Łatwiejsze typowanie danych
- Trwałość typów danych
- Szybszy (nawet do 3 razy szybszy, testowane na atraktorze)
- Tworzy nowe stopy dla zmiennych, co zapewnia bezpieczeństwo przed nadpisaniem wartości zmiennych

Argumenty przeciw:

- Twardsze zasady programowania
- Ciężka biblioteka uniemożliwiająca załadowanie w aplecie, chyba że użytkownik pobierze bibliotekę Jython na swój dysk wcześniej

Aby użyć silnika Jython (Python dla Javy) ściągnij wersję JavaBlock z Pythonem.

Następnie w ustawieniach programu w zakładce „Ogólne” wybierz *jython*.

Silnik w fazie testowej, może nie działać sprawnie.

15.4. Źródła

Źródła będą uwolnione w okolicach czerwca 2011 roku.

15.5. Pluginy - nowe obiekty

Pluginy w postaci plików `.class` umieszczamy w folderze `~/JavaBlock/classes` (gdzie `~/` to na Windowsie adres do folderu użytkownika).

Konstruktor klasy może przyjąć (ale nie musi) jeden argument - `ScriptEngine`. Nic nie stoi na przeszkodzie stworzenia dodatkowych metod inicjujących pobierających np. `canvas2d`.

Obiekt takiej klasy zwraca funkcja `addons.load(String)`, gdzie za `String` podajemy nazwę klasy.

Przewiduję serwer przechowujący takie dodatki, oraz downloader z poziomu programu.

15.6. Skrypt inicjujący silnik JavaScript

W pliku `.jar` w folderze config znajduje się plik `scriptInit.js`, który zawiera „bindy”

do funkcji z JavaScript i dostęp do obiektów z Javy.

Można dodać tam własne funkcje. Jest możliwość dodawania własnych lokalnych skryptów tworząc pliki ze skryptami w folderze `~/JavaBlock/scripts` z rozszerzeniem `.js`, gdzie `~/` to folder użytkownika: na unixach `~/`, Windows: `C:\users\user`, lub `C:\Documents and settings\user`

15.7. Składnia pliku `.jbf`

`.jbf` to dokument XML. Głównym węzłem jest `<JavaBlocks>`. Bezpośrednio w nim znajdują się ustawienia globalne pliku. Po nich następują opisy zawartych schematów.

Każdy opis schematu rozpoczyna się od opisu widoku - pozycja widoku oraz przybliżenie.

Następnie znajduje się węzeł opisu bloków (`<blocks>`)

W `<blocks>` opisywane są poszczególne bloki. Każdy ma przypisane unikalne ID służące do identyfikacji potrzebnej przy łączeniu, oraz typ zgodny z typem w Enum. Każdy blok jest opisywany przez elementy węzła:

- `options` - ustawienia wyświetlania (aktualnie tylko czy ma wyświetlać komentarz)
- `visual` - opis kolorów i wyglądu, oraz pozycji (w przyszłości także skala i obrót)
- `content` - treść kodu (zawartość, każda linia osobno)
- `comment` - komentarz (każda linia osobno)
- `connect` - opis połączeń - z którym ID, oraz o jakiej wartości (blok decyzyjny)
- `specjalne` - dla poszczególnych typów bloków

16. Kontakt

Jeżeli masz jakąś propozycję, znalazłeś jakiś bug, lub chcesz zdać relację z używania JavaBlock, skorzystaj z „Send a comment” w programie, lub pisz:

- Adres e-mail: razi91@o2.pl, lub xazax91@gmail.com
- GG: 4089770

Komentarze wysłane przez program będą wyświetlane na stronie głównej projektu: <http://javablock.sourceforge.net/?show=comments>