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

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
}
```

```
$ 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 = Pair(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

$ pattern matching
pattern zmienna {
      match Pair(x, y) {
            $ kod wykonywany jeśli ‘zmienna’ jest parą
            standardOutput(‘pierwsza liczba z pary:’+x);   $ konkatenacja stringów
      }
      match String(x) {
            $ kod wykonywany jeśli ‘zmienna’ jest ciągiem znaków
            standardOutput(‘ciąg znaków’);
      }
      match x {
            $ kod wykonywany jeśli ‘zmienna’ jest dowolną zmienną
      }
      none {
            $ kod wykonywany jeśli żaden z poprzednich wzorców nie pasuje
      }
}

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 | comment}
block                  = ”{”, {instruction | comment}, ”}”
comment                = ”$”, {any_character}
instruction            = block | single_instruction | statement
single_instruction     = (declaration | assign | function_call), ”;”
function_call          = identifier, "(", {expression, ”,”}, ”)”
statement              = if_statement | loop_statement | pattern_statement
declaration            = [mut], identifier, ["=”, constant | pair]
assign                 = identifier, "=”, constant | pair
pair                   = ”Pair”, ”(”, constant, ",”, constant, ")”
constant               = number | string
number                 = int | float
identifier             = letter, {letter | digit}, {".first” | ".second”}
if_statement           = "if”, bool_expression, block, [else_statement]
else_statement         = "else”, block
loop_statement         = "loop”, bool_expression, block
expression             = and_expression, {"||”, and_expression}
and_expression         = relative_expression, {"&&”, relative_expression}
relative_expression    = numeric_expression, {operator, numeric_expression}
numeric_expression     = term, {("+” | "-”), term }
term                   = factor, {("*” | "/” | "//” | "%”), factor};
factor                 = ["!”], number | identifier | function_call
                         | "(", expression, ")”
match_expression       = (match_variable | match_pair | match_string |"none”),
                          block
match_variable         = "match”, identifier
match_pair             = "match”, ”Pair”, ”(”, identifier, ",”, identifier, ")”
match_string           = "match”, "String”, ”(”, identifier, ",”, identifier, ")”
pattern_expression     = "pattern”, identifier, "{", {match_expression}, ”}”
```

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.
Po wygenerowaniu przez lekser będą następnie przekazywane do analizatora składniowego w postaci wektora obiektów odpowiedniej klasy. 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.