

# TKOM Projekt wstępny

Aleksandra Sypuła

## I. Temat projektu

Język umożliwiający tworzenie kolorowych scen z figurami geometrycznymi w 2D. Oprócz podstawowych funkcjonalności udostępnianych w tradycyjnych językach programowania, możliwe będzie tworzenie punktów, odcinków, figur czy kolekcji takich figur (będą one wbudowanymi typami języka). Użytkownik będzie mógł stworzyć dowolną liczbę oddzielnych scen z figurami, a także dla każdej z figur ustawić dowolnie wybrany kolor.

## II. Opis funkcjonalności

Obsługa podstawowych oraz niestandardowych typów danych wraz z dopuszczalnymi operacjami:

- Typy danych liczbowych: `int` – liczby całkowite, `double` – liczby zmiennoprzecinkowe
  - Operacje arytmetyczne ze standardowym priorytetem wykonania oraz obsługą nawiasów: dodawanie, odejmowanie, mnożenie, dzielenie; w przypadku operacji na różnych typach danych liczbowych, zmienne `'int'` będą konwertowane do typu `'double'` a wynik również będzie zwracany jako `'double'`
  - Porównania wartości liczb
- Typ znakowy – `string`
  - Konkatenacja
  - Brak automatycznej konkatenacji z innymi typami np. typu `int` oraz `string`
  - Obsługa metaznaków
- Typ `bool` przyjmujący wartości `'true'` lub `'false'`
- Typ listy z ograniczoną funkcjonalnością – `List`
  - Możliwość przechowywania dowolnych typów danych ale takich samych dla danej instancji listy
  - Funkcjonalność podobna do stosu – możliwość dodania elementu na koniec listy lub usunięcia, również z końca listy
- Typ reprezentujący punkt na płaszczyźnie – `Point`
  - Umożliwia stworzenie punktu ze współrzędnymi (x, y) będącymi liczbami całkowitymi
- Typ reprezentujący odcinek na płaszczyźnie – `Line` (dla uproszczenia `line` od `line segment`)
  - Umożliwia stworzenie odcinka poprzez przekazanie dwóch punktów
- Typ reprezentujący figurę na płaszczyźnie – `Figure`
  - Umożliwia stworzenie zamkniętej figury poprzez przekazanie listy odcinków w odpowiedniej kolejności
- Typ reprezentujący kolekcję figur – `FigCollection`
  - Lista przechowująca figury
  - Umożliwia dodawanie oraz wyświetlanie przechowywanych figur

Obsługa komentarzy

- Dopuszczalne jednolinijkowe komentarze poprzedzone znakiem `'#'`

Wyświetlanie ciągu znaków w konsoli za pomocą słowa kluczowego print

Wbudowane instrukcje:

- Warunkowa if else
- Pętla while

Tworzenie własnych zmiennych, przypisywanie do nich wartości oraz późniejsze odczytywanie:

- Typowanie dynamiczne
- Typowanie silne
- Przypisywanie mutowalne
- Zakres widoczności zmiennych zawężony będzie do zakresu, w którym występują (zostały zadeklarowane), ograniczonego przez nawiasy klamrowe '{', '}'

Wołanie i definiowanie własnych funkcji (ze zmiennymi lokalnymi), przekazywanie argumentów jedynie przez wartość. Definicja funkcji musi się zaczynać od słowa kluczowego 'function' z podaniem nazwy funkcji, następnie nawiasów z określeniem zmiennych i ciałem funkcji w nawiasach klamrowych.

Obsługa błędów

### III. Przykłady języka

# example program, this comment should be ignored

```
int main() {  
    x=3;  
    y=0;  
    p1=Point(x, y);  
    p2=Point(6, 2);  
    line1=Line(p1, p2);  
    w=line1.p1.x # should assign 3 to w  
    line2=Line(Point(6, 2), Point(0, 2));  
    line3=Line(Point(0, 2), p1);  
    figList1=List();  
    figList1.push(line1);  
    figList1.push(line2);  
    figList1.push(line3);  
    fig1=Figure(figList1);  
    fig1.setColor(123);  
    return 0;  
}
```

Wyświetlony powinien zostać trójkąt o niebieskich krawędziach i współrzędnych ((3, 0), (6, 2), 0, 2))

# second program

```
int main() {  
    i=0;  
    x1=3;  
    y1=0;  
    x2=4;  
    y2=1;  
    linesList=List();  
    while i<=2 {  
        print("Smile!");  
        linesList.push(Line(Point(x1, y1),Point(x2, y2)));  
        x1=x2;  
        y1=y2;  
        x2=x2+2;  
        y2=y2+2;  
        i=i+1;  
    }  
    fig1=Figure(linesList);  
  
    # should raise an error above as the line segments do not form a closed figure  
  
    return 0;  
}
```

Użytkownik powinien otrzymać komunikat o błędzie, ponieważ zamknięta figura nie może zostać otrzymana z dostarczonych odcinków.

#### IV. Formalna specyfikacja i składnia

Niestandardowe typy danych:

List:

- Metody:
  - Konstruktor List()
  - Push(T x)
  - Pop()
  - Size()

Point:

- Pola:
  - Współrzędna int x
  - Współrzędna int y
- Metody:
  - Konstruktor Point(int x, int y)

Line:

- Pola:
  - Punkt pierwszy Point p1
  - Punkt drugi Point p2
- Metody:
  - Konstruktor Line(Point p1, Point p2)

Figure:

- Wewnętrznie będzie przechowywała listę współrzędnych x oraz listę współrzędnych y uporządkowanych w kolejności podanej przez użytkownika
- Pola:
  - Kolor figury int color
- Metody:
  - Konstruktor Figure(linesList myList) przyjmuje listę odcinków, z których ma zostać utworzona figura. Wszystkie przekazane odcinki rozbija na współrzędne i na bieżąco weryfikuje czy figura może zostać stworzona. To użytkownik musi zadbać o odpowiednią kolejność odcinków. Jeśli z przekazanej listy odcinków oraz konkretnej kolejności odcinków nie można utworzyć zamkniętej figury, program zgłosi błąd: ErrorNotAFigure
  - angles() – zwrócenie liczby kątów figury
  - setColor(int rgb) – ustawienie koloru figury

FigCollection:

- FigCollection zaimplementowane jako List(Figure)
- Lista figur będzie polem prywatnym i dostęp do niej będzie jedynie przez operacje push i pop na danym obiekcie FigCollection
- Metody:
  - Konstruktor FigCollection() – tworzy pustą kolekcję
  - push(Figure fig) – dodanie figury na koniec listy
  - pop() – usunięcie figury końca listy
  - size() – wielkość kolekcji – liczba dodanych figur
  - show() – wyświetlenie sceny z dodanymi do danej kolekcji figurami

Pola, do których użytkownik może się odwoływać poprzez kropkę '.':

- współrzędne punktu: point.x; point.y
- współrzędne odcinków: line.p1.x
- kolor figury: fig.color

Główny program, który użytkownik chce uruchomić powinien rozpoczynać się od int main() i być zakończony return 0. Po poprawnym zakończeniu programu (bez zgłoszenia błędów czy wyjątków) i wykorzystaniu metody show() na kolekcji figur, na ekranie wyświetli się okno z dodanymi figurami w odpowiednich kolorach (lub domyślnym jeśli nie zostały ustawione). W przypadku utworzenia więcej niż jednej kolekcji figur i wywołaniu na każdej show(), wyświetlonych zostanie odpowiednio więcej scen z figurami (po jednym oknie dla każdej kolekcji figur). Jeśli użytkownik wywoła show() na jednej kolekcji

figur kilkakrotnie, wyświetlona zostanie najbardziej aktualna wersja (bazujące na stanie kolekcji z ostatniego wywołania).

Słowa kluczowe oraz znaki specjalne:

- |                   |              |        |
|-------------------|--------------|--------|
| – 'int'           | – 'return'   | – '<=' |
| – 'double'        | – 'main'     | – '>'  |
| – 'string'        | – 'print'    | – '>=' |
| – 'bool'          | – 'function' | – '≡'  |
| – 'true'          | – '+'        | – '.'  |
| – 'false'         | – '-'        | – ','  |
| – 'List'          | – '*'        | – '""" |
| – 'Point'         | – '/'        | – ';'  |
| – 'Line'          | – '&&'       | – '('  |
| – 'Figure'        | – '  '       | – ')'  |
| – 'FigCollection' | – '!'        | – '{'  |
| – 'if'            | – '!='       | – '}'  |
| – 'else'          | – '=='       | – '#'  |
| – 'while'         | – '<'        |        |

Tokeny:

- |                     |                   |                   |
|---------------------|-------------------|-------------------|
| – T_AND             | – T_FIG_COLL      | – T_NOT           |
| – T_ASSIGN          | – T_FIGURE        | – T_NOT_EQ        |
| – T_COLON           | – T_GREATER       | – T_OR            |
| – T_COMMENT         | – T_GREATER_OR_EQ | – T_PLUS          |
| – T_CURLY_BRACKET_L | – T_IDENT         | – T_POINT         |
| – T_CURLY_BRACKET_R | – T_IF            | – T_PRINT         |
| – T_DIV             | – T_INT           | – T_REG_BRACKET_L |
| – T_DOT             | – T_LESS          | – T_REG_BRACKET_R |
| – T_DOUBLE          | – T_LESS_OR_EQ    | – T_RETURN        |
| – T_ELSE            | – T_LINE          | – T_SEMICOLON     |
| – T_EOF             | – T_LIST          | – T_STRING        |
| – T_EQUALS          | – T_MINUS         | – T_WHILE         |
|                     | – T_MULT          |                   |

Gramatyka:

EBNF

digit\_non\_zero = "1" | "2" | ... | "8" | "9";  
digit = "0" | digit\_non\_zero;  
natural\_nr = digit\_non\_zero, { digit };

```
integer      = "0", natural_nr;
double      = "0" | natural_nr, ".", digit, { digit } ;
number      = [ "-" ], ( integer | double );
letter      = "a" | ... | "z" | "A" | ... | "Z";
escape_char = "/"n" | ... | "/"t";
identifier  = letter, { letter | " " | digit };
```

```
bool          = "true" | "false";
additive_sign = "+", "-";
mult_sign     = "*", "/";
equal_sign    = "=", "!";
rel_sign      = "<", "<=", ">", ">=";
data_type     = "int" | "double" | "bool" | "string" | "Point" | "Line" | "List" | "Figure"
              | "FigCollection";
```

```
seq_data      = ( ( identifier | number ), { ( identifier | number ), "," } ) | "";
comment       = "#", { letter | escape_char }, "/n";
string        = " " ", { letter | escape_char }, " " " ";
```

```
parent_cond = "(" , condition , ")";
condition  = and_cond , { "|" , and_cond };
and_cond   = equal_cond , { "&&" , equal_cond };
equal_cond = rel_cond , { equal_sign , rel_cond };
rel_cond   = prim_cond , { rel_sign , prim_cond };
prim_cond  = "[" ! "]" , ( parent_cond | expr | bool );
```

```
parent_expr = "(" , expr , ")";
expr        = mult_expr , { additive_sign , mult_expr } ;
mult_expr   = prim_expr , { mult_sign , prim_expr } ;
prim_expr   = identifier | number | func_call | parent_expr;
func_call   = identifier , "(" , seq_data , ")" , ";" ;
```

block	= "{", { stmt }, "}"
stmt	= if_stmt   while_stmt   return_stmt   print_stmt   assign_stmt   (func_call, ";")   block;
if_stmt	= "if", "(", condition, ")", block, ["else", block];
while_stmt	= "while", "(", condition, ")", block;
return_stmt	= "return", expr, ";";
assign_stmt	= { data_type }, identifier, "=", ( expr   string ), ";";
func_def	= "function", "(", seq_data, ")", block;
program	= { func_def };

## Obsługa błędów

W przypadku wykrycia błędu, bieżąca pozycja w tekście będzie zapamiętywana. Dla krytycznych błędów program będzie przerywany i użytkownik otrzyma informację z numerem linii (i kolumny), która spowodowała błąd wraz z jej zawartością oraz doprecyzowaną wiadomością, jaka była przyczyna błędu.

Przykładowe typy błędów:

- Dzielenie przez 0
- Konkatenacja typu liczbowego oraz stringa
- Próba utworzenia figury, która nie będzie spójna (nie będzie się domykała)
- Dodanie do obiektu List innego typu danych niż już znajdujący się wewnątrz
- Wywołanie niezdefiniowanej funkcji lub metody na obiekcie
- Wykroczenie poza zakres dopuszczalnych liczb
- Wykroczenie poza max długością słowa lub komentarza

## V. Sposób uruchomienia, wej/wyj

Do zbudowania projektu wykorzystane zostanie narzędzie Maven, a samo uruchomienie programu będzie korzystało z polecenia `java -jar ...`

Jako wejście musi zostać przekazana ścieżka do pliku z programem, który chcemy uruchomić (plik z rozszerzeniem `txt`).

## VI. Sposób realizacji

Język programowania – Java z wykorzystaniem narzędzia Maven.

GUI:

- Wyświetlane użytkownikowi okno zostanie zrealizowane z wykorzystaniem biblioteki Swing (głównie klasy `JFrame`), a figury będą dodawane do panelu `JPanel`, który następnie zostanie przekazany do głównej ramki
- Do rysowania figur wykorzystana zostanie biblioteka AWT, z klasą `Graphics` i `Graphics2D` i metodami takimi jak `drawPolygon`, który za argumenty przyjmuje listy współrzędnych `x` oraz `y`

## VII. Sposób testowania

Do testowania wykorzystana zostanie biblioteka `JUnit`. Testami zostaną objęte wszystkie komponenty projektu: lekser, parser oraz interpreter.

Testowanie leksera:

- Wykrycie wszystkich dostępnych typów tokenów
- Zgłoszenie błędu przy otrzymaniu symbolu/konstrukcji nierozpoznawalnej przez język np. `'%` w kodzie programu czy niekończącego się komentarza

Przykłady:

asypula

@Test

```
public void test_T_OR() throws IOException, InvalidTokenException {
    Token tokenExp=new Token(TokenType.T_OR, val: "|", new Position( row: 0, col: 0));
    String x = "|";
    initLexer(x);
    Token t = myLexer.getToken();
    assertToken(tokenExp, t);
}
```

asypula

@Test

```
public void testException_unknownChar() throws IOException {
    Position pos = new Position( row: 0, col: 0);
    String x = "%";
    initLexer(x);
    Exception exception = assertThrows(InvalidTokenException.class, () -> myLexer.getToken());
    String expectedMessage = "Invalid token " + x + " at the position: " + pos;
    String actualMessage = exception.getMessage();
    assertTrue(actualMessage.contains(expectedMessage));
}
```

new \*

@Test

```
public void test_Sequence1() throws IOException, InvalidTokenException {
    ArrayList<Token> expectedTokens = new ArrayList<Token>();
    expectedTokens.add(new Token(TokenType.T_WHILE, val: "while", new Position( row: 0, col: 0)));
    expectedTokens.add(new Token(TokenType.T_REG_BRACKET_L, val: "(", new Position( row: 5, col: 0)));
    expectedTokens.add(new Token(TokenType.T_IDENT, val: "i", new Position( row: 6, col: 0)));
    expectedTokens.add(new Token(TokenType.T_LESS, val: "<", new Position( row: 7, col: 0)));
    expectedTokens.add(new Token(TokenType.T_INT, val: "20", new Position( row: 8, col: 0)));
    expectedTokens.add(new Token(TokenType.T_REG_BRACKET_R, val: ")", new Position( row: 10, col: 0)));
    expectedTokens.add(new Token(TokenType.T_PRINT, val: "print", new Position( row: 0, col: 1)));
    expectedTokens.add(new Token(TokenType.T_REG_BRACKET_L, val: "(", new Position( row: 5, col: 1)));
    expectedTokens.add(new Token(TokenType.T_STRING, val: "Hello", new Position( row: 6, col: 1)));
    expectedTokens.add(new Token(TokenType.T_REG_BRACKET_R, val: ")", new Position( row: 13, col: 1)));
    expectedTokens.add(new Token(TokenType.T_SEMICOLON, val: ";", new Position( row: 14, col: 1)));
    String x = "while (i < 20)\n print(\"Hello\");";
    ArrayList<Token> returnedTokens = new ArrayList<Token>();
    initLexer(x);
    while (myLexer.isRunning()) {
        Token newToken = myLexer.getToken();
        returnedTokens.add(newToken);
    }
    for (int i = 0; i<returnedTokens.size(); i++)
        assertToken(expectedTokens.get(i), returnedTokens.get(i));
    Token newToken = myLexer.getToken();
    assertToken(newToken, new Token(TokenType.T_EOF, val: "EOF", new Position( row: 15, col: 1)));
}
```



Testowanie parsera:

- Sprawdzenie odpowiedniego rozpoznawania wszystkich symboli terminalnych i nieterminalnych
- Zgłoszenie błędu w przypadku wykrycia nieznanej produkcji np. sekwencji tokenów: T\_POINT, T\_CURLY\_BRACKET\_L

Testowanie interpretera:

- Krótkie przykładowe testowe „programy”, które weryfikowałyby poprawność całego potoku przetwarzania – na koniec testowały wyniki np. utworzone figury czy zawartość utworzonej kolekcji figur – sceny