

20.04.2021

Mateusz Machura

Wydział: Automatyki Elektroniki i Informatyki

Kierunek: Informatyka

PROGRAMOWANIE KOMPUTERÓW

SPRAWOZDANIE PROJEKTOWE

Temat projektu:

Interpreter prostego języka programowania

Prowadzący:

dr inż. Krzysztof Taborek

1. Temat i realizacja projektu

Celem projektu było zrealizowanie algorytmu interpretera prostego, autorskiego języka programowania zapewniającego możliwość inicjacji podstawowych typów zmiennych oraz kontenera tablicy. Język powinien dostarczać wbudowane metody umożliwiające operacje na zaimplementowanych typach zmiennych. Drugim aspektem tematu projektu było wygenerowanie prostego środowiska programistycznego o podstawowej funkcjonalności edycji kodu źródłowego.

Interpretowany język jest własnego autorstwa, natomiast składnia i semantyka języka jest zaczerpnięta z istniejącego, powszechnie stosowanego języka programowania Python. Zrealizowane środowisko programistyczne oprócz samego narzędzia do edycji i analizy pisanego skryptu umożliwia ono uruchamianie skryptu w różnych trybach, podgląd zainicjowanych zmiennych i funkcji oraz dostarcza okno konsoli danych wyjściowych. Kod źródłowy jest bezpośrednio wprowadzany do środowiska przez użytkownika. Zakłada się, że użytkownik posiada podstawową wiedzę z zakresu programowania oraz zapoznał się z samą składnią i ideą działania języka, która jest przedstawiona w dalszej części dokumentacji.

Środowisko umożliwia uruchomienie wprowadzonego kodu na dwa sposoby. Pierwszy z nich to tryb „execute”, który powoduje natychmiastową Interpretację kodu i wyprowadzenie odpowiednich komunikatów na konsolę. Drugi to tryb „debug”, który pozwala na debugowanie kodu, przerywanie uruchomienia, wkroczenia w następną ścieżkę skryptu. W trybie tym użytkownik ma możliwość podejrzenia zmian jakie zachodzą na zmiennych i ich wartościach w wyniku uruchamiania kolejnych linii kodu.

2. Uogólniony schemat działania programu

Dalsza analiza rozwiązań zaimplementowanych w projekcie jest podzielona na zbiór klas i algorytmów wykorzystywanych przez Interpreter oraz środowisko programistyczne.

2.1 Interpreter

Na algorytm interpretera składa się główna klasa Interpreter, istnieje jeden obiekt tej klasy w programie, definiuje metodę odpowiedzialną za uruchomienie interpretacji utworzonego skryptu w środowisku. Klasa Interpreter tworzy obiekty klasy wirtualnej InterFunctions, której klasy dziedziczące reprezentują zachowanie takich komponentów wchodzących w skład języka jak pętle, konstrukcje warunkowe, definiowanie funkcji czy mechanizmy tworzenia zmiennych. Klasy dziedziczące po klasie InterFunctions mogą tworzyć obiekty klasy Variable, DynamicField w zależności od potrzeby danej klasy pochodnej. Klasa Variable to klasa wirtualna, której klasy pochodne reprezentują podstawowe typy zmiennych jakie mogą być inicjowane w tworzonej języku programowania. W ich skład wchodzi następujące typy danych: integer, double, boolean, string oraz kontener danych tablicy o rozszerzonej funkcjonalności. Klasa

DynamicField definiuje niektóre pola klas pochodnych klasy InterFunctions i w współpracy z klasą Field odpowiada za wykonywanie złożonych operacji matematycznych z uwzględnieniem wartości zwracanych przez zdefiniowane zmienne czy funkcje.

2.2 Środowisko programistyczne

Podstawową klasą odpowiedzialną za wygenerowanie obiektów Interfejsu GUI oraz wywoływanie odpowiednich zdarzeń i metod z nimi związanymi jest klasa Form1, która została wygenerowana automatycznie przez środowisko Visual studio. Istotnym polem klasy jest obiekt klasy Interpreter za pomocą, którego klasa Form1 jest w stanie uruchomić algorytm interpretera. Następnym polem klasy Form1 jest obiekt klasy RuntimeParse, który jest odpowiedzialny za odpowiednie reagowanie komponentów środowiska na zmiany wprowadzane w kodzie źródłowym w polu edycji kodu. Klasa aktualizuje drzewo utworzonych funkcji i zmiennych oraz odpowiada za nadawanie odpowiednich kolorów poszczególnym komponentom języka, co w znacznym stopniu poprawia czytelność pisanego kodu.

3. Specyfikacja zewnętrzna

3.1 Interfejs GUI

Rys. 3.1 Interfejs środowiska widoczny po uruchomieniu programu



Opis segmentów widocznych na rys. 3.1:

1. Przeznaczony do wprowadzenia kodu źródłowego użytkownika, zgodnego z obowiązującymi zasadami w Interpretowanym języku.
2. Wyświetla zainicjowane zmienne i funkcje, zmienia swoją zawartość w czasie edycji skryptu.
3. Konsola, wyświetlane są w niej wartości zdefiniowane przez programistę za pomocą funkcji `print()` oraz komunikaty o błędach.
4. Wyświetla wbudowane metody dla istniejących typów danych w języku, jest swego rodzaju pomocą dla programisty.

Znaczenie symboli:



Standardowe uruchomienie Interpretera, skrót (F5) .



Uruchomienie trybu debugowania, tryb ten pozwala na kontrolowanie przejść do kolejnych ścieżek kodu oraz podgląd wartości zmiennych, skrót (F6).



Przerwanie trybu debugowania, po jego naciśnięciu Interpreter zakończy prace w trybie standardowym, skrót (F9).



Jednokrotne naciśnięcie pozwala w trybie debugowania wkroczyć do kolejnej ścieżki kodu, skrót (F7).

3.2 Język programowania, logika i przykłady implementacji

W katalogu programu w folderze „Sources” udostępniono pliki tekstowe zawierające kod źródłowy, które można otworzyć i uruchomić bezpośrednio w środowisku. Pliki te zawierają wybrane algorytmy zrealizowane w stworzonym języku programowania. Poniżej przedstawiono reguły obowiązujące w języku oraz przykłady.

Podstawowe Reguły obowiązujące w języku:

- Język jest typowany dynamicznie, oznacza to, że interpreter w czasie swojego działania przypisuje typ zmiennym.
- Zmienne deklarowane są poprzez użycie znaku '\$' i wymagają wstępnej inicjacji.
- Funkcje definiowane są poprzez użycie konstrukcji '#def '
- Jawny typ zmiennych może ulec zmianie w czasie Interpretacji kodu, jednak może być to niebezpieczne ze względu na dostępność wbudowanych metod dla określonych typów zmiennych.

- Do funkcji przekazywana jest zawsze oryginalna zmienna, aby zapobiec jej modyfikacji wewnątrz funkcji należy utworzyć jej kopie.
- Funkcja może zwracać dowolny typ zmiennej lub nie zwracać nic, należy uważać jeśli zwracana wartość jest przypisywana do zmiennej, a nie wszystkie ścieżki kodu zwracają wartość, w takim przypadku do zmiennej przypisywana jest wartość NULL, co może wygenerować błędy w przypadku jej użycia.
- Podstawową funkcją, która musi istnieć w skrypcie jest funkcja main(), w jej polu muszą zostać wywołane wszystkie zdefiniowane funkcje.
- Zmienną typu string deklaruje się poprzez użycie '... '.
- Kontener danych, tablice deklaruje się poprzez (...)
- Przerwanie działania z kodem -1 oznacza błędy Interpretacji, natomiast kod 0 oznacza brak błędów i tym samym uruchomienie wszystkich ścieżek kodu.
- Tablice można inicjować dowolnymi typami zmiennych.
- Nazwy zmiennych oraz funkcje mogą zawierać liczby, cyfry oraz znak podkreślenia.

Typy danych i wbudowane w język metody

Tab. 1, tabela wbudowanych metod języka.

Typ	Metoda	Przyjmowane argumenty	Zwracana Wartość	Funkcjonalność
Int	Sqrt	Int	Double	Pierwiastkowanie
	Pow	Int, Int/Double	Double	Potęgowanie
Double	Sqrt	Double	Double	Pierwiastkowanie
	Pow	Double, Int/Double	Double	Potęgowanie
String	UpdateIndex	String, Int, String	NULL	Aktualizacja znaku o zadanym indeksie
	Length	String	Int	Zwraca długość ciągu znakowego
	AtIndex	String, Int	String	Zwraca znak na zadanym indeksie
	IndexOf	String, String	Int	Zwraca indeks pierwszego wystąpienia znaku
Bool	True	–	Bool	zwraca wartość logiczną 1
	False	–	Bool	zwraca wartość logiczną 0
Array	UpdateIndex	Array, Int, Variable	NULL	Aktualizacja elementu o zadanym indeksie
	Length	Array	Int	Zwraca długość tablicy
	Append	Array, Variable	NULL	Dodaje element na koniec tablicy
	AtIndex	Array, Int	String	Zwraca znak na zadanym indeksie
	RemoveIndex	Array, Int	NULL	Usuwa element na zadanym indeksie
	print	Int/Double/String/Bool [inf]	NULL	Wypisuje wartość zmiennej na konsole
	input	NULL	Variable	Wczytuje zmienną z konsoli
	Type	Variable	String	Zwraca typ zmiennej

Variable – oznacza, że przyjmowana, bądź zwracana wartość może być dowolnego typu.

[inf] – oznacza nieograniczoną ilość argumentów.

Semantyka i struktury

- **Inicjalizacja zmiennych**

Zmienne należy inicjować poprzez użycie konstruktora '\$', wartość początkowa musi być zadeklarowana przed przystąpieniem do interpretacji.

Przykład skryptu inicjujący wszystkie dostępne typy zmiennych.

```

1  #def fun():
2      @
3  #main():
4      $x = 6
5      $a = -9.5
6      $b = True
7      $s = 'str'
8      $arr = (1,2,False,-7.2,'strrrr',(3,4))
9      print(x, ' ', a, ' ', b, ' ', s, ' ', arr(0), '\n')
10     print(Type(x), ' ', Type(a), ' ', Type(b), ' ', Type(s), ' ', Type(arr), '\n')
11
12     x = a
13     print(x, ' ', Type(x), '\n')
14     s = x
15     UpdateIndex(s,0,'a')
16     print(s, ' ', Type(s), '\n')
17
18     b = fun()
19     print(b)

```

6 -9.5 True str 1
 Int Double Bool String Array
 -10 Int
 a10 String
 NULL
 Execute time: 0,0335237 s.
 Exit code: 0

Skrypt inicjuje i wypisuje wartości zmiennych oraz ich typy. W linii 12 i 14 można zauważyć konwersje typów dzieje się to wtedy gdy przypisujemy zmienną jednego typu do innego, w efekcie czego przypisanie x do a daje wynikowy typ Int, a x do s daje String. W linii 18 do zmiennej zostaje przypisana wartość zwracana funkcji, funkcja zwraca NULL zatem taka wartość zostaje przypisana do zmiennej b, należy zadbać o możliwe wyeliminowanie takich sytuacji gdyż dalsze używanie zmiennej b będzie generować błędy interpretacji.

- **Pętla for** – struktura umożliwiająca cykliczne wykonywanie ciągu instrukcji określoną liczbę razy, do momentu zajścia pewnych warunków. W interpretowanym języku pętla for może przyjmować od trzech do czterech argumentów typu Int lub Double.
 1. Argument to nazwa zmiennej, do której wartości będzie można odwoływać się wewnątrz pętli, bądź już po jej wykonaniu.
 2. Argument to wartość początkowa zadeklarowanej zmiennej w argumencie pierwszym, oprócz wartości jawnej Int lub Double można tutaj podać nazwę zainicjowanej wcześniej w kodzie zmiennej, pod warunkiem, że spełnia założenia, co do typu.
 3. Argument jest wartością graniczną iteracji pętli, jeśli wartość zadeklarowanej zmiennej pętli osiągnie wartość równą lub większą od wartości tego argumentu, następuje opuszczenie pętli.
 4. Argument, którego wartość domyślna jest równa 1, deklaruje wartość jaka zostanie dodana do zainicjowanej zmiennej iterującej.

Przykład skryptu z użyciem pętli for

```

1  #main():
2      for(i, 0, 5):
3          print(i, ' ')
4      print('\n')
5      for(j, 5, 0, -1):
6          print(j, ' ')
7      print('\n')
8      for(j, 0, 10, 2):
9          print(j, ' ')
10
11

```

0 1 2 3 4
5 4 3 2 1
0 2 4 6 8
Execute time: 0,0216401 s.
Exit code: 0

- **Konstrukcja warunkowa if, elif, else** – struktury pozwalają na wykonanie różnych instrukcji w zależności od tego czy zdefiniowane przez programistę wyrażenie logiczne jest prawdziwe, czy fałszywe. Konstrukcje if, elif muszą przyjąć dwa argumenty tego samego typu aby było możliwe ich porównanie za pomocą dostępnych operatorów: ==, <>, <=, >=, <, > .

Przykład skryptu z użyciem konstrukcji warunkowych

```

1  #main():
2      $arr=(3, 8, -2, 0, True, False, ())
3      for(i,0,Length(arr)):
4          if(Type(arr(i)) == 'Int'):
5              if(arr(i) < 0):
6                  print('Zmienna: ', arr(i), ' jest typu integer i jest mniejsza od zera','\\n')
7              elif(arr(i) > 0):
8                  print('Zmienna: ', arr(i), ' jest typu integer i jest wieksza od zera','\\n')
9              else:
10                 print('Zmienna: ', arr(i), ' jest typu integer i jest rowna zero','\\n')
11          elif(Type(arr(i)) == 'Bool'):
12              print('Zmienna: ', arr(i), ' jest typu bool','\\n')
13          elif(Type(arr(i)) == 'Array'):
14              if(Length(arr(i)) <> 0):
15                  print('Zmienna na pozycji: ', i, ' jest tablica nie pusta','\\n')
16              else:
17                  print('Zmienna na pozycji: ', i, ' jest tablica pusta','\\n')
18
19

```

Zmienna: 3 jest typu integer i jest wieksza od zera
Zmienna: 8 jest typu integer i jest wieksza od zera
Zmienna: -2 jest typu integer i jest mniejsza od zera
Zmienna: 0 jest typu integer i jest rowna zero
Zmienna: True jest typu bool
Zmienna: False jest typu bool
Zmienna na pozycji: 6 jest tablica pusta
Execute time: 0,0232186 s.

- **Konstrukcja #def** – służy do definiowania funkcji.

Przykład skryptu z użyciem definiowanej funkcji

```

1  #def funkcja(zmienna):
2      if(zmienna == 2):
3          zmienna = zmienna+4
4      elif(zmienna >= 5.5):
5          @1
6      else:
7          @Sqrt(zmienna)
8
9  #main():
10     $z = 4.5
11     print(funkcja(z), '\n')
12     $x = 2
13     funkcja(x)
14     print(x, '\n')
15     print(funkcja(10), '\n')
16

```

2,12132034355964
6
1
Exit code: 0

W powyższym przykładzie zainicjowano zmienne oraz wywołano zdefiniowaną funkcję. W przypadku zmiennej 'x' można zauważyć, że po przekazaniu jej do funkcji jej wartość została zmodyfikowana. Funkcja Sqrt() jest funkcją wbudowaną w język i zwraca pierwiastek kwadratowy z podanej liczby. Nie wszystkie ścieżki kodu funkcji muszą zwracać wartość, natomiast takie użycie funkcji może być niebezpieczne w sytuacji gdy wykonanie funkcji zakończy się po pierwszym warunku if, a wartość zwracana ma zostać przypisana do zmiennej, wtedy zostaje zwrócona wartość NULL, co może wygenerować błąd w przypadku użycia takiej zmiennej.

- **Operatory arytmetyczne**

Język udostępnia pięć podstawowych operatorów arytmetycznych: +, -, *, /, %, ich funkcjonalność jest adekwatna do tej w powszechnie stosowanych językach programowania.

Przykłady zaimplementowanych prostych algorytmów

Rys. 3. 1, Skrypt, Przykład 1

```

1  #def silnia_rekurencyjnie(n, o):
2      if(n<2):
3          @
4          o=0*n
5          silnia_rekurencyjnie(n-1, o)
6
7
8  #def silnia_iteracyjnie(n):
9      $t = 1
10     for(x, 1, n+1):
11         t = t*x
12     @t
13
14 #def fib(n):
15     $a = 0
16     $b = 1
17     $t = 0
18     for(i, 0, n):
19         t = a+b
20         a = b
21         b = t
22     @a
23
24 #main():
25     $wynik = 1
26     print('silnia liczby 5:', '\n')
27     silnia_rekurencyjnie(5, wynik)
28     print(' ', 'Rekurencyjnie: ', wynik, '\n')
29     $f = silnia_iteracyjnie(5)
30     print(' ', 'Iteracyjnie: ', f, '\n')
31     print('Wyraz 14 ciagu Fibonacciego:', '\n' )
32     $wy = 1
33     $g = fib(14)
34     print(' ', g, '\n')
35
36
37 silnia liczby 5:
38 Rekurencyjnie: 120
39 Iteracyjnie: 120
40 Wyraz 14 ciagu Fibonacciego:
41 377
42
43 Exit code: 0

```





Przedstawione zostały trzy algorytmy, dwa pierwsze to funkcje liczące silnie podanej liczby w sposób rekurencyjny i iteracyjny, trzecia funkcja fib() wyznacza wartość liczby na n-tej pozycji ciągu Fibonacciego.

Rys. 3. 2, Skrypt, Przykład 2

```

1  #def alloc(m,n):
2      $temp = ()
3      for(x,0,n):
4          Append(temp,0.0)
5      for(y,0,n):
6          Append(m,temp)
7
8  #def print_macierz(ma):
9      for(w,0,Length(ma)):
10         for(c,0,Length(ma(w))):
11             print(ma(w,c), ' ')
12         print('\n')
13     print('\n')
14
15 #def mnozenie(m1,m2):
16     $m=()
17     alloc(m,Length(m1))
18     for(row,0,Length(m1)):
19         for(col,0,Length(m1)):
20             for(in,0,Length(m2)):
21                 m(row, col) = m(row, col)+m1(row,in)*m2(in,col)
22
23     @m
24
25 #main():
26     $m2 = ((2,-1,5),(6,0,4))
27     $m1 = ((0.5,-4),(3,2),(1,1))
28     $macierz = mnozenie(m1,m2)
29     print_macierz(macierz)
30
31
32

```

 -23 -0,5 -13,5
 18 -3 23
 8 -1 9
Exit code: 0


W przykładzie przedstawiono algorytm mnożenia macierzy. Macierze są reprezentowane jako dwuwymiarowa tablica, funkcja mnozenie() wykorzystuje funkcje alloc(), w celu zainicjowania macierzy wynikowej zerami. W wyniku mnożenia macierzy powstaje dwuwymiarowa tablica, która jest zwracana przez funkcje mnozenie() i wypisywana w funkcji main().

Rys. 3. 3, Skrypt, Przykład 3

```

1  #def Bubble_sort(list):
2      $len = Length(list)
3      $temp = 0
4      for(i, 0, len-1):
5          for(j, 0, len-i-1):
6              if(list(j) > list(j+1)):
7                  temp = list(j)
8                  list(j) = list(j+1)
9                  list(j+1) = temp
10
11 #main():
12     $tab = (1,2,8,11,-5,4,23,9,0,-2)
13     Bubble_sort(tab)
14     for(x, 0, Length(tab)):
15         print(tab(x), ' ')
16
17
18 -5 -2 0 1 2 4 8 9 11 23
19 Exit code: 0

```

Przykład przedstawia realizację algorytmu sortowania bąbelkowego (ang. bubble sort) dla tablicy wartości typu Int. W konsoli wypisano zawartość tablicy po jej modyfikacji przez funkcję Bubble_sort().

Rys 3.4. Skrypt, Przykład 4

```

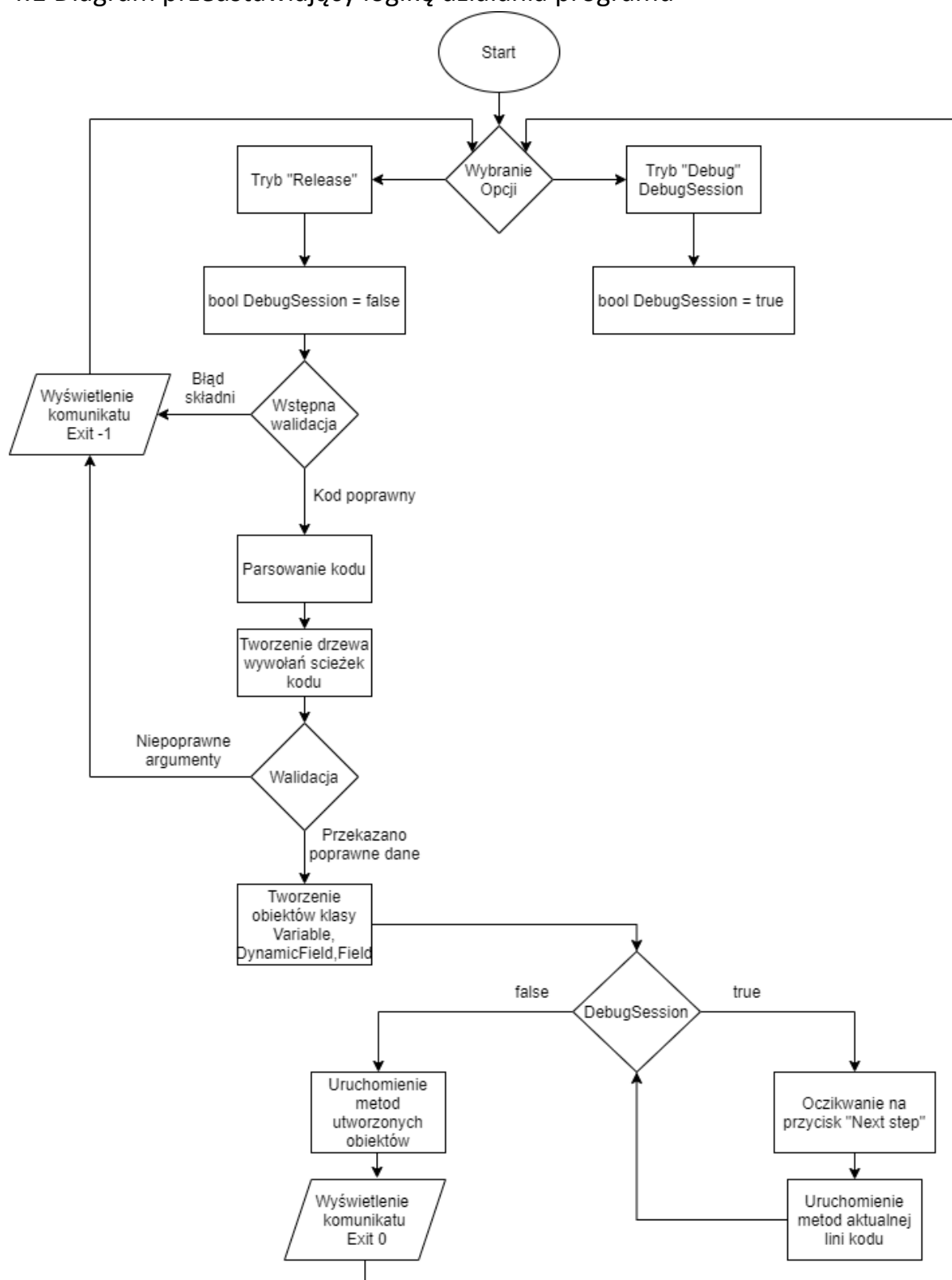
1  #main():
2      $array = ()
3      for(x,0,5):
4          $t = input()
5          Append(array, t)
6
7
8      for(i,0,Length(array)):
9          print(array(i), ' ')
10
11
12 4
13 False
14 -9
15 3,5
16 'znaki'
17 4 False -9 3,5 znaki
18 Exit code: 0

```

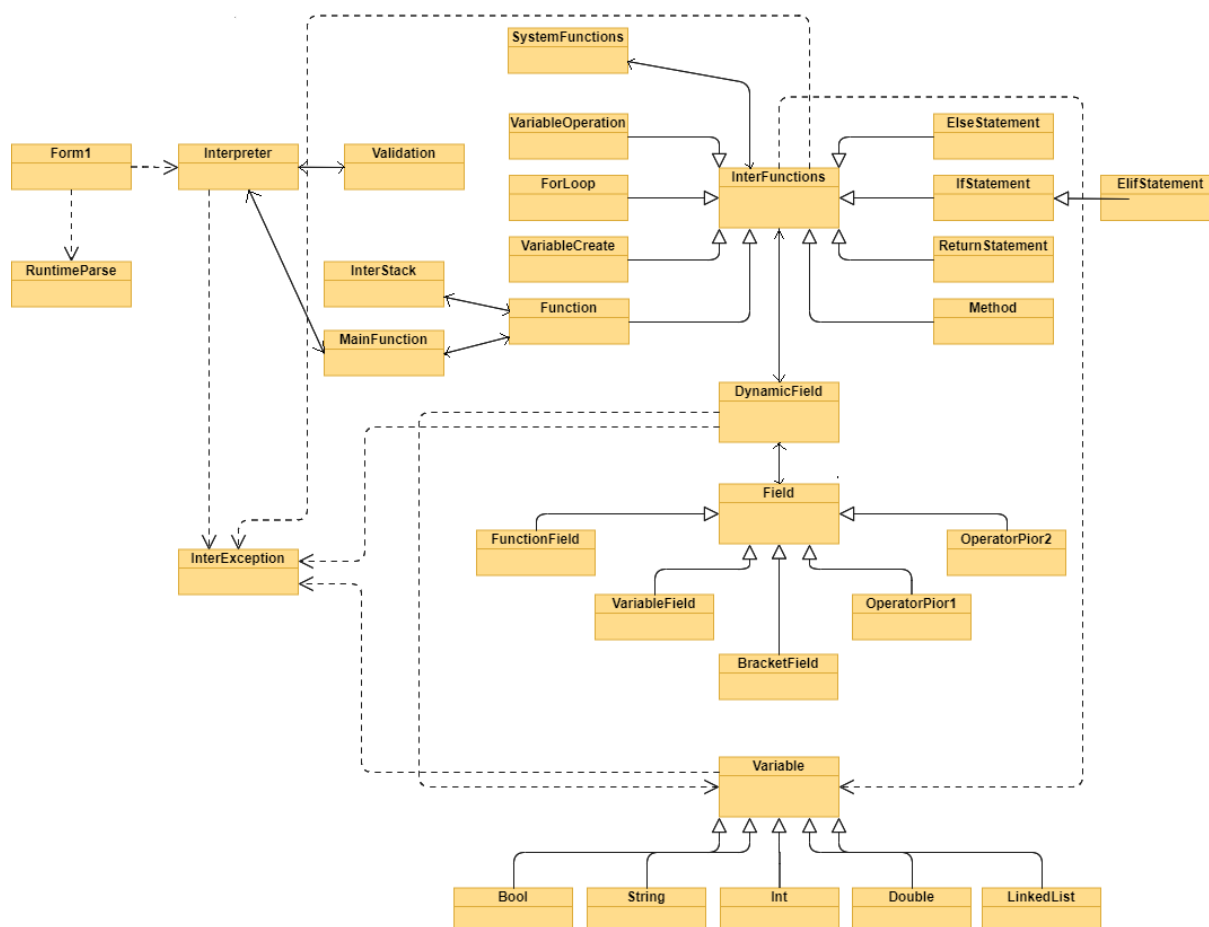
W przykładzie przedstawiono działanie funkcji input(). Funkcja ta przyjmuje 5 wartości wpisywanych do konsoli dodając je jako kolejne elementy tablicy za pomocą funkcji Append().

4. Specyfikacja wewnętrzna

4.1 Diagram przedstawiający logikę działania programu



4.2 Diagram klas



Poniżej zaprezentowano opisy klas znajdujących się na diagramie, wraz z ich kluczowymi polami i metodami.

1. Klasa abstrakcyjna `InterFunctions` - klasy dziedziczące odpowiadają za symulowanie działania podstawowych komponentów języka, a sama klasa bazowa przechowuje listę obiektów jej klas pochodnych, która bierze udział w generowaniu drzewa wywołań ścieżek kodu.
 - Klasa `ForLoop` symuluje działanie pętli.
 - Klasa `VariableOperation` definiuje operacje przypisania wartości zmiennej.
 - Klasa `VariableCreate` odpowiada za tworzenie zmiennych lokalnych funkcji.
 - Klasy `IfStatement`, `ElseStatement`, `ElifStatement` (dziedzicząca po `IfStatement`) symuluje działanie konstrukcji warunkowych.
 - Klasa `ReturnStatement` odpowiada za wygenerowanie i zwrócenie wartości przez funkcje.
 - Klasa `Method` odpowiada za wywołanie funkcji zdefiniowanej przez użytkownika, bądź wbudowanej.

- Klasa Function przechowuje obiekty InterFunctions, jednocześnie będąc obiektem tej klasy abstrakcyjnej, przechowuje zbiór zmiennych lokalnych zainicjowany wewnątrz niej.

1.1 Metody

- Action() - jest nadpisywana w klasach dziedziczących i odpowiada za wywołanie odpowiednich funkcjonalności związanych z poszczególnymi komponentami.
- GlobalFunctionInitialize() - inicjuje słownik funkcji przy każdym uruchomieniu Interpretera.
- AddToGlobalFunctions() – dodaje do słownika funkcji, funkcje zdefiniowane przez użytkownika.
- Parse() – jest funkcją prywatną każdej z klas dziedziczących po klasie InterFunctions, odpowiada za odpowiednią inicjację zmiennych prywatnych tych klas.

1.2 Pola

- Dictionary<string, Func<List<Variable>, Variable>> functions – przechowuje nazwy i wskaźniki na funkcje wbudowane w język oraz te zdefiniowane przez użytkownika.
- List<InterFunctions> child_list – przechowuje liste kolejnych kroków drzewa wywołań kodu.

2. Klasa SystemFunctions przechowuje statyczne metody, które definiują wbudowane metody języka, wywoływane w klasie InterFunctions.

3. Klasa DynamicField jest składową każdej z klas dziedziczących InterFunctions, odpowiada za uruchomienie operacji matematycznych oraz parsowanie kodu zawierającego takie operacje.

3.1 Metody

- DynamicFieldSeparate() – parsuje wejściowy ciąg znaków w taki sposób aby było możliwe rozdzielenie go i zainicjowanie odpowiednich klas.
- FieldAction() – odpowiada za wywołanie akcji związanych z przechowywanymi obiektami klasy Field.

3.1 Pola

- List<Field> MainElements – przechowuje obiekty klasy Field aby możliwe było wywołanie odpowiednich funkcji klas dziedziczących po Field.

4. Klasa abstrakcyjna Field - klasy dziedziczące to pewnego rodzaju podział operacji matematycznych jakie umożliwia język. Klasa powstała ze względu na potrzebę wykonywania operacji arytmetycznych nie tylko na liczbach ale również na wartościach zwracanych przez funkcje, czy tych aktualnie przechowywanych przez zmienne. Dzięki czemu zainicjowane równania matematyczne wykonują się w czasie Interpretacji i uwzględniają zmieniające się wartości zmiennych, a napisany kod nie jest statycznym bytem.

- Klasa `FunctionField` przechowuje wyseparowaną funkcję z równania matematycznego, następnie inicjuje jej argumenty jako `DynamicField`.
- Klasa `VariableField` przechowuje wysperowane zmienne z równania.
- Klasa `BracketField` traktowana jest jako klasa nadrzędna w przypadku uruchomienia kodu, ponieważ zapewnia wykonanie równania zgodnie z zasadami obowiązującymi w matematyce uwzględniając kolejność wykonywania działań, zawartość nawiasu dalej jest przekształcana w `DynamicField`.
- Klasa `OperatorPior1` przechowuje pozycje w równaniu operatorów mnożenia, dzielenia, modulo.
- Klasa `OperatorPior2` przechowuje pozycje w równaniu operatorów dodawania i odejmowania.

4.1 Metody

- `ActionfieldElements()` - jest nadpisywana w klasach dziedziczących i odpowiada za wywołanie odpowiednich funkcjonalności związanych z poszczególnymi komponentami.

4.1 Pola

- `List<Field> Childelements` - przechowuje listę kolejnych kroków drzewa wywołań kodu dla obiektów klasy `Field`.

5. Klasa abstrakcyjna `Variable` – klasy pochodne definiują zmienne możliwe do inicjalizacji w Interpretowanym języku i ich zachowanie w przypadku wywołania na ich rzecz metod wbudowanych w język.

- `Int` definiuje zmienną liczbową typu całkowitego.
- `Double` definiuje zmienną liczbową typu zmiennoprzecinkowego.
- `String` definiuje ciąg znakowy.
- `Bool` definiuje wartości logiczne `True` i `False`
- `LinkedList` definiuje kontener danych zwany tablicą, której rozmiar może się zmieniać w trakcie wykonywania kodu skryptu.

5.1 Metody

- `ReturnType()` – rozpoznaje i tworzy odpowiedni typ zmiennej z wprowadzonego kodu źródłowego.

5.2 Pola

- `string name` – przechowuje nazwę deklarowanej zmiennej.
- `dynamic value` – przechowuje wartość zmiennej.

6. Klasa `Validation` odpowiada za wstępną walidację kodu przed przystąpieniem do Interpretacji, jeśli wykryje błąd składni proces Interpretacji jest przerywany a odpowiednie komunikaty wyświetlone zostają w konsoli. Klasa definiuje także statyczne metody używane przez inne klasy w celu validacji przekazywanych danych.

6.1 Metody

- `Validate()` – przyjmuje wprowadzony kod źródłowy i sprawdza jego poprawność.

6.2 Pola

- Wszystkie pola klasy są prywatne i definiują wyrażenia regularne używane następnie w poszczególnych jej metodach.
7. Klasa InterStack przechowuje kopie obiektów klasy Function i monitoruje ich wywoływanie, jeśli ilość wywołanych po sobie obiektów klasy Function przekroczy zdefiniowany zakres, Interpretacja kończy się błędem, a komunikat „Stack overflow” zostaje wyświetlony w konsoli.

7.1 Metody

- OnStack() dodaje obiekt klasy Function na stos.
- RemoveStackLast() usuwa ze stosu ostatnio wywołaną funkcję.
- RemoveStackSpecific() uswa określona funkcję ze stosu

7.2 Pola

- List<Function> StackFunctions – reprezentuje stos.
- Function CurrentFunction – przechowuje funkcję z końca stosu, aby zapewnić szybszy dostęp do obiektu przez inne klasy.

8. Klasa Interpreter odpowiada za wstępne parsowanie kodu, weryfikację podstawowych błędów składniowych. Klasa inicjuje obiekt klasy MainFunction, który odpowiada za uruchomienie kodu. Klasa tworzy drzewo wywołań zbudowane z obiektów klasy InterFunctions.

8.1 Metody

- Interpret() – uruchamia akcje związane z Interpretacją kodu.
- InterpretDebug() - uruchamia akcje związane z Interpretacją kodu w trybie debugowania.

8.1 Pola

- MainFunction mainFunction – przechowuje obiekt klasy MainFunction za pomocą którego jest w stanie wywołać kod.
- Validation validation – dostęp do obiektu pozwala na wywołanie funkcji Validate() klasy Validation i wstępnego sprawdzenia poprawności składni kodu.
- RichTextBox ConsoleTextBox – wskaźnik na konsolę, umożliwia sterowanie funkcjami obiektu RichTextBox z poziomu klas powiązanych z klasą Interpreter.

9. Klasa Mainfunction - przechowuje zbiór zdefiniowanych obiektów klasy Function oraz swoje własne zmienne lokalne. W klasie tej rozpoczyna się proces wykonywania kodu oraz kończy czyszczeniem drzewa wywołań klas InterFunctions i DynamicField.

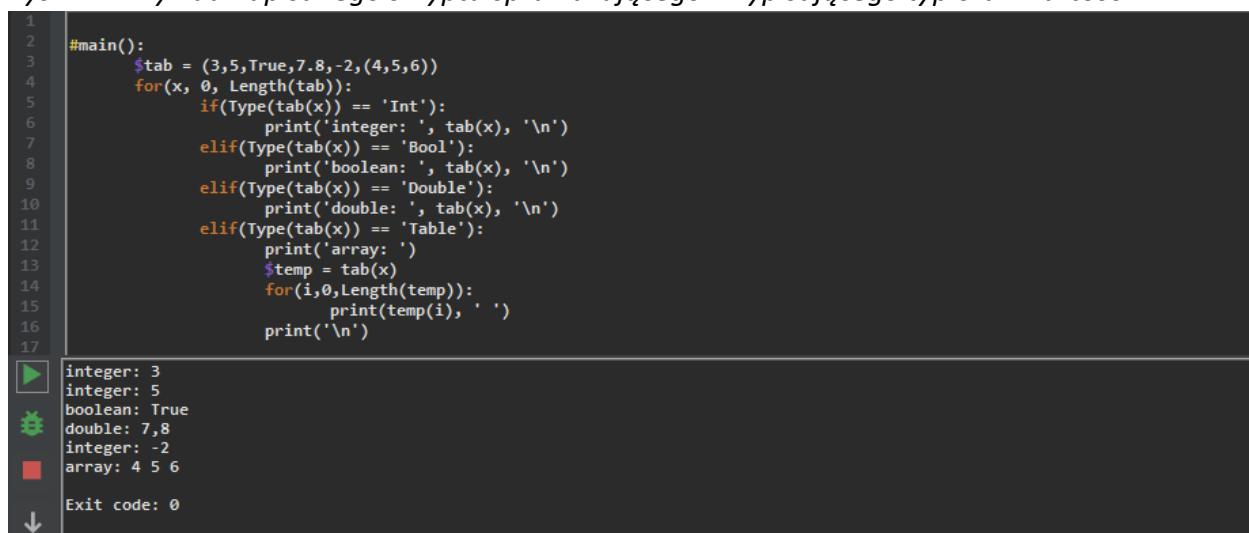
9.1 Metody

- Start() – metoda uruchamia drzewo wywołań całego wczytanego kodu źródłowego.

10. Klasa `InterException` jest klasą dziedziczącą po wbudowanej klasie `Exception`, dzięki jej zdefiniowaniu błędy jakie napotkane zostaną w czasie Interpretacji są wyświetlane bezpośrednio w konsoli podczas wywołania jej konstruktora.
11. Klasa `RuntimeParse` odpowiada za stylizowanie kodu podczas jego wprowadzania do obszaru 1 (rys. 1) oraz odpowiednie umieszczenie definiowanych zmiennych i funkcji w obszarze 2 (rys. 1).
12. Klasa `Form1` jest klasą wygenerowaną automatycznie przez środowisko programistyczne, dodatkowo zdefiniowano w niej zdarzenia takie jak naciśnięcie przycisku czy zmiany w wprowadzanym kodzie źródłowym, co z kolei uruchamia odpowiednie metody klasy `RuntimeParse`.

4.3 Tworzenie drzewa wywołań obiektów klasy `InterFunctions`.

Rys. 4.1 Przykład napisanego skryptu sprawdzającego i wypisującego typ oraz wartość.



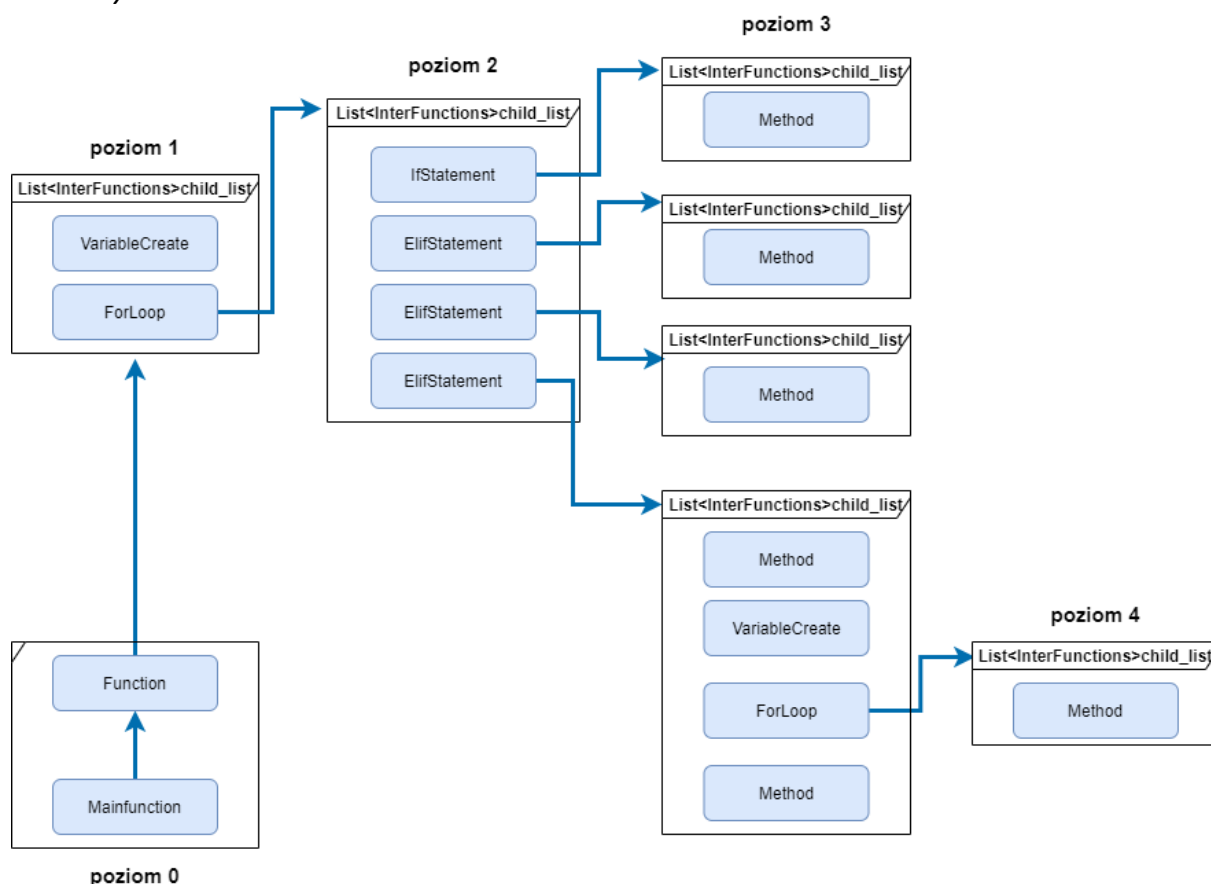
```

1  #main():
2
3  $tab = (3,5,True,7.8,-2,(4,5,6))
4  for(x, 0, Length(tab)):
5      if(Type(tab(x)) == 'Int'):
6          print('integer: ', tab(x), '\n')
7      elif(Type(tab(x)) == 'Bool'):
8          print('boolean: ', tab(x), '\n')
9      elif(Type(tab(x)) == 'Double'):
10         print('double: ', tab(x), '\n')
11     elif(Type(tab(x)) == 'Table'):
12         print('array: ')
13         $temp = tab(x)
14         for(i,0,Length(temp)):
15             print(temp(i), ' ')
16         print('\n')
17

```

integer: 3
integer: 5
boolean: True
double: 7,8
integer: -2
array: 4 5 6
Exit code: 0

Rys. 4.2 Diagram reprezentujący utworzone obiekty w programie na rzecz wywołania kodu z rys. 4.1



Na diagramie została przedstawiona idea tworzenia drzewa wywołań dla poszczególnych obiektów klasy `InterFunctions`, poszczególne kontenery widoczne na diagramie reprezentują zawartość listy 'child_list' znajdującą się w polu klasy `InterFunctions` i przechowującą obiekty tejże klasy. Interpretacja kodu odbywa się poprzez iterowanie po liście 'child_list' i uruchamianie dla każdego obiektu metody `Action()`. Takie podejście zapewnia wywołanie ścieżek zdefiniowanego kodu w taki sposób, że w pierwszej kolejności będą wywołane w sposób rekurencyjny metody elementów znajdujących się coraz dalej od funkcji wywołującej `MainFunction`, następnie jeśli lista na najdalszym poziomie zostanie przeiterowana oznacza to powrót na poziom niższy i dalszą iterację poprzez wkraczanie w coraz bardziej zagnieżdżone poziomy i powroty, w efekcie czego Interpretacja kodu zaczyna się oraz kończy w metodzie `Start()` obiektu `Mainfunction`.

4.4 Algorytm tworzący drzewo wywołań klas DynamicField i Field

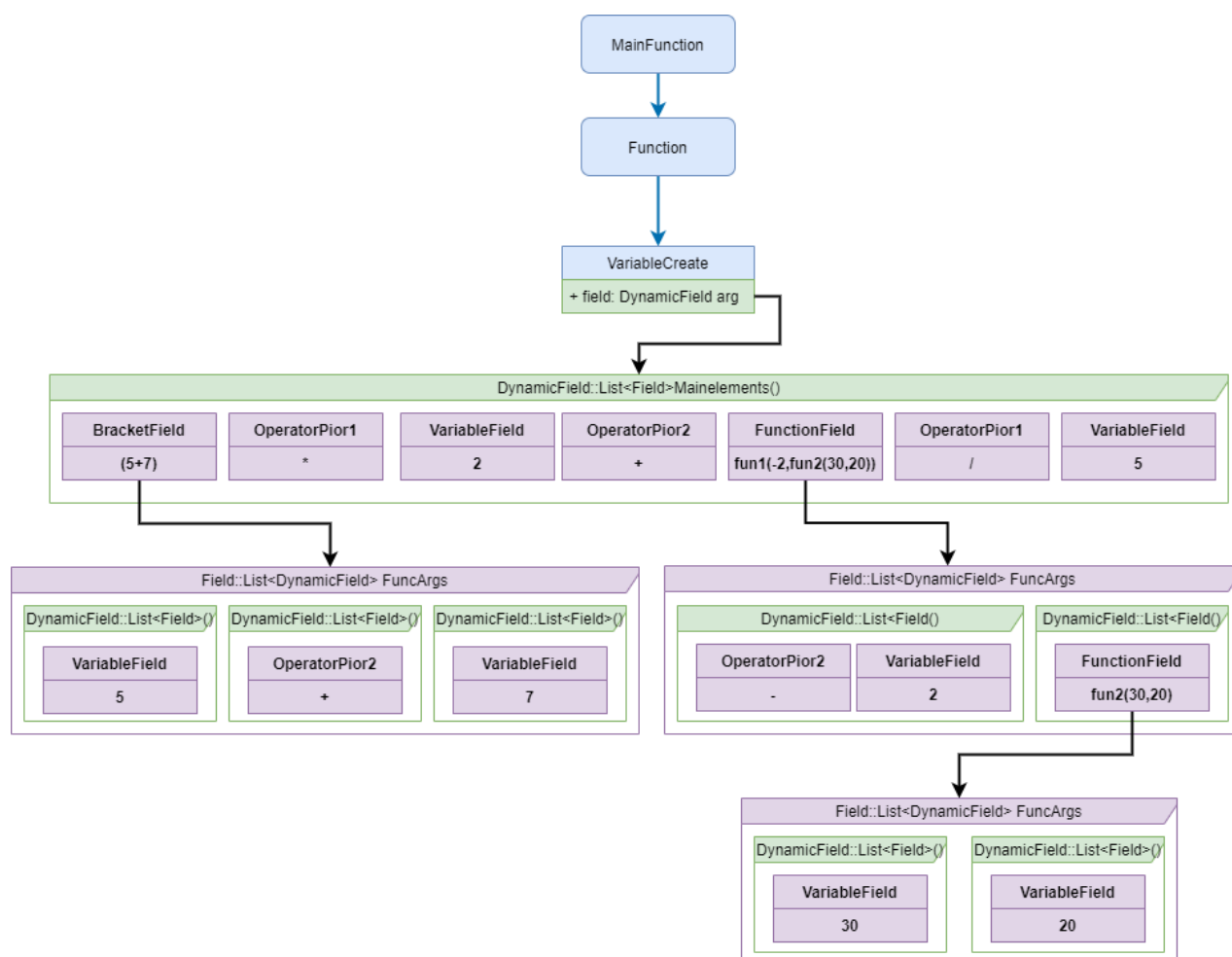
Rys. 4.3 Przykładowy skrypt

```

1 #def fun1(a, b):
2   @a+b
3
4 #def fun2(a, b):
5   @a-b
6
7 #main():
8   $z = (5+7)*2+fun1(-2, fun2(30, 20))/5
9   print(z)
10
11 25,6
Exit code: 0

```

Rys. 4.4 Diagram reprezentujący utworzone obiekty w programie na rzecz wywołania kodu z rys. 4.3, kolorami oznaczono obiekty klasy, bądź jej klas dziedziczących. Zielonym klasy DynamicField, fioletowym klasy Field, a niebieskim klasy InterFunctions.



Na diagramie z rys. 4.4 został przedstawiony sposób w jaki Interpreter reprezentuje obiekty `DynamicField`, które znajdują się między innymi w polach klas pochodnych od `InterFunctions`. Ciąg wejściowy jest dzielony poprzez operatory logiczne na obiekty pochodne klasy `Field`, jeśli obiektem jest `BracketField` lub `FunctionField` procedura jest powtarzana rekurencyjnie aż obiekty te ulegną całkowitemu rozkładowi, to znaczy, że najbardziej zagnieżdżone obiekty w drzewie wywołań będą obiektami klas `VariableField`, `OperatorPior1` lub `OperatorPior2`. Oczekuje się, że po uruchomieniu metody `FieldAction()` klasy `DynamicField` zwrócony zostanie obiekt pochodny klasy `Variable`. Metoda `FieldAction()` iteruje po liście `Mainelements` klasy `DynamicField`, jeśli napotkany zostanie obiekt klasy `BracketField` lub `FunctionField` metoda ta wywoływana jest rekurencyjnie dla zbudowanego poddrzewa tych obiektów, w efekcie czego uzyskany zostaje obiekt klasy `VariableField`. W kolejnej iteracji w liście `Mainelements` znajdują się już tylko obiekty reprezentujące operatory i zmienne, zatem zostają uruchomione metody klasy `OperatorPior1`, dzięki czemu kolejność wykonywania operacji matematycznych jest zgodna z zasadami matematycznymi. W trzeciej iteracji zostają uruchomione metody klasy `OperatorPior2` w ten sposób w liście `Mainelements` zostaje jeden obiekt klasy `VariableField`, który jest zwrócony przez funkcję, a tak wygenerowana jego wartość jest następnie przetwarzana przez klasę pochodną od `InterFunctions`, na rzecz, której operacje zostały wywołane.

4.5 Techniki obiektowe

Techniką obiektowa, która jest wykorzystywana przez klasy `InterFunctions`, `Field` oraz `Variable` jest polimorfizm dynamiczny, metoda polega na przesłanianiu metod wirtualnych klasy bazowej na rzecz metod klas pochodnych. Klasy pochodne tworzą Interfejs polimorficzny, oznacza to, że wszystkie obiekty wchodzące w skład drzewa wywołań dla klasy `InterFunctions` i `Field` są wywoływane tą samą metodą przez, co wykonywane są różne operacje w zależności od obiektu a nazwa metody pozostaje ta sama dla wszystkich obiektów, zatem nie zachodzi potrzeba sprawdzania typu obiektu przed wywołaniem metody.

4.6 Technologia

- Program był rozwijany w środowisku Visual Studio 2019
- Język programowania C# 9.0
- Struktura .NET Framework 4.7.2

4.7 Wykorzystywane biblioteki

- System.Collections.Generic
- System.Text.RegularExpressions
- System.Threading
- System.Windows.Forms
- System.Linq
- System.Drawing
- System.IO
- System.Threading.Tasks

5. Testowanie i uruchamianie

Pierwszym krokiem poprzedzającym testowanie poprzez uruchomienie programu była inspekcja kodu w celu znalezienia niezainicjowanych zmiennych, sprawdzenia czy wszystkie zmienne są używane oraz weryfikacji czy odwoływanie do elementów kontenerów danych poprzez indeks może przekroczyć zakres.

Metody przed implementacją w programie były poddawane testom jednostkowym, szczególną wagę zwrócono tutaj na definiowanie obiektów klasy Regex, testy te przeprowadzane były zgodnie z ideą „Black-box”, która zakłada brak wglądu w kod źródłowy a jedynie poddawanie testom funkcji i próbie wprowadzenia takich argumentów aby doprowadzić do wygenerowania błędu, bądź wyprowadzenia innych niż oczekiwane dane wyjściowe. Po przeprowadzeniu testów jednostkowych dla kluczowych metod programu podejmowano testy Integracji systemu, w celu sprawdzenia czy nowo dodana funkcja nie wpływa na generowanie błędów w innych komponentach programu.

Ostatni etap testowania nastąpił po wprowadzeniu wszystkich funkcjonalności programu. Testy były oparte na użytkowaniu oprogramowania i sprawdzaniu wszystkich funkcjonalności Interfejsu i różnych sekwencji uruchomienia, a generowane błędy były na bieżąco diagnozowane i usuwane.