# TKOM - Dokumentacja Finalna

**Język: Polang**

Maciej Scheffer

28 maja 2025

## 1 Temat Pracy

Celem projektu jest wykonanie interpretera własnego języka według poniższych wymagań:

### Indywidualne wymagania

- **Statycznie** oraz **silnie** typować dane,
- Zmienne mają być **domyślnie mutowalne**,
- Obiekty mają być przekazywane przez **referencję**,

### Ogólne wymagania języka

- Posiadać co najmniej **dwa operatory na funkcjach**,
- Wspierać definiowanie **funkcji wyższych**,
- Operować na podstawowych typach danych liczbowych (`int`, `float`, `bool`),
- Udostępniać **operacje arytmetyczne** oraz **logiczne**, wraz z priorytetem oraz nawiasami,
- Umożliwiać pisanie **komentarzy**,
- Zmienne mają **określony czas żywotności**,
- Zawierać przynajmniej jedną **instrukcję warunkową**,
- Zawierać przynajmniej jedną **pętlę**.

## 2 Opis realizacji

Zadanie zrealizowane zostało w postaci języka programowania imitującego polską mowę. Założeniem było jak największe upodobnienie języka do mówionych zdań, dbając jednocześnie o zwięzłość struktury. Słowa kluczowe języka są zatem w języku polskim, zrezygnowałem również z używania znaków specjalnych z paroma wyjątkami drastycznie szkodzącymi czytelności (np. lewy/prawy nawias), pominąłem też zasady interpunkcji.

```
numer x to 10.
numer y to 5.

powtórz 10 razy od
       jeżeli x większe_niż y wtedy od
            Wypisz("x > y").
       do
       inaczej gdy x mniejsze_niż x wtedy od
            Wypisz("y > x").
       do
       ostatecznie od
            Wypisz("y = x").
       do
       y to y plus 1.
do
```

# 3 Analiza wymagań

- **statyczne i silne typowanie**: Wykonane za pomocą obiektu zmiennej przetrzymywanej w kontekście. Interpreter przed przypisaniem wartości / zdefiniowaniem funkcji sprawdza najpierw jej typ. Język umożliwia jednak jawne rzutowanie na inne typy.
- **przekazywanie poprzez referencję**: Funkcja tworzy kopię argumentów w swoim kontekście, aby możliwe było przekazywanie wartości przez referencję obiekty zmiennych oraz funkcji mają flagę 'isReference', która wpływa na zachowanie getterów i setterów.
- **funkcje jako oddzielny typ**: W ramach uproszczenia wszystkie zmienne przechowujące funkcje są w praktyce natychmiastowo konwertowane na zdefiniowane funkcje.
- **priorytety oraz nawiasy**: Kolejność wykonywania równań została wykonana poprzez polimorfizm wyrażeń oraz odpowiednie umiejscawianie ich w drzewie przez parser.
- **określony czas żywotności zmiennych**: wykonany poprzez wektor zakresów który jest dynamicznie aktualizowany.

# 4 Założenia Języka

## 4.1 Typy Danych

| Typ danych | Słowo kluczowe | Wartości |
|---|---|---|
| Całkowity | **numer** | [-2147483648; 2147483647] |
| Zmiennoprzecinkowy | **ułamek** | [-inf ; inf], przecinek oddziela część całkowitą |
| Logiczny | **fakt** | prawda / fałsz |
| Znakowy | **wyraz** | ciągi znakowe w "" |
| Funkcja | **praca** | praca <nazwa>(<argumenty>) dająca <typ> |

Tabela 1: Typy danych

## 4.2 Operatory danych

Ponieważ nie chcemy niejawnie konwertować danych, operatory działają tylko na zmiennych o tym samym typie

| Operacja | Słowo kluczowe | numer | ułamek | fakt | wyraz |
|---|---|---|---|---|---|
| mnożenie | **razy** | TAK | TAK | NIE | NIE |
| dodawanie | **plus** | TAK | TAK | NIE | NIE |
| odejmowanie | **minus** | TAK | TAK | NIE | NIE |
| konkatenacja | **złącz** | NIE | NIE | NIE | TAK |
| dzielenie | **przez** | TAK | TAK | NIE | NIE |
| negacja | **nie** | NIE | NIE | TAK | NIE |
| logiczny and | **oraz** | NIE | NIE | TAK | NIE |
| logiczny or | **lub** | NIE | NIE | TAK | NIE |
| równość | **równe** | TAK | NIE | TAK | TAK |
| nierówność | **nierówne** | TAK | NIE | TAK | TAK |
| większość | **większe_niż** | TAK | TAK | NIE | NIE |
| mniejszość | **mniejsze_niż** | TAK | TAK | NIE | NIE |

Tabela 2: Operatory

## 4.3 Znaki specjalne

Aby język przypominał pisownią polską mowę, zminimalizowałem użycie znaków specjalnych, w miarę możliwości zastępując je słowami, dbając jednak o zachowanie względnej czytelności (dlatego koniec komendy do '.', a nie "kropka", a '(' to nie "lewy nawias"

| Operacja | Słowo kluczowe |
|---|---|
| **koniec komendy** | '.' |
| **wymienianie (np argumentów)** | 'i' |
| **nawiasy bloku** (scope) | 'od' 'do' |
| **nawiasy priorytetu** | '(' ')' |
| **wartość pusta** (None) | 'nic' |
| **komentarz** | 'PS:' |

Tabela 3: Znaki specjalne

## 4.4 Zmienne

Ponieważ zmienne są domyślnie mutowalne, możliwe jest przypisanie im pustej wartości "nic"(odpowiednika "None"). Zmienna musi natomiast posiadać typ podczas definicji jak również przydzieloną wartość. Wartość "nic" różni się w zależności od typu, dla numeru i ułamka będzie to 0, dla faktu "fałsz", a dla wyrazu pusty wyraz (""). Zmiennych nie można przykrywać w tym samym bloku, ale można przydzielać im nowe wartości. Znakiem przypisania wartości jest wyraz "to".

```
numer x to 10.
numer z to nic.
numer x to z. PS: przykrycie nazwy zmiennej
x to z.
```

### 4.4.1 Zakres żywotności

Widoczność zmiennej oraz jej żywotność będzie definiowana przez znaki bloku "od""do"działające w identyczny sposób jak nawiasy klamrowe w językach rodziny C.

```
od
      numer x to 10.
do
x to x plus 10. PS: zmienna nie istnieje
numer x to 10.
```

Możliwe jest natomiast przykrywanie nazw w zagnieżdżonych zakresach

```
numer x to 10.
od
      numer x to 5.
      x to 20.
do
wypisz(x). PS: x = 10
```

Zmienne zdefiniowane poza zakresem są w nim również widoczne

```
numer x to 10.
od
      x to 20.
do
wypisz(x). PS: x = 20
```

### 4.4.2 Mutowalność

Wartość zmiennych można dowolnie zmieniać, jeżeli chcemy zrobić zmienną niemutowalną należy jej typ poprzedzić słówkiem kluczowym "stały" na przykład:

```
numer x to 10.
stały numer z to 5.
z to x. PS: przypisanie wartości stałej zmiennej
x to z.
```

### 4.4.3 Typowanie

Ponieważ typowanie jest silne oraz statyczne każda zmienna musi mieć przypisany typ, a konwersja między typami możliwa jest tylko jawnie poprzez zdefiniowane w języku słowo "na" przykładowo:

```
numer x to 10.
wyraz y to x na wyraz.
numer z to nic.
z to y na numer.
```

| **Typ źródłowy** | **numer** | **ułamek** | **fakt** | **wyraz** |
|---|---|---|---|---|
| numer | - | TAK | TAK | TAK |
| ułamek | TAK | - | TAK | TAK |
| fakt | TAK | TAK | - | TAK |
| wyraz | TAK | TAK | TAK | - |

Tabela 4: Możliwe konwersje typów

Jak widać każdy typ danych można rzutować na inny, jednak należy wziąć pod uwagę że niektóre konwersje mogą wyrzucić błąd np. rzutowanie wyrazu na numer, jeżeli wyraz zawiera litery.

## 4.5 Funkcje

Funkcje, nazywane tutaj "pracami" traktowane są jako osobny typ danych mający swoje argumenty oraz zwracany typ. Ponieważ przecinek zarezerwowany jest dla "ułamków", argumenty oddzielane są

słowem kluczowym "i". Prace definiuje się w następujący sposób:

**praca** <nazwa>(<argumenty>) **daje**<typ>**robi** od <instrukcje> do

Na przykład:

```
praca czy_większa(numer x i numer y) daje fakt robi od
      zwróć x większe_niż y.
do
```

Słowo return zostało zastąpione polskim "zwróć", które zwraca typ podany po słowie kluczowym "daje" w definicji. Jeżeli praca nie zwraca nic, należy intuicyjne napisać "daje nic", z racji że "nic" zastępuje słowo None.

Argumenty pracy są kopiowane w zakresie, który dana praca wpycha na stos zakresów w trakcie jej wywołania. Jeżeli argumentami są zmienne, a nie wartości, to zostaną one przekazane przez referencję.

Pracy nie można przeciążać, ani redefiniować. Typ danych jakim jest praca reprezentowany jest w następujący sposób:

```
praca (numer, numer) dająca fakt nowa_praca to czy_większa.
```

Z założenia zezwolona jest rekursja, zwracająca błąd jeżeli zagnieżdżenie zajdzie za daleko. Funkcje mogą być przekazywane w argumentach oraz zwracane.

### 4.5.1 Operatory na funkcjach

Operator "**przykład**"
Rzuca dowolną funkcję biorącą argumenty podstawowego typu (nie pobierającej innej funkcji) na funkcję bezargumentową, losując wartości jej zmiennych.
**numer**: [0; 1000]
**ułamek**: [0; 1]
**wyraz**: losowy ciąg 5 znaków
**fakt**: prawda / fałsz

```
praca Wypisz_wartości(numer a i fakt b i ułamek c i wyraz d) daje nic robi od
      Wypisz(a na wyraz złącz b na wyraz złącz c na wyraz złącz d).
do

praca () dająca nic Wypisz_los to przykład Wypisz_wartości.
Wypisz_los().
```

Operator "**skrócona_o**"
To funkcja Bind Front z podanych przykładów. Działa w identycznyc sposób: Dla otrzymanej funkcji N argumentowej oraz X argumentów, operator zwraca funkcję N-X argumentową, gdzie pierwsze X parametrów pierwotnej funkcji ma ustalone wartości na argumenty operatora.

```
praca Wypisz_wartości(numer a i fakt b i ułamek c i wyraz d) daje nic robi od
      Wypisz(a na wyraz złącz b na wyraz złącz c na wyraz złącz d).
do

praca (wyraz) dająca nic Wypisz_wyraz to skrócona_o (1 i prawda i 1,23)
Wypisz_wartości.
Wypisz_wyraz("hej!").

Możliwe jest łączenie ze sobą operatorów, jednak należy pamiętać, że przykład
zawsze zwróci funkcję bezargumentową.
```

### 4.5.2 Wbudowane funkcje

Jedyną wbudowaną funkcją jest funkcja "wypisz" biorąca argument dowolnego typu, wypisująca jego zawartość na konsolę. Przykład użycia

```
numer x to 25.
wypisz("Hej tutaj napis!").
wypisz((x plus 3) razy 2).
```

## 4.6 Komentarze

Aby uniknąć pisania znaków specjalnych, komentarz rozpoczyna się słowem kluczowym PS: i trwa do końca linii.

```
numer x to 4 plus 2. PS: Tutaj jest komentarz
```

## 4.7 Instrukcja Warunkowa

Taka sama metoda działania jak w C++, jedynie podmienienie słów kluczowych.

**jeżeli** <warunek> **wtedy** od <instrukcje> do
**inaczej gdy** <warunek> **wtedy** od <instrukcje> do
**ostatecznie** od <instrukcje> do

```
jeżeli x równe 10 wtedy od
      Wypisz("x to 10!").
do
inaczej gdy x równe 12 wtedy od
      Wypisz("x to 12!").
do
ostatecznie od
      Wypisz("nie wiadomo...").
do
```

## 4.8 Pętla

Mechanizm pętli jest bardzo podstawowy, przypominający lekko ten z Pythona, nie tworzy ona jednak zmiennej tymczasowej.

**powtórz** <numer> **krotnie** od <instrukcje> do

```
numer x to 0.
powtórz 10 krotnie od
      Wypisz("To jest " złącz x na wyraz złącz " wykonanie pętli").
      x to x plus 1.
do
```

## 4.9 EBNF

Składowe

```
program     = { statement };

scope       = "od", {statement}, "do";

statement   = (action, ".")
            | func_decl
            | conditional
            | loop
            | scope;

action      = var_decl
            | var_assign
            | expression
            | ret_action;
```

Pętle i instrukcje warunkowe

```
conditional = "jeżeli", expression, "wtedy", statement,
              {"inaczej_gdy", expression, "wtedy", statement},
              ["ostatecznie", scope];

loop        = "powtórz", expression, "krotnie", statement;
```

Funkcje

```
arg_list_t_n = "(", (type_var, identifier), {"i", (type_var, identifier)}, ")";

arg_list_t   = "(", type_var, {"i", type_var}, ")";

arg_list_v   = "(", expression, {"i", expression}, ")";

ret_action   = "zwróc", (expression | none);

func_decl    = "praca", identifier, arg_list_t_n, "daje", type_ret, "robi", scope
```

Wyrażenia

```
expression  = rel_exp, {logic_op, rel_exp};

rel_exp     = addit_exp, {relation_op, addit_exp};

addit_exp   = multi_exp, {additive_op, multi_exp};

multi_exp   = cast_exp, {multipli_op, cast_exp};

cast_exp    = unary_exp, {"na", type_decl_no_fun};

unary_exp   = value_expr
            | (unary_op, value_expr);

przyk_expr  = rozp_expr
            | (przykład_op, rozp_expr);

rozp_expr   =  value_expr
            | (rozpocz_op, arg_list_v, value_expr)

value_expr  = value, {arg_list_v};

value       = literal
            | identifier
            | ( "(", expression, ")" );

przykład_op = "przykład";
rozpocz_op  = "skrócona_o";

logic_op    = "oraz"
            | "lub";

unary_op    = "nie";

relation_op = "równe"
            | "nierówne"
            | "większe_niż"
            | "mniejsze_niż";

multipli_op = "razy"
            | "przez";

additive_op = "plus"
            | "minus"
            | "złącz";
```

Dane

```
var_assign  = identifier, "to", expression;

var_decl    = type_var, identifier, "to", (expression | none);

type_func   = "praca", arg_list_t, "dająca", type_ret;

type_ret    = type_decl
            | none;

type_var    = ["stały"], type_decl;

type_decl_no_fun    = "wyraz"
                    | "ułamek"
                    | "numer"
                    | "fakt";

type_decl   = "wyraz"
            | "ułamek"
            | "numer"
            | "fakt"
            | type_func;

none        = "nic";

literal     = string
            | float
            | integer
            | bool;

string      = '"', {char | escape}, '"';

float       = integer, ",", {digit};

integer     = ["-"], natural
            | "0";

bool        = "prawda"
            | "fałsz";
```

Elementarne

```
identifier  = letter, {letter | digit | "_"};

natural     = non_zero, {digit};

char        = letter | digit | spec_char;

escape     = "\", ("n" | "t" | "r" | "\" | '"');

digit       = non_zero | zero;

non_zero    = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";

zero        = "0";

letter      = "A" | "B" | "C" | "D" | "E" | "F" | "G"
            | "H" | "I" | "J" | "K" | "L" | "M" | "N"
            | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
            | "V" | "W" | "X" | "Y" | "Z" | "Ą" | "Ć"
            | "Ę" | "Ł" | "Ń" | "Ó" | "Ś" | "Ż" | "Ź"
            | "a" | "b" | "c" | "d" | "e" | "f" | "g"
            | "h" | "i" | "j" | "k" | "l" | "m" | "n"
            | "o" | "p" | "q" | "r" | "s" | "t" | "u"
            | "v" | "w" | "x" | "y" | "z" | "ą" | "ć"
            | "ę" | "ł" | "ń" | "ó" | "ś" | "ż" | "ź" ;

spec_char   = " " | "!" | "#" | "$" | "%" | "&" | "'" | "("
            | ")" | "*" | "+" | "," | "-" | "." | "/" | ":"
            | ";" | "<" | "=" | ">" | "?" | "@" | "[" | "]"
            | "^" | "_" | "`" | "{" | "|" | "}" | "~";
```

# 5 Testowanie i Błędy

Przetestowana została większość funkcjonalności języka, wraz z testami na rzucanie błędów. Testy znajdują się w repozytorium pod adekwatnymi nazwami, testując każdy z trzech modułów: leksera, parsera oraz interpretera. Pliki wykonywalne testów tworzone są razem z plikiem wykonywalnym całego projektu, co opisane zostało poniżej.

Błędy rzucane są oczywiście w języku polskim, dlatego ich poprawne działanie może wymagać zmiany kodowania znaków w konsoli. Przykładowy błąd:

```
wyraz a to 123 na wyraz.
numer x to "abc".

- - - - - - - - - konsola: - - - - - - - - -
Błąd Interpretera: [2:1]: Typ danej 'x' nie zgadza się z jej wartością.
```

# 6 Przykłady użycia

Bardziej skomplikowany warunek

```
praca sprawdź_warunek(numer a i fakt b i wyraz c) daje fakt robi od
       jeżeli a większe_niż 10
            oraz (a minus 3 plus 2) razy 5 na wyraz nierówne "20"
            lub c złącz "!" równe "hasło!" lub b równe prawda wtedy od
             wypisz("Warunek spełniony!").
             zwróć prawda.
       do ostatecznie od
             wypisz("Warunek niespełniony!").
             zwróć fałsz.
       do
do

numer x to 5.
sprawdź_warunek(x i fałsz i "hasło").
```

Funkcje wyższego rzędu

```
praca do_przekazania(numer x) daje nic robi od
      wypisz("niższa funkcja z numerem: " złącz x na wyraz).
      zwróć nic.
do

praca wysoka(praca (numer) dająca nic niższa i numer x) daje praca(numer)
dająca nic robi od
      praca wewnętrzna(numer z) daje nic robi od
            wypisz("Wywołanie wewnętrznej z numerem: " złącz z na wyraz).
            zwróć nic.
      do

      numer y to x.
      powtórz y krotnie od
            wypisz("To jest " złącz y na wyraz złącz " wykonanie pętli").
            niższa(y).
            y to y plus 1.
      do
      zwróć wewnętrzna.
do

wysoka(do_przekazania i 10)(100).
```

Rekursja

```
praca fib_rek(numer n) daje numer robi od
      jeżeli n mniejsze_niż 0 wtedy od
            zwróć 0.
      do
      jeżeli n równe 1 wtedy od
            zwróć 1.
      do
      zwróć fib_rek(n minus 1) plus fib_rek(n minus 2).
do
numer x to 0.
powtórz 10 krotnie
od
      x to x plus 1.
      wypisz(x na wyraz złącz ". liczba fibb to: "złącz fib_rek(x) na wyraz).
do
```

Generator liczb losowych

```
praca los(numer mnożnik i numer x) daje numer robi od
      zwróć mnożnik razy x.
do

wypisz("Losowa to: " złącz (przykład (skrócona_o (999) los))() na wyraz).
```

# 7 Sposób uruchomienia

Aby interpreter uruchomić najpierw należy go skompilować. Jest to możliwe dzięki pliku Makefile oraz toolchain'owi vcpkg z zainstalowaną biblioteką boost, przy pomocy komend:

**mkdir build**
**cd build**
**cmake .. -DCMAKE_TOOLCHAIN_FILE=C:/path/to/vcpkg/vcpkg.cmake**
**cmake –build .**
**chcp 65001**
**./Debug/main.exe [plik_testowy]**

Interpreter można uruchomić na dwa sposoby, albo uruchamiając w konsoli plik wykonywalny main.exe wraz z argumentem będącym ścieżką pliku docelowego, albo uruchamiając samo polecenie main.exe które umożliwi wpisywanie komend na bieżąco w konsoli. Aby komunikaty wypisywane przez interpreter były czytelne może być konieczne użycie komendy **chcp 65001** aby zmienić kodowanie terminala. **Kodowanie pliku testowego powinno być w formacie UTF-16 LE**.

# 8 Implementacja

**Lekser**: leniwe zamienianie ciągu znaków z konsoli/string'a/pliku tekstowego przy pomocy klas pomocnicznych 'reader'.
**Parser**: leniwe pobieranie tokenów z leksera transformując je w złożone drzewa 'statements'.
**Interpreter**: pobieranie gotowych drzew 'statements' następnie za pomocą wzorca wizytatora ewaluowanie każdego z liści modyfikując stos kontekstów przechowujących zmienne oraz funkcje.