

TKOM Dokumentacja projektu

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
 - Przekazanie listy punktów
 - Ustawienie figurze koloru przez wartości RGB
- Wyświetlanie sceny figur odbywa się poprzez stworzenie listy z figurami, które chcemy wyświetlić i na koniec wywołanie metody `show()`

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.

Wymagane jest zdefiniowanie funkcji 'main', od którego wykonania program się rozpoczyna.

Obsługa błędów

III. Przykłady języka

Przykłady podstawowych instrukcji:

- if (condition) {} else {}
- while (condition) {}
- function identifier (params) {}
- identifier = List();
- list.add(Point(x, y));

example program, this comment should be ignored

```
function main() {
```

```
    x=30;
```

```
    y=0;
```

```
    p1=Point(x, y);
```

```
    p2=Point(60, 20);
```

```
    line1=Line(p1, p2);
```

```

        line2=Line(Point(60, 20), Point(0, 20));
        line3=Line(Point(0, 20), p1);
        figList1=List();
        figList1.add(line1);
        figList1.add(line2);
        figList1.add(line3);
        fig1=Figure(figList1);
        fig1.color(123, 123, 123);
    }

```

Wyświetlony powinien zostać trójkąt o szarych krawędziach i współrzędnych ((30, 0), (60, 20), 0, 20))

second program

```

function main() {
    i=0;
    x1=300;
    y1=0;
    x2=400;
    y2=100;
    linesList=List();
    while (i<=2) {
        print("Smile!");
        linesList.add(Line(Point(x1, y1),Point(x2, y2)));
        x1=x2;
        y1=y2;
        x2=x2+20;
        y2=y2+20;
        i=i+1;
    }
    fig1=Figure(linesList);

    # should raise an error above as the line segments do not form a closed figure
}

```

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

Więcej przykładów w pliku test.txt, a także w testach interpretera.

IV. Formalna specyfikacja i składnia

Niestandardowe typy danych:

List:

- Metody:
 - Konstruktor List()
 - Add (Value v)
 - Remove ()
 - Show (), dostępne jedynie dla list wypełnionych figurami

Point:

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

Line:

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

Figure:

- Wewnętrznie będzie przechowywała listę punktów ze współrzędnymi x oraz y uporządkowanych w kolejności podanej przez użytkownika
- Pola:
 - Lista punktów
 - Wartości całkowite dla poszczególnych składowych kolorów: RGB
- Metody:
 - Konstruktor Figure(List myList) może przyjmować listy z dwoma różnymi zawartościami:
 - 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: IncorrectFigureException
 - listę punktów, z których ma zostać zbudowana figura. W tym przypadku „zamkniętość” nie musi być weryfikowana – domyślnie ostatni punkt zostanie połączony z pierwszym
 - Dodatkowo sprawdzane jest zawsze, czy dostarczona została odpowiednia liczba linii/punktów do zbudowania figury

- `color(int r, int g, int, b)` – ustawienie koloru figury

Główny program, który użytkownik chce uruchomić powinien rozpoczynać się od function `main()`. Po poprawnym zakończeniu programu (bez zgłoszenia błędów czy wyjątków) i wykorzystaniu metody `show()` na liście 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:

– <code>'true'</code>	– <code>'/'</code>	– <code>'='</code>
– <code>'false'</code>	– <code>'&&'</code>	– <code>'.'</code>
– <code>'if'</code>	– <code>' '</code>	– <code>';</code>
– <code>'else'</code>	– <code>'!'</code>	– <code>'''</code>
– <code>'while'</code>	– <code>'!='</code>	– <code>','</code>
– <code>'return'</code>	– <code>'=='</code>	– <code>'('</code>
– <code>'function'</code>	– <code>'<'</code>	– <code>)'</code>
– <code>'+'</code>	– <code>'<='</code>	– <code>'{'</code>
– <code>'_'</code>	– <code>'>'</code>	– <code>'}'</code>
– <code>'*'</code>	– <code>'>='</code>	– <code>'#'</code>

Tokeny:

– <code>T_AND</code>	– <code>T_FALSE</code>	– <code>T_NOT_EQ</code>
– <code>T_ASSIGN</code>	– <code>T_FUNCTION</code>	– <code>T_OR</code>
– <code>T_COLON</code>	– <code>T_GREATER</code>	– <code>T_PLUS</code>
– <code>T_COMMENT</code>	– <code>T_GREATER_OR_EQ</code>	– <code>T_REG_BRACKET_L</code>
– <code>T_CURLY_BRACKET_L</code>	– <code>T_IDENT</code>	– <code>T_REG_BRACKET_R</code>
– <code>T_CURLY_BRACKET_R</code>	– <code>T_IF</code>	– <code>T_RETURN</code>
– <code>T_DIV</code>	– <code>T_INT</code>	– <code>T_SEMICOLON</code>
– <code>T_DOT</code>	– <code>T_LESS</code>	– <code>T_STRING</code>
– <code>T_DOUBLE</code>	– <code>T_LESS_OR_EQ</code>	– <code>T_TRUE</code>
– <code>T_ELSE</code>	– <code>T_LIST</code>	– <code>T_WHILE</code>
– <code>T_EOF</code>	– <code>T_MINUS</code>	
– <code>T_EQUALS</code>	– <code>T_MULT</code>	
	– <code>T_NOT</code>	

Klasy wbudowane:

- Point
- Line
- List
- Figure

Gramatyka:

EBNF

program	= { func_def };
func_def	= "function", identifier, "(", [params], ")", block;
block	= "{", { stmt }, "}"
stmt	= if_stmt while_stmt return_stmt ident_start_stmt;
if_stmt	= "if", "(", expr, ")", block, ["else", block];
while_stmt	= "while", "(", expr, ")", block;
return_stmt	= "return", expr, ";";
assign_stmt	= "=", expr;
ident_start_stmt	= identifier, { assign_stmt rest_func_call rest_obj_access };
rest_func_call	= '(', [args], ')', ';' ;
rest_obj_access	= '.', identifier, [rest_func_call], { '.', identifier, [rest_func_call] } ;
expr	= and_expr, { " ", and_expr };
and_expr	= rel_expr, { "&&", rel_expr };
rel_expr	= arithm_expr, { rel_operator, arithm_expr };
arithm_expr	= mult_expr, { ("+" "-") mult_expr };
mult_expr	= prim_expr, { ("*" "/") prim_expr };
prim_expr	= [negation], (literal ident_start_stmt "(", expr, ")");
params	= identifier, { ",", identifier };
args	= expr, { ",", expr };
identifier	= letter, { letter "_" digit };
literal	= bool integer double string identifier;
string	= " " " ", { letter escape_char digit }, " " " " ;
bool	= "true" "false";
letter	= "a" ... "z" "A" ... "Z";
escape_char	= "/n" ... "/t";
integer	= "0", natural_nr;
double	= "0" natural_nr, ".", digit , { digit } ;
natural_nr	= digit_non_zero, { digit };
digit	= "0" digit_non_zero;
digit_non_zero	= "1" "2" ... "8" "9";

rel_operator = "==" | "!=" | "<" | "<=" | ">" | ">=";

Obsługa błędów, typy zgłaszanych wyjątków

- DuplicatedElementException: przy zdefiniowaniu dwóch funkcji o takiej samej nazwie lub zdefiniowania funkcji z takimi samymi nazwami dwóch parametrów
- ExceededLimitsException: przy próbie dostępu do elementu listy, który wykracza poza jej rozmiar
- IncorrectFigureException: przy błędach związanych z figurą np. próbie stworzenia figury z nieodpowiednich obiektów (nie listy z punktami lub liniami) czy ustawienia koloru, nie poprzez podanie wartości RGB
- IncorrectTypeException: przy błędach związanych z typem danych np. wywołanie działania mnożenia na stringach czy stworzenia obiektu klasy Point z niecałkowitych wartości
- IncorrectValueException: przy dostarczeniu do wyrażenia, które oczekuje wartości końcowej true albo false (np. if) wyrażenia/zmiennnej, która nie może być do tego ewaluowana
- InvalidMethodException: wywołanie nieodpowiedniej metody na obiekcie np. metody, która dla danego obiektu nie istnieje
- InvalidTokenException: przy napotkaniu na niepoprawny – nieistniejący w języku ciąg tokenów
- MissingPartException: brak wymaganego elementu np. nawiasu zamykającego, argumentu w wywołaniu funkcji, deklaracji 'function'
- OverflowException: przy próbie dodania, odjęcia, mnożenia lub dzielenia, które spowodowałyby wykroczenia poza dopuszczalny zakres wartości dla danego typu
- UnknownVariableException: przy próbie odniesienia do niezadeklarowanej i niezdefiniowanej zmiennej
- ZeroDivisionException: przy próbie dzielenia przez 0

V. Sposób uruchomienia, wej/wyj

Do zbudowania projektu wykorzystane zostanie narzędzie Maven, a przy uruchamianiu programu należy podać jako argument ścieżkę do pliku z naszym własnym programem.

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

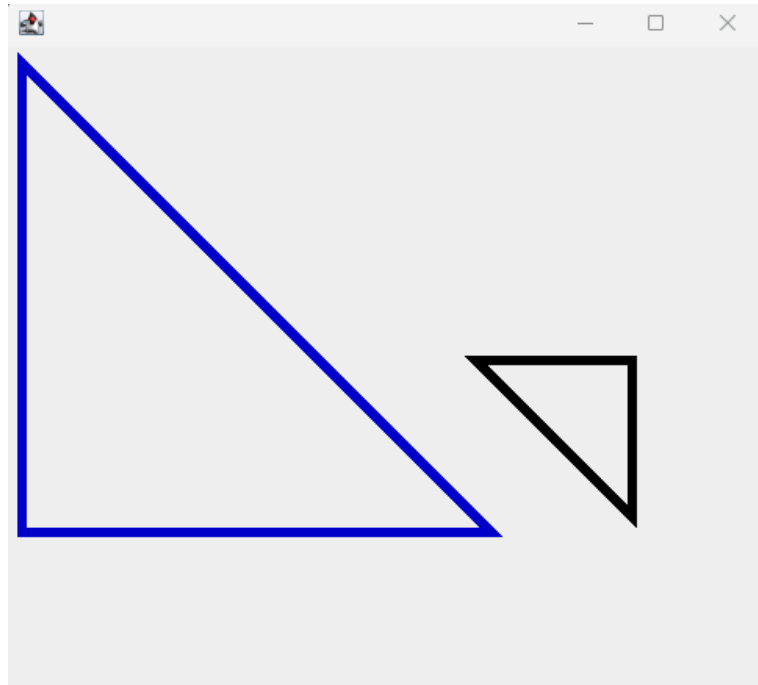
Testowanie interpretera:

- Podstawowe funkcjonalności takie jak dodawanie, instrukcje warunkowe, odpowiednia kolejność przetwarzania zagnieżdżonych operacji czy wywołanie zdefiniowanych przez użytkownika funkcji
- Poprawne zgłaszanie wyjątków np. przy dzieleniu przez 0, niepoprawnego tworzenia obiektów, czy wykorzystywania odpowiednich typów danych z odpowiednimi metodami
- Nietrywialnie zachowania np. odpowiednie wychodzenie z funkcji po wcześniejszym return'ie, nieutrzymywanie w rezultatach wartości zwróconej z wywołania funkcji, jeśli nie została ona do żadnej zmiennej przypisana
- Krótkie przykładowe testowe „programy”, które weryfikowałyby poprawność całego potoku przetwarzania – tworzenia punktów, linii, figur a następnie wyświetlania całej sceny z figurami

Przykład z wyświetlaniem sceny:

```
function main(){
    test();
}

function test(){
    i = 0;
    x = 10;
    y = 10;
    list = List();
    while (i<4){
        list.add(Point(x, y));
        x = x+100;
        y = y+100;
        i = i+1;
        print(x);
        print(y);
        print(i);
    }
    list.add(Point(10, y-100));
    fig=Figure(list);
    fig.color(0, 0, 200);
    figList = List();
    figList.add(fig);
    list2 = List();
    list2.add(Point(300, 200));
    list2.add(Point(400, 300));
    list2.add(Point(400, 200));
    fig2 = Figure(list2);
    figList.add(fig2);
    figList.show();
}
```



```
function diamond(){
    list = List();
    list.add(Point(200, 200));
    list.add(Point(220, 185));
    list.add(Point(230, 185));
    list.add(Point(250, 200));
    list.add(Point(225, 240));
    fig = Figure(list);
    figList = List();
    fig.color(169, 169, 13*13);
    figList.add(fig);
    figList.show();
}
```

