TKOM Dokumentacja końcowa

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.

W ramach projektu został napisany w języku C++ interpreter zdefiniowanego przeze mnie języka spełniającego powyższe kryteria.

Język nie posiada niejawnych konwersji typów danych, posiada natomiast jawne. Nie jest możliwe porównywanie zmiennych różnych typów za pomocą operatorów relacji (z wyjątkiem operatora is, która po prostu zwraca 0 dla zmiennych różnych typów).
Zmienne są przekazywane zawsze przez kopie.

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(a, b) {
      $ ciało funkcji o nazwie ‘templateFunction2’ przyjmującej parametry
      $ a i b
      $ 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
            $ konkatenacja stringów
            standardOutput(”pierwsza liczba z pary:”+String(x));
      }
      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’
      }
      match 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:
Error: DIVISION_BY_ZERO
    line: 1 column: 13

Błąd modyfikowania niemutowalnej zmiennej o nazwie test:
Error: REASSIGN_IMMUTABLE_VARIABLE
    line: 3 column: 1
Reassign immutable variable: test

Brakujący nawias:
Error: BRACKET_EXPECTED
    line: 1 column: 7

Sposób uruchomienia:

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

`./code source.tko`
Gdzie `source.tko` jest plikiem zawierającym kod źródłowy.
`./code source.tko --print-tokens`
Zamiast interpretacji zostaną wyświetlone tokeny.
`./code source.tko --print-object-tree`
Zamiast interpretacji zostanie wyświetlone drzewo obiektów wygenerowane przez parser.

Wyjściem programu jest standardowe wyjście konsoli.

Sposób realizacji projektu:

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

Źródłem, z którego korzysta analizator leksykalny jest ciąg znaków lub plik. Wejście z pliku stanowi podstawę działania interpretera natomiast alternatywna opcja przydaje się w testach.
Wyróżniane są następujące typy tokenów:
-Identyfikatory,
-Stałe,
-Operatory,
-Komentarze,
-Słowa kluczowe,
-Symbol (np. nawias),
-Koniec pliku.

Analizator składniowy buduję drzewo obiektów, a następnie kod jest interpretowany poprzez przechodzenie po tym drzewie. Do tego celu został wykorzystany wzorzec projektowy wizytatora. Na każdym etapie mogą pojawić się błędy, które nie muszą powodować zakończenia interpretacji, ale są wyświetlane użytkownikowi. Żadna niepoprawna konstrukcja nie powoduje błędu w samym interpreterze.

Opis testowania:

Do każdego modułu są napisane testy jednostkowe sprawdzające, czy wszystko działa w typowych sytuacjach, ale też monitorujące poprawne zachowanie interpretera w przypadku błędów. Oprócz tego wykonane zostały testy integracyjne w postaci programów w zdefiniowanym języku. Jest kilka przypadków testowych, jak najbardziej złośliwych, sprawdzających np. czy wynik funkcji jest konsumowany tylko raz, lub czy zmienne nie są widoczne poza ich zakresem. Do testów została wykorzystana biblioteka Google Test.