# TKOM Projekt wstępny

Daniel Kobiałka, 310744

**Temat:**
Implementacja języka programowania obsługującego mechanizm Pattern Matching, zawierającego typowanie dynamiczne silne, zmienne domyślnie niemutowalne oraz przekazywanie przez kopię.
Zmienne są widoczne tylko w bloku, w którym zostały zadeklarowane.

Projekt zakłada napisanie w C++ interpretera zdefiniowanego przeze mnie języka programowania spełniającego powyższe kryteria.

## Przykłady konstrukcji językowych:

```
$ komentarz w kodzie do końca linii
zmienna = 1;            $ deklaracja niemutowalnej zmiennej typu int
mut a = 3;              $ deklaracja mutowalnej zmiennej typu int o nazwie ‘a’
a = ‘tekst’;            $ zmiana typu zmiennej a na String oraz zmiana jej wartości

a = 3 / 5               $ dzielenie
a = 3 // 5              $ dzielenie całkowitoliczbowe
a = 3 % 5               $ reszta z dzielenia
a = (3 + 5) * 8         $ złożone działanie arytmetyczne z użyciem nawiasów
a = Int (”2”)           $ rzutowanie zmiennej na inny typ

if zmienna == 1 {
      $ ciało instrukcji warunkowej ‘if’
      a = 4;            $ kod wykonywany w przypadku spełnienia warunku
}

loop a < 5 {
      $ ciało pętli ‘loop’
      a = a + 1;        $ zmiana wartości zmiennej i operacja dodawania
}

func templateFunction(a, b) {
      $ ciało funkcji o nazwie ‘templateFunction’ przyjmującej parametry
      $ a i b (są kopiowane)
      return a;   $ wartość zmiennej ‘a’ jest zwracana jako wynik funkcji
}

func templateFunction2(ref a, b) {
      $ ciało funkcji o nazwie ‘templateFunction2’ przyjmującej parametry
      $ a (przez referencję) i b (przez kopię)
      $ funkcja nic nie zwraca
```

```
}

$ przykład rekurencyjnego wywołania funkcji
func fibonacci(num) {
      if num <= 0 {
            return "Wprowadź poprawną wartość num > 0";
      }
      if num == 1 {
            return 0;
      }
      if num == 2 {
            return 1;
      }
      return fibonacci(num – 1) + fibonacci(num – 2);
}

$ wywołanie funkcji i użycie wartości przez nią zwróconej w instrukcji warunkowej
if templateFunction(a, b) > 6 {
      standardOutput(”wartość większa niż 6”); $ wypisanie na konsole
} else {
      standardOutput(”wartość mniejsza lub równa 6”);
}

mut para = 9, 1;        $ wbudowany typ danych do przechowywania pary wartości
a = para.first;         $ odwołanie się do pierwszego elementu pary
b = para.second;        $ odwołanie się do drugiego elementu pary

func even(a) {
      return a % 2 == 0;
}

$ pattern matching
pattern zmienna {
      match x, y {
            $ kod wykonywany jeśli ‘zmienna’ jest parą
            standardOutput(”pierwsza liczba z pary:”+x);   $ konkatenacja stringów
      }
      match String (s) {
            $ kod wykonywany jeśli ‘zmienna’ jest ciągiem znaków
            standardOutput(”ciąg znaków”);
      }
      match Float (f) {
            $ kod wykonywany jeśli ‘zmienna’ jest typu float
      }
      match (5 * 7 + 3) (x) {
            $ kod wykonywany jeśli ‘zmienna’ jest równa 38
      }
      match (even) (x) {
            $ kod wykonywany jeśli funkcja (w tym przypadku ‘even’) zwróci wartość
```

```
            $ prawdziwą, dla argumentu ‘zmienna’
      }
      none {
            $ kod wykonywany jeśli żaden z poprzednich wzorców nie pasuje
      }
}

if zmienna is Float {
      $ zmienna jest typu Float
}

if zmienna is x, 5 {
      $ zmienna jest typu Pair, a jej druga wartość wynosi 5
}

standardOutput(”jakiś tekst”);     $ wypisanie tekstu na standardowe wyjście
                                   $ funkcja wbudowana; domyślnie dopisuje znak
                                   $ nowej linii na końcu tekstu
```

## Gramatyka:

### Poziom leksyki:

```
operator          = ==|!=|<|>|<=|>=
digit             = [1-9]|[0]
nonZeroDigit      = [1-9]
float             = [0-9]+\.[0-9]+
int               = [1-9][0-9]*|0
string            = ”([^”\\]|\\.)*”
letter            = [A-Za-z]
any_character     = .
```

### Poziom składni:

```
program                 = {instruction}
block                   = ”{”, {instruction}, ”}”
instruction             = block | single_instruction | statement
single_instruction      = (function_declaration | variable_declaration
                          | assign_or_function_call)
assign_or_function_call = identifier, (assign | function_call), ”;”
function_call           = "(", [expression, {",", expression}], ”)”
statement               = if_statement | loop_statement | pattern_statement
                          | return_statement
variable_declaration    = ["mut”], identifier, ["=”, (expression | pair)], ”;”
function_declaration    = ”func”, identifier, arguments_list, block
```

```
arguments_list          = ”(", [argument, {",", argument}], ”)”
argument                = [”ref”], identifier
assign                  = "=”, (expression | pair)
pair                    = expression, expression
constant                = number | string
number                  = int | float
identifier              = (letter | ”_”), {letter | digit | ”_”}
if_statement            = "if”, expression, block, [else_statement]
else_statement          = "else”, block
loop_statement          = "loop”, expression, block
return_statement        = ”return”, [expression], ”;”
expression              = and_expression, {"||”, and_expression}
typename                = ”String” | ”Float” | ”Int”
and_expression          = relative_expression, {"&&”, relative_expression}
relative_expression     = (numeric_expression, {operator, numeric_expression})
                          | is_expression
numeric_expression      = term, {("+” | "-”), term }
term                    = factor, {("*” | "/” | "//” | "%”), factor};
factor                  = ["!" |"-"], constant | typename | id_or_function_call
                          | field | cast_or_nested
cast_or_nested          = [typename], "(", expression, ")”
id_or_function_call     = identifier, [function_call]
field                   = id_or_function_call, ”.”, ”first” | ”second”
match_statement         = "match", (match_expression | match_pair | match_string
                          | match_float | match_int | match_none)
match_expression        = expression, ”(”, identifier, ”)”, block
match_pair              = pair, block
match_string            = "String”, ”(”, identifier, ”)”, block
match_float             = "Float”, ”(”, identifier, ”)”, block
match_int               = "Int”, ”(”, identifier, ”)”, block
match_none              = "none", block
pattern_statement       = pattern”, expression, "{", {match_statement}, ”}”
numeric_pair            = numeric_expression, ",", numeric_expression
is_expression           = numeric_expression, {”is”, numeric_expression |
                          numeric_pair}
```

## Obsługa błędów:

Przykładowe komunikaty:

Błędy arytmetyczne:

Przykładowy komunikat: Arithmetic Error: Division by zero at: line 8, source.tko.

Błąd modyfikowania niemutowalnej zmiennej: Immutable Variable Error: Trying to modify variable a, which is immutable at: line 7, source.tko.

Brakujący nawias: Missing Bracket Error: Missing bracket at: line 16, source.tko.

## Sposób uruchomienia:

Interpreter uruchamiamy z linii poleceń z jednym argumentem – ścieżką do pliku z kodem źródłowym

`./interpreter source.tko`
Gdzie `source.tko` jest plikiem zawierającym kod źródłowy.

Wyjściem programu jest standardowe wyjście konsoli.

## Sposób realizacji projektu:

Projekt będzie podzielony na następujące moduły:
-analizator leksykalny
-analizator składniowy
-analizator semantyczny
-interpreter.

Źródłem, z którego będzie korzystał analizator leksykalny będzie ciąg znaków lub plik. Wejście z pliku będzie stanowić podstawę działania interpretera natomiast alternatywna opcja przyda się do testów.
Będą wyróżniane następujące typy tokenów:
-Identyfikatory,
-Stałe,
-Operatory,
-Komentarze,
-Słowa kluczowe,
-Symbol (np. nawias),
-Koniec pliku.
Analizator składniowy będzie grupował tokeny w drzewa. Pogrupowane tokeny będą analizowane przez analizator semantyczny, a następnie kod będzie interpretowany. Na każdym etapie mogą pojawić się błędy, które nie będą powodować zakończenia interpretacji, ale będą wyświetlane użytkownikowi. Istotnym aspektem będzie to, żeby żadne konstrukcje językowe nie spowodowały wystąpienia wyjątku w samym interpreterze.

## Opis testowania:

Do każdego modułu będą napisane testy jednostkowe sprawdzające, czy wszystko działa w typowych sytuacjach, ale też monitorujące poprawne zachowanie interpretera w przypadku błędów. Oprócz tego wykonam testy integracyjne w postaci programów w zdefiniowanym języku. Będzie kilka przypadków testowych, jak najbardziej złośliwych, np. plik o rozmiarze powyżej 4kB. Do testów jednostkowych najprawdopodobniej zostanie wykorzystana biblioteka Google Test.