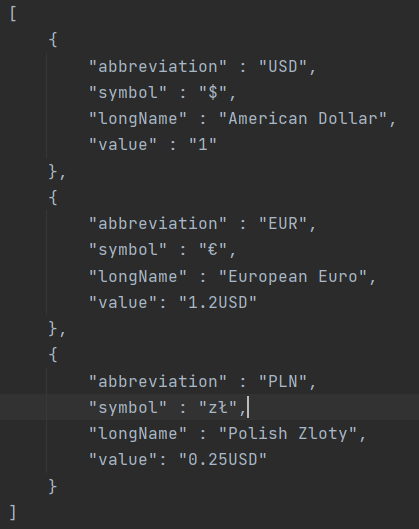
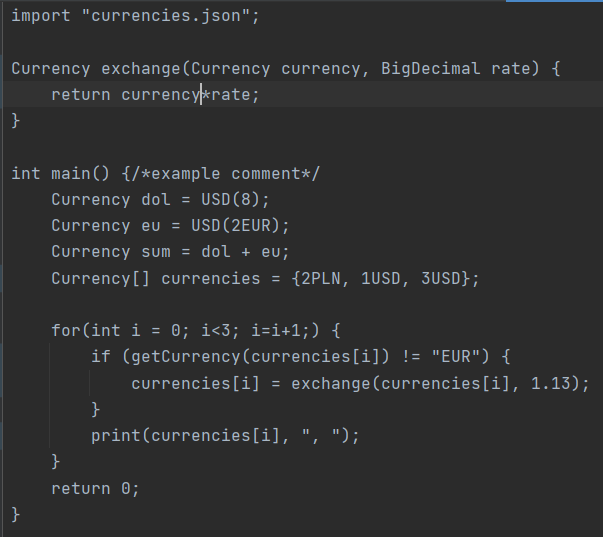
1. ***Opis funcjonalny***

Język do operacji na walutach powinien, jak wskazuje sama nazwa, dać użytkownikowi możliwość w łatwy, nie wymagający sposób przeprowadzać operacje matematyczne dotyczące pieniędzy w jednej i przede wszystkim w różnych jednostkach (sumowanie, odejmowanie, dzielenie). Również ma na celu umożliwić operacje logiczne (takie jak porównanie) oraz konwertowanie z podanej jednostki do pożądanej.

Plik definiujący waluty będzie mieć postać tablicy w formacie JSON i będzie importowany do kodu walutowego:



Kod walutowy wygląda następująco (zawiera instrukcję warunkową, definicję funkcji, instrukcję pętli, komentarze):



Inne przykłady konstrukcji, które umożliwia język:

- deklaracja i definicja zmiennych typu boolean, String, int, BigDecimal, Currency i tablic tych typów:

boolean found;

found = false;

String s = ”Hello World!”;

int[] numbers = {1, 3, 8, 16};

- zagnieżdżona pętla if, zawierająca zagnieżdżone warunki logiczne:

if (a < b && (b < c && c !=0)) {

if (a < d) {

a=d;

}

} else {

a = 0;

}

- operacje matematyczne z pilnowaniem priorytetów operatorów:

int mandate = sum + rate\*3 – basic;

Najpierw zostanie wykonane mnożenie, następnie suma i w ostatniej kolejnoście odejmowanie od wyniku basic.

**2. Formalny opis gramatyki** (w EBNF):

**Program** = {Import | FuncDeclaration}, MainFunction;

**MainFunction** = ‘int’, ‘main’, ‘(‘, ‘)’,’{‘, { Instruction }, ‘}’;

**Import** = ‘import’, ConstString, ‘;’;

**FuncDeclaration** = ReturnType, Identifier, ‘(‘, [ArgDeclaration, {‘,’, ArgDeclaration },], ‘)’, ‘{‘, { Instruction }, ‘}’;

**Instruction** = (FunctionCall, ‘;’) | VarDeclaration | VarDefinition | ForStatement | IfStatement | ReturnExpression**;**

?? null

**ReturnExpression =** ‘return’, ( VarValueArray | Expression), „;”;

**VarDeclaration** = Type, Identifier, [‘=‘, VarValueArray | Expression], ‘;’;

**ArgDeclaration** = Type, Identifier, [‘=‘, ArgValue];

**VarValueArray =** ‘{‘, [VariableValue, {‘,’, VariableValue}], ‘}’;

**VariableValue** = ArrayElementReference | ArgValue | FunctionCall;

**ArgValue** = Number | ConstString | ConstBoolean | BigDecimalNumber | CurrencyConversion | CurrencyAssignment;

**FunctionCall** = Identifier, ‘(‘, [VarValueArray | ComplexExpression, {‘,’, VarValueArray | ComplexExpression }], ‘)’;

**VarDefinition** = ArrayElementReference, ‘=‘, (VarValueArray | ComplexExpression), ‘;’;

**CurrencyConversion =** CurrencyAbb, ‘(‘, Expression,‘)’;

**CurrencyAssignment =** (Number | BigDecimalNumber), CurrencyAbb;

**CurrencyAbb** = CapitalLetter, CapitalLetter, CapitalLetter;

**IfStatement** = ‘if’, ‘(‘, Condition, ‘)’, ‘{‘, { Instruction }, ‘}’, [‘else’, (ifStatement | ‘{‘, { Instruction }, ‘}’)];

**ForStatement** = ‘for’, ‘(‘, ForExpression, ‘)’ ‘{‘, { Instruction }, ‘}’;

**ForExpression** = VarDeclaration, ComplexExpression, ‘;’, VarDefinition;

**Condition** = [‘!’], IfExpression, {‘||’, Condition};

**IfExpression** = (SimpleExpression | ComplexExpression), {‘&&’, IfExpression };

**ComplexExpression =** SimpleExpression RelOperator SimpleExpression;

**SimpleExpression** = [‘!’], (Expression | FunctionCall);

**Expression** = Term, {(‘+’ | ‘-’), Expression};

**Term** = Factor, {(‘\*’ | ‘/’), Term };

**Factor** = VariableValue | ‘(‘, Expression, ‘)’;

**ReturnType** = Type | ‘void’;

**Type** = SimpleType | ArrayType;

**ArrayType** = SimpleType, ‘[‘, [Number] ‘]’;

**SimpleType** = ‘int’ | ‘boolean’ | ‘BigDecimal’ | ‘Currency’ | ‘String’;

**BigDecimalNumber** = Number, ‘.’, Digit, [Digit], [Digit], [Digit];

**Number** = [‘-‘], (Digit, { Digit });

**ArrayElementReference =** Identifier | (Identifier, ‘[‘, (Identifier | Number), ‘]’);

**Comment** = ‘/\*’, .\*?, ‘\*/’;

**Identifier** = Letter , { Letter | Digit | "\_" } ;

**Letter** = SmallLetter | CapitalLetter;

**ConstBoolean** = ‘true’ | ‘false’;

**ConstString** = ‘”’, .\*?, ‘”’;

**RelOperator** = ‘>‘ | ‘<‘ | ‘>=‘ | ‘<=‘ | ‘!=‘;

**CapitalLetter** = "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";

**SmallLetter** = "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" ;

**Digit** = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

1. **Opis techniczny realizacji**

Język programowania, który wybrałam do realizacji projektu, to Java. Ma dość łatwą i dobrze znaną mi składnie, jest językiem typowanym statycznie. Biblioteki, z których skorzystałam to

- Lombok (znacznie zmniejsza ilość kodu w klasach będących odpowiednikami elementów gramatyki na etapie parsowania dzięki adnotacjom @Getter, @Setter oraz @AllArgumentsConstructor (@NoArgumentsConstructor))

- java.io, java.nio do wczytywania zawartości plików z kodem walutowym oraz definicją walut

- java.math

- biblioteka Gson do parsowania JSON’a do kodu w języku Java

Do testowania jednostkowego na etapach analizatora leksykalnego i składniowego użyłam biblioteki JUnit.

Na standardowe wejście program będzie przyjmował plik tekstowy, zawierający kod w języku walutowym. Na wyjściu zostanie wyprowadzony wynik wykonania kodu walutowego.

Moduły:

- analizator leksykalny (scanner) czyta źródłowy kod walutowy znak po znaku i tworzy z niego tokeny. Następnie przekazuje je do analizatora składniowego (parsera).

- analizator składniowy (parser) sprawdza, czy tokeny na wejściu tworzą poprawne konstrukcje składniowe, zgodne z gramatyką języka walutowego oraz grupuje tokeny w struktury składniowe. Otrzymany wynik przekazuje do generatora (interpretera) kodu

-generator (interpreter) kodu tworzy kod wynikowy, sprawdzając w trakcie zgodność typów, zakres widoczności zmiennych, poprawność operacji matematycznych i logicznych itp.

-główny moduł tworzy abstrakcję komunikacji analizatora leksykalnego z plikami kodu walutowego i uruchamia kod wynikowy w języku Java

(Osobny) etap analizy semantycznej kodu został pominięty z tego powodu, że podstawą języka walutowego jest Java - język, do którego on jest interpretowany.

1. **Jak nauczyć się języka walutowego?**

Po pierwsze, należy nauczyć się Javy. Funkcje dla walut są tylko małym , łatwym i szybkim w zrozumieniu dodatkiem do niej. Należy zwrócić uwagę na to, że konwersja walut odbywa się na zasadzie, że jeżeli są różne, to konwertują się do pierwszej waluty. Konieczne jest zdefiniowanie waluty bazowej (z value równym 1). Jeżeli użytkownik (błędnie) poda kilka bazowych walut, zostanie wykorzystana pierwsza znaleziona w liście. Jeżeli w programie będzie próba użycia skróconej nazwy waluty, która nie została zdefiniowana w jednym albo kilku zaimportowanych plikach, działanie skończy się błędem. Tak samo jak w przypadku braku zaimportowanych plików.

Są dostępne tylko 5 typów danych: int, boolean, String, BigDecimal(przeważnie do operacji walutowych) oraz Currency, który pozwala definiować zmienne postaci wartośćSkrót:

4USD, 18.25 EUR

Akceptowalna jest wersja zarówno ze spacją, jak i bez niej.

Co do szczegółow operacji matematycznych, jest możliwe dodawanie, odejmowanie, mnożenie oraz dzielenie waluty przez stałą typu Integer i BigDecimal:

Currency sum = 2PLN + 3.5RUB;

Currency subtraction = 2PLN - 3.5RUB;

Currency multiplication = 3.15USD \* 3;

Currency division = 15USD / 2.5;

Do wypisywania na ekran w języku walutowym służy funkcja print (wielkość liter ma znaczenie):

print(1+2, 3-1);

Argumenty powinny być podawane przez spację.