[TKOM] Dokumentacja końcowa

Szymon Kisiel

Temat projektu

Interpreter języka z pozycją geograficzną jako niestandardowy typ danych.

Elementy języka

- Typ całkowitoliczbowy **int**
- Typ zmiennoprzecinkowy **float**
- Typ tekstowy **string**
- Obsługa zmiennych (statyczne, mutowalne, typowanie silne)
- Obsługa nawiasów
- Operacje matematyczne, logiczne oraz porównania
- Instrukcja warunkowa **if**, **elsif**, **else**
- Instrukcja pętli **while**
- Możliwość definiowania funkcji i ich wołania z argumentami przekazywanymi przez wartość
- Komentarze
    - liniowy: **#** …
    - blokowy: **/*** … ***/**

Niestandardowe typy danych:

- Symbole ^, ', '' będą używane jako odpowiednio: stopnie, minuty, sekundy
- Niestandardowy typ **geo** - pozycja geograficzna w postaci szerokości i długości geograficznej
    - Przykład: 55^ 45' 30'' N 37^ 37' 45'' E
    - Maksymalna szerokość geograficzna: 90^ N
    - Maksymalna szerokość geograficzna: 90^ S
    - Maksymalna długość geograficzna: 180^ W
    - Maksymalna długość geograficzna: 180^ E
- Niestandardowy typ **geocoord** - współrzędna geograficzna w stopniach, minutach i sekundach
    - Przykład: 55^ 45' 30''
- Niestandardowy typ **geodist** – odległość geograficzna
    - Przykład: 80^ 11' 22'', 280^ 40' 25''
    - Maksymalna długość wertykalna: 180^
    - Maksymalna długość horyzontalna: 360^
    - geodist jest wynikiem odejmowania typów geo
    - Możliwość dodania typu geodist do typu geo w celu uzyskania nowej wartości typu geo

Inne założenia:

- Warunek (w instrukcjach if, elsif i while) jest fałszywy tylko, gdy jest liczbą całkowitą o wartości 0 (int = 0).
- Instrukcja else jest przypisywana do najbliższej instrukcji if

Przykłady użycia języka

Operacje matematyczne

```
int zmienna = 2 > 1 and 2 != 1 and -1 <= 1;
int test = ((2+2*4)+(2*2+4)-0.5 == 17.5) and zmienna or 0;
print(test);
```

Niestandardowy typ

```
geo warszawa = 52^13'5''N  21^30''E;
geo gdansk = 54^20'51''N 18^38'43''E;
geodist diff = warszawa - gdansk;
print(diff);
```

Instrukcje warunkowe i pętle

```
int i = 0;
while (i < 10) {
    int j = 10;
    while (j > 0) {
        if (i == 5) {
            if (j == 5) {
                print("a");
            }
            else
                print("b");
        }
        elsif (i == 6) {
            print("c");
        }
        j = j - 1;
    }
    i = i + 1;
}
```

Funkcje

```
int factorial(int x) {
    if (x <= 1)
        return 1;
    return x * factorial(x-1);
}
int add(int x, int y) {
    return x + y;
}
void count(int x) {
    print(x);
    if (x > 0)
        count(x-1);
}
string concatenate(string text1, string text2) {
    return text1 + " " + text2;
}
```

```
print("count:");
count(5);

print("add:");
print(add(3, 6));

print("factorial:");
print(factorial(6));

print("concatenate:");
string test1 = "test";
string test2 = "konkatenacji";
print(concatenate(test1, test2));
```

Opis gramatyki

```
program = {statement | function} ;

function = type , id ,  "(" , [parameters] , ")" , "{" , {statement} , "}" ;

statement        = if_statement | while_statement | simple_statement
                 | "{" , {statement} , "}" ;
if_statement     = "if" , "(" , expression, ")" , statement
                 , {"elsif" , "(" , expression, ")" , statement}
                 , ["else" , statement] ;
while_statement  = "while" , "(" , expression, ")", statement ;
simple_statement = (var_declaration | assignment | function_call | return_statement) , ";";

var_declaration  = type , id, "=", expression ;
assignment       = id, "=", expression ;
function_call    = id ,  "(" , [arguments] , ")" ;
return_statement = "return" , [expression] ;

parameters = parameter , {"," , parameter} ;
parameter  = type , id ;
arguments  = argument , {"," , argument} ;
argument   = expression ;

expression      = add_expression , {comp_operator, add_expression } ;
add_expression  = mult_expression , {add_operator, mult_expression } ;
mult_expression = factor , { mult_operator , factor } ;
factor          = ["-"] , ( integer | float | id | function_call | "(" , expression , ")" )
                | string | geo_dist | geo | geo_coord  ;

id   = letter , {letter | digit | "_"} ;
type = "void" | "int" | "float" | "string" | "geo" | "geocoord" | "geodist" ;

comp_operator = "<" | ">" | "<=" | ">=" | "=="| "!=" ;
add_operator  = "+" | "-" | "or" ;
mult_operator = "*" | "/" | "and" ;

geo       = geo_coord , ("N" | "S") , [","] , geo_coord , ("W" | "E") ;
geo_dist  = geo_coord , "," , geo_coord ;
geo_coord = integer , "^" , [integer , "'"] , [integer , "‘‘"]
          | [integer , "^"] , integer , "'"  , [integer , "‘‘"]
         | [integer , "^"] , [integer , "'"] , integer , "‘‘" ;
```

```
string    = '"' , {character} , '"' ;
character = letter | digit | ... ;
letter    = "A" | "B" | "C" | ...

float          = integer , "." , digit, {digit}
integer        = zero | digit_not_zero , {digit} ;
digit          = zero | digit_not_zero ;
zero           = "0" ;
digit_not_zero = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

Komponenty

Lekser generuje kolejne tokeny na podstawie źródła danych (pliku lub ciągu znaków) .

Parser na podstawie kolejnych tokenów z leksera tworzy drzewo składniowe.

Wykonanie programu polega na wywołaniu metody execute na drzewie składniowym.

Wbudowana funkcja print pisze wartość argumentu do pliku wyjściowego output.txt.

Obsługa wyjątków

- Wyjątki leksera
  - Brak zakończenia typu tekstowego
  - Brak zakończenia komentarza
  - Przekroczenie maksymalnego rozmiaru typu danych
  - Nieznany token
  - Nieznany escape sequence
- Wyjątki parsera
  - Błędy składniowe
  - Błędy budowania typów geograficznych
- Wyjątki wykonania
  - Użycie niezadeklarowanej zmiennej
  - Użycie niezadeklarowanej funkcji
  - Ponowna deklaracja zmiennej
  - Ponowna deklaracja funkcji
  - Niepoprawna ilość argumentów przekazana do funkcji
  - Niepoprawne operacje matematyczne/logiczne (np. dodawanie typów tekstowych)
  - Niepoprawny typ wartości zwracanej w funkcji
  - Niepoprawny typ wartości przypisywanej do zmiennej
  - Niepoprawny typ wartości argumentu przekazanego do funkcji

Testowanie

Testy jednostkowe leksera – sprawdzenie poprawności tworzenia pojedynczych tokenów.

Testy parsera – porównanie wygenerowanego drzewa składniowego z oczekiwanym drzewem.

Testy interpretera – przekierowanie wyników na wyjście za pomocą funkcji wbudowanej print i porównanie ich z oczekiwanymi wartościami.