TKOM - etap 2

Hubert Truszewski

Temat: 4 - język opisu brył i ich właściwości. Wersja ze statycznym i silnym typowaniem.

Opis funkcjonalności

W ramach języka będą wspierane różne typy brył i możliwe będzie ich wyświetlenie na ekranie. Dostępne bryły:

- prostopadłościan
- ostrosłup prawidłowy trójkątny
- stożek
- walec
- kula

Dla prostopadłościanu, ostrosłupa, stożka oraz walca dostępne są metody do obliczenia:

- objętości
- pola podstawy
- pola powierzchni bocznej
- pola powierzchni całkowitej

Dla kuli dostępne bedą metody do obliczenia średnicy, objętości oraz pola powierchni całkowitej.

Dostępne typy primitywne: int, double, bool, string.

Przykładowy kod

```
void showFigures(List<SolidFigure> figuresList) {
    Screen screen = Screen;
    int counter = 0;
    while (counter < figuresList.length) {
        screen.add(figuresList.next());
    }
    screen.show();
}

int totalVolumes(List<SolidFigure> figuresList) {
    int volumes = 0;
    while (counter < figuresList.length) {
        counter += figuresList.next().volume();
    }
    return volumes;
}

void main() {
    List<SolidFigure> figureList;
```

```
    Sphere s = Sphere(5);
    figureList.add(s);
    Cone c = Cone(2, 5);
    figureList.add(c);
    showFigures(figureList);
}
```

Operatory

Precedencja

1 = najwyższy priorytet

Priorytet	Operator	Łączność
1	Dostęp do obiektu (.)	lewostronny
2	Negacja (!)	prawostronny
3	Mnożenie oraz dzielenie (*, /)	lewostronny
4	Dodawanie oraz odejmowanie (+, -)	lewostronny
5	Porównanie (==, !=, >, <, >=, <=>)	lewostronny
6	operacja AND (&&)	lewostronny
7	operacja OR (||)	lewostronny

Opis działania

Operatory +, -, *, / działają w przypadku działań na typach primitywnych. Operatory porównania działają dla brył i wykorzystują wartość objętości dla porównań, ponieważ jest to jedyna wspólna cecha wszystkich typów.

Komentarze

- jednolinikowe np. `// jakis tekst`

Obsługa błędów

Jeżeli w czasie działania zostanie napotkany błąd, np. niedomnknięty nawias zostanie rzucony wyjątek z odpowiednim komunikatem. Kod:

```
    if (a {
        int a = 5;
    }
```

Przykład:

```
Error in line 4, position 10: missing right parenthesis
```

Podział na komponenty:

- aplikacja - odpowiada za otworzenie aplikacji oraz załadowanie pozostałych komponentów i ich zależności
- input - wejście kodu - z pliku lub ze standardowego wejścia
- lekser - odpowiada za analizę leksykalną, generuje tokeny
- parser - odpowiada za analizę składniową
- interpreter - wykonuje faktyczny kod programu, odpowiada za przechwycenie zgłaszanych wyjątków

Testowanie

Testować zamierzam głównie pisząc testy jednostkowe przy użyciu `JUnit`. Każdy z komponentów będzie posiadał osobny zestaw testów.

Składnia w formacie EBNF:

```
program                   = {function_declaration};
function_declaration      = type, identifier, "(", {type, identifier},
")", code_block;
code_block                = "{", statement, "}";
statement                 = conditional_statement
                          | varibale_declaration
                          | assignment
                          | return_statement
                          | function_call;
conditional_statement     = if_statement | while_loop;
if_statement              = "if (", expression, ")", code_block, ["else",
code_block];
while_loop                = "while (", expression, ")", code_block;
variable_declaration      = type, identifier;
assignment                = identifier, "=", expression, ";";
return_statement          = "return", expression, ";";
function_call             = identifier, "(", {type, expression}, ")", ";",
;
expression                = number | identifier | function_call |
string_literal | or_expression;
or_expression             = and_expression, {or_operator, and_expression};
and_expression            = coparison_expression, {and_operator,
coparison_expression};
coparison_expression      = addition_expression, {comparison_opearator,
addition_expression};
addition_expression       = multiplication_expression, {addition_operator,
multiplication_expression};
multiplication_expression = negation_expression, {multiplication_operator,
negation_expression};
negation_expression       = [negation_operator], access_expression;
access_expression         = identifier, {access_operator, identifier, "(",
```

```
[simple_expression | identifier], ")"};
simple_expression          = number | string_literal;
type                       = "int"
                           | "string"
                           | "double"
                           | "bool"
                           | "void"
                           | "Cone"
                           | "Cylinder"
                           | "Sphere"
                           | "Cuboid"
                           | "Pyramid";
identifier                 = letter, {letter | digit};
string_literal             = '"', string_element, {string_element}, '"';
string_element             = letter | escape_character;
letter                     = "A" | "B" | ... | "Z" | "a" | "b" | ... | "z";
escape_character           = "\";
number                     = digit_non_zero, {digit};
digit                      = "0" |
digit_non_zero             = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
| "9" |;

addition_operator          = "+" | "-";
multiplication_operator    = "*" | "/";
negation_operator          = "!";
access_operator            = ".";
comparison_operator        = "==", "!=", ">", "<", ">=", "<=";
and_operator               = "&&";
or_operator                = "||";
```