TKOM – Dokumentacja końcowa

Język programowania z wbudowanym typem liczb zespolonych

Illia Yatskevich

Cel projektu:

Celem projektu jest stworzenie języka programowania pozwalającego na wykonywanie operacji na typach standardowych i na liczbach zespolonych.

Opis ogólny:

Projekt został napisany w języku Java. Zmienna typu liczby zespolonej ma atrybuty (część rzeczywista i urojona) po których można sięgać. Interpreter można rozszerzać o funkcje biblioteczne (np. do wypisywania na ekran, wczytywania z konsoli).

Język zawiera takie słowa kluczowe jak: function, return, if, else, for, while, import.

Język oferuje 4 typy danych: liczby całkowite, liczby ułamkowe, string i liczby zespolone. Liczba zespolona deklaruje się przy pomocy słowa kluczowego Complex i dwóch argumentów: pierwszy to część rzeczywista, druga – urojona; możliwe jest sięganie do części rzeczywistej i urojonej (np. c.real i c.imag) oraz wykonywanie operacji dodawania, odejmowania, mnożenia, dzielenia.

Wbudowanymi funkcjami są:

- modulus() – moduł liczby zespolonej
- conjugate() – liczba zespolona sprzężona
- pow() – potęgowania liczb

Funkcję które można dorzucić z zewnątrz:

- print() – wypisywanie na ekran
- scan() – wczytywanie z klawiatury

Pola składowe typu Complex:

- real (część rzeczywista)
- imag(część urojona)

Dostępna tylko relacja równości(==) i nierówności(!=), ponieważ nie istnieje dokładnego matematycznego sposobu na porównanie liczb zespolonych. Można tylko stwierdzić, że liczby zespolone są równe, gdy ich części rzeczywiste są równe i części urojone też są równe.

Przykładowy kod źródłowy:

```
import “hello.txt”
```

```
function MyFunction(a, b, c)
{
	if(a > 2 && b < 10)
	{
		c.real = c.real + a;
		c.imag = c.imag + b;
	}

	counter = 0;

	while(counter < 5)
	{
		c = c*3;
	}
	d = Complex(1,1);
	return c+d;
}

function main()
{
	a = 15;
	b = 5.4;
	c = Complex(10, 12.3);
	d = “Hello world!\n”;
	print(d);
	e = MyFunction(a,b,c);
	f = scan();
	string = d + “ ” + c.imag + “+” + a;
	for(i=0; i < modulus(c); i=i+1)
	{
		c = pow(c,2);
		print(conjugate(c));
	}
	return a;
}
```

Gramatyka(EBNF):

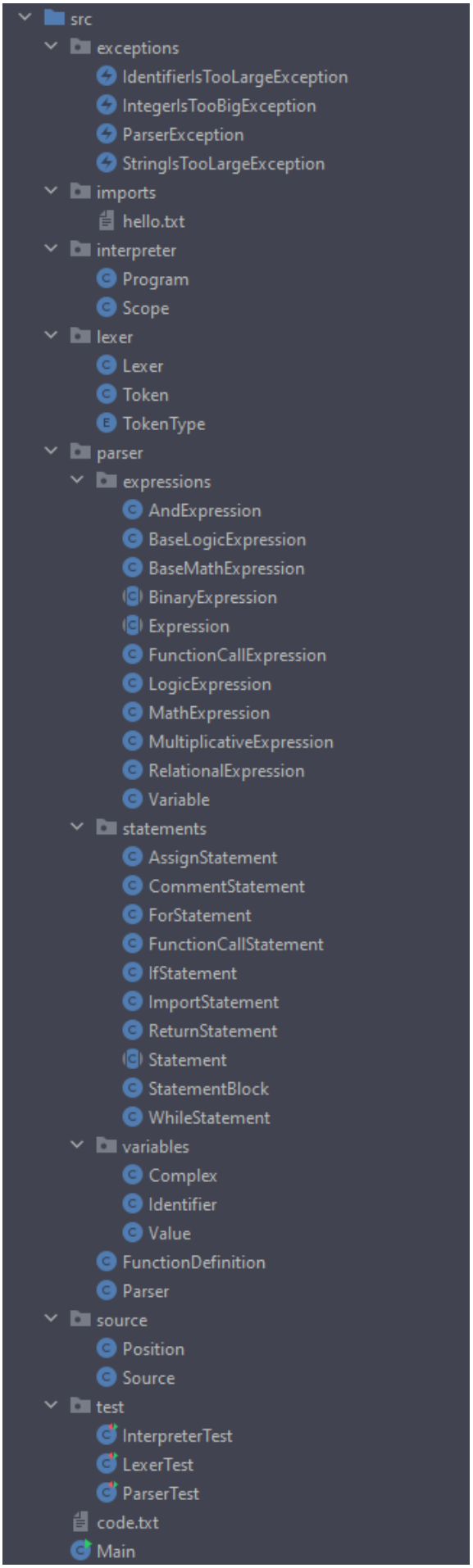
```
program = { functionDef | importStatement } ;
importStatement = "import", stringLiteral;
functionDef = "function", identifier, "(", parameters , ")",
statementBlock ;
parameters =  [ identifier { "," identifier } ] ;

statementBlock = "{", { assignOrFunctionCall | returnStatement |
ifStatement | whileStatement | forStatement, ";"}, "}" ;
```

```
assignOrFunctionCall = identifier,(restAssignStatement |
restFunctionCall) ;
returnStatement = "return", logicExpression, ";" ;
restAssignStatement = restVariable, "=", logicExpression ;
ifStatement = "if", "(", logicExpression, ")", statementBlock, [
"else", statementBlock ] ;
whileStatement = "while", "(", logicExpression, ")", statementBlock ;
forStatement = “for”, “(”, assignStatement, “;”, logicExpression, “;”,
numberLiteral, “)”, statementBlock ;
restFunctionCall = "(", arguments, ")" ;
arguments =  [ logicExpression { "," logicExpression } ] ;

logicExpression = andExpression, { orOperator, andExpression } ;
andExpression = relationalExpression, { andOperator,
relationalExpression };
relationalExpression = baseLogicExpression, [ relationOperator
baseLogicExpression ] ;
baseLogicExpression = [ unaryLogicOperator ], mathExpression ;

mathExpression = multiplicativeExpression, { additiveOperator,
multiplicativeExpression } ;
multiplicativeExpression = baseMathExpression,{
multiplicativeOperator, baseMathExpression } ;
baseMathExpression = [unaryMathOperator ], (value |
parentLogicExpression) ;

parentLogicExpression = "(", logicExpression, ")" ;
value = numberLiteral | complexLiteral | floatLiteral | stringLiteral
| identifier,( restVariable | restFunctionCall ) ;

unaryMathOperator = "-" ;
unaryLogicOperator = "!" ;
additiveOperator = "+" | "-" ;
multiplicativeOperator = "*" | "/" ;
orOperator = "||" ;
andOperator = "&&" ;
relationOperator = "==" | "<" | ">" | "<=" | ">="| "!="  ;
commentary = “//”

restVariable = [ ".", complexAttr ];
stringLiteral = '"', { allCharacters }, '"' ;
complexLiteral = “Complex”, “(”, numberLiteral | floatLiteral,
numberLiteral | floatLiteral, “)” ;
numberLiteral = digitNonZero { digit } ;
floatLiteral = numberLiteral, ".", {digit};
complexAttr = "real" | "imag" ;
identifier = letter | specialElement { letter | digit | specialElement
} ;

specialElement = "_" ;
letter = upper | lower ;
```

```
upper = "A" | "B" | "C" | "D" | "E" | "F" | "G"
       | "H" | "I" | "J" | "K" | "L" | "M" | "N"
       | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
       | "V" | "W" | "X" | "Y" | "Z";
lower = "a" | "b"
       | "c" | "d" | "e" | "f" | "g" | "h" | "i"
       | "j" | "k" | "l" | "m" | "n" | "o" | "p"
       | "q" | "r" | "s" | "t" | "u" | "v" | "w"
       | "x" | "y" | "z";
digitNonZero = “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” |"9";
digit = “0” | digitNonZero;
allCharacters = ? all visible characters ? ;
```

Struktura plików:

Struktura plików przedstawiona na rysunku po prawej stronie.



Obiekty i klasy:

Interpreter:

Folder Interpreter zawiera klasy Program i Scope, klasa Program zawiera listę importów i funkcji oraz ma metodę run, od której zaczyna się interpretacja.

Lexer:

Folder Lexer zawiera klasy Lexer, Token i enumerację TokenType.
Klasa Lexer jest odpowiedzialna za przetwarzanie źródła danych i wyprodukowanie obiektów Token. Obiekt tej klasy dostaje od obiektu klasy Source symbole i próbuje je dopasować do znanych typów tokenów.
TokenType trzyma w sobie wszystkie typy tokenów rozumiane przez Lexer.

Parser:

W tym folderze znajdują się pakiety expression, statements, variables, a także klasa FunctionDefiniton. Te wszystkie klasy reprezentują gramatykę języka. Każdy Statement ma metodę execute() i każdy expression ma metodę evaluate() które są potrzebne dla interpretacji programu.
Zadaniem klasy Parser jest przetwarzanie obiektów otrzymanych od Lexera i stworzenie drzewa wyprowadzenia. Parser parsuje obiekty kierując się gramatyką języka.

Source:

W folderze source znajdują się klasy Source i Position. Zadaniem klasy Source jest przygotowanie źródła i pobieranie z niego po jednym symbolu, a także śledzenie aktualnie przetwarzanej pozycji w źródle.

Main:

Klasa main zawiera metodę main, od której zaczyna się wykonanie programu.

Sposób uruchomienia, we/wy:

Program jest uruchomiany z konsoli. Na wejściu trzeba podać nazwę pliku z kodem źródłowym. Wynik poszczególych kroków analizy oraz wynik uruchomienia programu będzie wyświetlony na standardowym wyjściu.

Obsługa błędów:

Moduł obsługi błędów tłumaczy błędy z poszczególnych modułów na postać czytelną dla

człowieka. IdentifierIsToolargeException, StringIsTooLargeException i IntegerIsTooBigException są to wyjątki lexera, rzucane jeżeli odpowiednio identyfikator, string albo liczba jest za długa. ParserException służy dla błędów parsowania, dla błędów interpretera została wykorzystana klasa Exception.

Sposób testowania:

W celu sprawdzenia poprawności działania zostałe napisane testy jednostkowe prze użyciu biblioteki JUnit dla trzech głównych części projektu: leksera, parsera i interpretera. Przetestowane zostały różne aspekty każdej części.