Akademia Górniczo-Hutnicza

Wydział Informatyki, Elektroniki i Telekomunikacji

Języki Formalne i Kompilatory

Translator podzbioru języka Java do Pythona

Igor Dzierwa
Adrian Nędza

Cel zadania:

Celem projektu jest realizacja translatora podzbioru języka Java do języka Python.
Podczas implementacji należy skorzystać z generatora skanerów oraz parserów, w zależności od wybranego języka programowania do implementacji translatora:

- Python – SLY, PLY
- Java – ANTLR

Stos technologiczny/Narzędzia:

- **Java** – implementacja translatora.
- **ANTLR** – generator analizatora składni.

Teoria:

Kompilator — program, który czyta kod napisany w jednym języku *(języku źródłowym)* i tłumaczy go na równoważny kod w innym języku (języku wynikowym) z jednoczesnym wykrywaniem ewentualnych błędów popełnionych w trakcie programowania.

- Kompilator to translator tłumaczący program w języku wysokiego poziomu na program w języku maszynowym.

Translator — program lub zespół programów, tłumaczący program źródłowy na równoważny mu program docelowy. Translator ze względu na dane wejściowy i wyjściowe traktujemy jako pewne uogólnienie kompilatora

Interpreter – jest programem, który działa podobnie jak kompilator, jednak nie generuje kodu wynikowego, tylko od razu wykonuje instrukcje zawarte w kodzie źródłowym programu.

Algorytm LL(*) – lewostronny parser z podglądem dowolnej liczby symboli. Celem statycznej analizy gramatycznej LL jest obliczenie wyrażeń wyprzedzających, które przewidują alternatywne produkcje w dowolnym punkcie decyzji gramatycznej.

- Kluczową ideą parserów LL(*) jest użycie wyrażeń regularnych.
- W porównaniu z prawostronnymi, produkcje mogą być w naturalny sposób przestawione w kodzie jako funkcje rekurencyjne, jednak w stosunku do nich parsują mniejszą klasę języków.

ANTLR – generator analizatora składni, który używa algorytmu LL(*) do parsowania języków. Standardowy projekt procesowania danej struktury danych składa się z pliku Grammar zawierającego reguły dla Lexera oraz Parsera, które zaś używane są do tokenizacji oraz przetwarzania danych według wcześniej stworzonych reguł.

Lexer/Skaner – wczytuje strumień znaków składający się na program wejściowy. Jego dane wyjściowe to postać pośrednia programu źródłowego zawierająca atomy wraz z krótkim opisem.

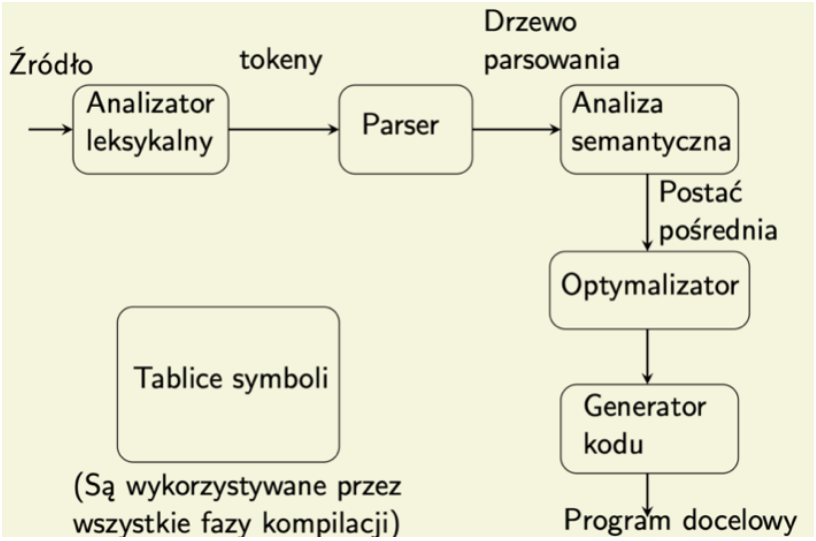
- Grupuje znaki ze strumienia w symbole leksykalne *(tokeny)*
- Buduje i wypełnia tablicę symboli leksykalnych
- Usuwa z pliku wejściowego komentarze, białe znaki, znaki nowych wiersz.

Parser – ma na celu sprawdzenie poprawności syntaktycznej przez dokonanie rozbioru podprogramu na części składowe i zbudowanie odpowiedniego drzewa składniowego. Wykorzystuje pierwsze składniki tokenów, produkowane przez analizator leksykalny, aby utworzyć reprezentację pośrednią, która przedstawia strukturę gramatyczną strumienia tokenów.

- Typową reprezentacją składni jest drzewo syntaktyczne, w którym każdy węzeł wewnętrzny reprezentuje operację, natomiast "dzieci" tego węzła stanowią argumenty operacji. Na jego podstawie parser decyduje, czy składnia programu jest poprawna.

Ogólny schemat działania kompilatora:

1. **Analiza**
   - **Leksykalna *(liniowa)*** – ma miejsce wszędzie tam, gdzie wczytuje się dane o określonej składni.
     Najpierw wczytywany ciąg znaków podzielić na elementarne cegiełki składniowe (leksemy), by dopiero dalej analizować ciąg leksemów.
   - **Składniowa (hierarchiczna)** – zbadanie czy jednostki leksykalne tworzą poprawne konstrukcje danego języka programowania.
   - **Semantyczna** – faza procesu kompilacji, wykonywana po analizie syntaktycznej, a przed generowaniem kodu, w której sprawdzana jest poprawność programu na poziomie znaczenia poszczególnych instrukcji oraz programu jako całości.
2. **Generator kodu pośredniego –** generuje kod w pewnym niskopoziomowym języku, którego przekształcenie na kod bajtowy powinno być ułatwione.
3. **Optymalizator kodu** – poprawa efektywności poprzez przyspieszenie oraz redukcję kodu.
4. **Generator kodu wynikowego** – na podstawie pośredniej reprezentacji programu źródłowego wytwarzany jest równoważny program docelowy.



Realizacja projektu:

1.Dodanie do projektu ANTLR 4 Maven plugin

Dołączenie do projektu ANTLR 4 w postaci Maven plugin pozwala w wygodny sposób skonfigurować środowisko, bez konieczności pobierania i instalowania biblioteki `.jar`.

W celu właściwego ustawienia wzorowaliśmy się na instrukcji zawartej na głównej stronie ANTLR: *https://www.antlr.org/api/maven-plugin/latest/examples/simple.html*.

Konfiguracja ogranicza się do prawidłowego uzupełnienia pliku `pom.xml` (project object model). Jest to dokument XML, który kompleksowo opisuje projekt.

- Dodanie zależności, czyli bibliotek od których będzie zależał nasz projekt – w momencie kiedy nie ma ich na dysku, zostają pobrane z repozytorium Mavena.

```xml
<dependencies>
    <dependency>
        <groupId>org.antlr</groupId>
        <artifactId>antlr4-runtime</artifactId>
        <version>${antlr.version}</version>
    </dependency>
</dependencies>
```

- Skonfigurowanie procesu budowy aplikacji – wybrany zostaje cel (goal) oraz zostaje określony folder domyślny, w którym znajdzie się plik gramatyki oraz folder docelowy, czyli miejsce, w którym zostaną wygenerowane pliki wynikowe.

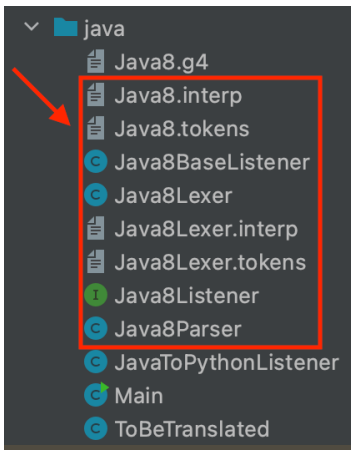
```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.antlr</groupId>
            <artifactId>antlr4-maven-plugin</artifactId>
            <version>${antlr.version}</version>
            <executions>
                <execution>
                    <goals>
                        <goal>antlr4</goal>
                    </goals>
                    <configuration>
                        <outputDirectory>${basedir}/src/main/java</outputDirectory>
                        <sourceDirectory>${basedir}/src/main/java</sourceDirectory>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

2.Określenie gramatyki

Określenie postaci języka wejściowego dla narzędzia ANTLR następuje za pośrednictwem pliku z rozszerzeniem `.g4`.
Skorzystaliśmy z gotowego pliku gramatyki Javy w wersji 8, który znajdował się w głównym repozytorium projektu ANTLR – plik `Java8.g4`.

3.Wygenerowanie Lexera oraz Parsera

Za pośrednictwem komendy **`mvn package`** zostaje wygenerowane kilka plików na bazie zdefiniowanej gramatyki – `Java8.g4`.



Najważniejsze wygenerowane pliki to:

- **`Java8Lexer.java`** – zawiera implementację lexera dla gramatyki. Plik zawiera reguły, które przekształcają słowa wejściowe na tokeny. Zapisane są w gramatye jako reguły rozpoczynające się dużymi literami lub jako symbole w apostrofach

- **`Java8Paraser.java`** – zawiera implementację parsera dla gramatyki. Plik ten zawiera reguły, które przekształcają tokeny wejściwe na drzewo syntaktyczne zgodnie z produkcjami zapisanymi w gramatyce.

- **`Java8Listener.java`** oraz **`Java8BaseListener`** – to interfejs i klasa, które umożliwiają przechodzenie po drzewie syntaktycznym poprzez mechanizm listenerów.

4.Instalacja ANTLR

By móc przetestować Lexer oraz Parser za pośrednictwem **`org.antlr.v4.gui.TestRig`**, musieliśmy zainstalować ANTLR, pobierając najnowszą wersję – `antlr-4.9.2-complete.jar`.

Dodanie klas z pobranego pliku do zmiennej CLASSPATH oraz utworzenie aliasów, które uproszczą składnię z linii komend – plik `.bash_profile` /plik `.zshrc`:

```
# ANTLR 4
export ANTLR_HOME="/Users/igordzierwa/Desktop/java-to-python-translator-master/src/main/java/ANTLR"
export ANTLR_JAR="$ANTLR_HOME/antlr-4.9.2-complete.jar"
export CLASSPATH=".:$ANTLR_JAR:$CLASSPATH"
alias antlr4="java -jar $ANTLR_JAR"
alias grun="java org.antlr.v4.gui.TestRig"
```

Następnie należy skompilować wszystkie wygenerowane pliki `.java`:

- `javac Java8*.java`

5.Testy działania Lexera

Aby przetestować działanie Lexera, korzystamy z narzędzia dostarczonego przez ANTLR – Testrig. Komenda będzie przyjmować następujące argumenty:

- Nazwa gramatyki / paczki implementującej Lexer – `Java8`
- Nazwa reguły, co oznacza, że istnieje możliwość testowania gramatyki niezależnie – `java8_file`
- Opcja wygenerowanego wyniku – `tokens`, ponieważ zależy nam na analizie wygenerowanych tokenów.
- Ścieżka do pliku testowego – nazwa pliku testowego `ToBeTranslated.java`

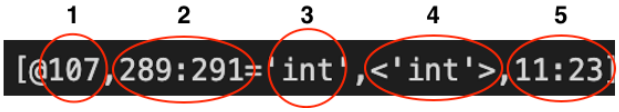
Finalna wersja polecenia:

- `grun Java8 java8 -tokens ToBeTranslated.java`

Część wygenerowanych tokenów:

```
[@0,0:5='public',<'public'>,1:0]
[@1,6:6=' ',<WS>,channel=1,1:6]
[@2,7:11='class',<'class'>,1:7]
[@3,12:12=' ',<WS>,channel=1,1:12]
[@4,13:26='ToBeTranslated',<Identifier>,1:13]
[@5,27:27=' ',<WS>,channel=1,1:27]
[@6,28:28='{',<'{'>,1:28]
[@7,29:33='\n    ',<WS>,channel=1,1:29]
[@8,34:37='void',<'void'>,2:4]
[@9,38:38=' ',<WS>,channel=1,2:8]
[@10,39:57='arithmeticOperation',<Identifier>,2:9]
[@11,58:58='(',<'('>,2:28]
[@12,59:61='int',<'int'>,2:29]
[@13,62:62=' ',<WS>,channel=1,2:32]
[@14,63:63='a',<Identifier>,2:33]
[@15,64:64=',',<','>,2:34]
[@16,65:65=' ',<WS>,channel=1,2:35]
[@17,66:71='double',<'double'>,2:36]
[@18,72:72=' ',<WS>,channel=1,2:42]
[@19,73:73='d',<Identifier>,2:43]
[@20,74:74=',',<','>,2:44]
[@21,75:75=' ',<WS>,channel=1,2:45]
[@22,76:81='double',<'double'>,2:46]
[@23,82:82=' ',<WS>,channel=1,2:52]
[@24,83:83='b',<Identifier>,2:53]
[@25,84:84=')',<')'>,2:54]
[@26,85:85='{',<'{'>,2:55]
[@27,86:94='\n        ',<WS>,channel=1,2:56]
[@28,95:100='double',<'double'>,3:8]
[@29,101:101=' ',<WS>,channel=1,3:14]
[@30,102:102='c',<Identifier>,3:15]
[@31,103:103=' ',<WS>,channel=1,3:16]
[@32,104:104='=',<'='>,3:17]
[@33,105:105=' ',<WS>,channel=1,3:18]
[@34,106:106='a',<Identifier>,3:19]
[@35,107:107=' ',<WS>,channel=1,3:20]
[@36,108:108='+',<'+'>,3:21]
[@37,109:109=' ',<WS>,channel=1,3:22]
[@38,110:110='b',<Identifier>,3:23]
[@39,111:111=' ',<WS>,channel=1,3:24]
[@40,112:112='+',<'+'>,3:25]
[@41,113:113=' ',<WS>,channel=1,3:26]
[@42,114:114='d',<Identifier>,3:27]
[@43,115:115=';',<';'>,3:28]
[@44,116:124='\n        ',<WS>,channel=1,3:29]
[@45,125:130='System',<Identifier>,4:8]
[@46,131:131='.',<'.'>,4:14]
[@184,504:504=')',<')'>,19:50]
[@185,505:505=';',<';'>,19:51]
[@186,506:514='\n        ',<WS>,channel=1,19:52]
[@187,515:515='}',<'}'>,20:8]
[@188,516:526='\n\n\n        ',<WS>,channel=1,20:9]
[@189,527:528='if',<'if'>,23:8]
[@190,529:529=' ',<WS>,channel=1,23:10]
[@191,530:530='(',<'('>,23:11]
[@192,531:531='x',<Identifier>,23:12]
[@193,532:533='!=',<'!='>,23:13]
[@194,534:534='y',<Identifier>,23:15]
[@195,535:535=')',<')'>,23:16]
[@196,536:536='{',<'{'>,23:17]
[@197,537:549='\n            ',<WS>,channel=1,23:18]
[@198,550:555='System',<Identifier>,24:12]
[@199,556:556='.',<'.'>,24:18]
[@200,557:559='out',<Identifier>,24:19]
[@201,560:560='.',<'.'>,24:22]
[@202,561:567='println',<Identifier>,24:23]
[@203,568:568='(',<'('>,24:30]
[@204,569:589='"inside is statement"',<StringLiteral>,24:31]
[@205,590:590=')',<')'>,24:52]
[@206,591:591=';',<';'>,24:53]
[@207,592:600='\n        ',<WS>,channel=1,24:54]
[@208,601:601='}',<'}'>,25:8]
[@209,602:602=' ',<WS>,channel=1,25:9]
[@210,603:606='else',<'else'>,25:10]
[@211,607:607=' ',<WS>,channel=1,25:14]
[@212,608:608='{',<'{'>,25:15]
[@213,609:621='\n            ',<WS>,channel=1,25:16]
[@214,622:627='System',<Identifier>,26:12]
[@215,628:628='.',<'.'>,26:18]
[@216,629:631='out',<Identifier>,26:19]
[@217,632:632='.',<'.'>,26:22]
[@218,633:637='print',<Identifier>,26:23]
[@219,638:638='(',<'('>,26:28]
[@220,639:661='"inside else statement"',<StringLiteral>,26:29]
[@221,662:662=')',<')'>,26:52]
[@222,663:663=';',<';'>,26:53]
[@223,664:672='\n        ',<WS>,channel=1,26:54]
[@224,673:673='}',<'}'>,27:8]
[@225,674:678='\n    ',<WS>,channel=1,27:9]
[@226,679:679='}',<'}'>,28:4]
[@227,680:680='\n',<WS>,channel=1,28:5]
[@228,681:681='}',<'}'>,29:0]
[@229,682:682='\n',<WS>,channel=1,29:1]
[@230,683:682='<EOF>',<EOF>,30:0]
```

- **W skład każdego tokenu wchodzi:**
    1. Numer tokenu.
    2. Numer początkowego i końcowego znaku, z którego składa się token (licząc od początku pliku).
    3. Tekst z jakiego składa się token.
    4. Typ tokenu, czyli reguła z gramatyki zaczynająca się z dużej litery lub w symbol w apostrofach.
    5. Pozycja tokenu – numer wiersza oraz numer znaku w tym wierszu.



- Można zauważyć, że koniec pliku jest również określany poprzez token:

```
231    [@230,683:682='<EOF>',<EOF>,30:0]
```

5.Testy działania Parsera

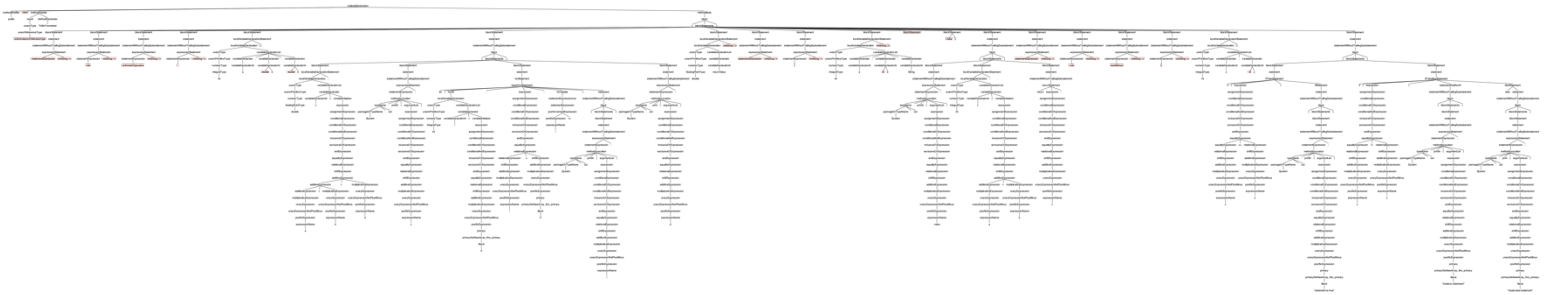
Aby przetestować działanie Parsera, korzystamy z narzędzia dostarczonego przez ANTLR – Testrig. Komenda będzie przyjmować następujące argumenty:

- Nazwa gramatyki / paczki implementującej Lexer – `Java8`
- Nazwa reguły, co oznacza, że istnieje możliwość testowania gramatyki niezależnie – `java8_file`
- Opcja wygenerowanego wyniku – `tree`, ponieważ zależy nam na analizie wygenerowanego drzewa syntaktycznego.
- Ścieżka do pliku testowego – nazwa pliku testowego `ToBeTranslated.java`
- Opcjonalna flaga – `gui` – pozwala na wizualizację drzewa syntaktycznego *(bez niej generowane jest w formie tekstu).*

Finalna wersja polecenia:

```
grun Java8 methodDeclaration -tree ToBeTranslated.java -gui
```

Wygenerowane drzewo syntaktyczne dla deklaracji metod:



6.Opis programu

Klasa Main:

- **Określenie danych wejściowych:**

```
// args [0] - path to java file to be translated
CharStream stream = CharStreams.fromFileName("src/main/java/ToBeTranslated.java"); //args[0]
```

- **Utworzenie lexera oraz parsera:**

```
// Get our lexer
Java8Lexer lexer = new Java8Lexer(stream);

// Get a list of matched tokens
CommonTokenStream tokens = new CommonTokenStream(lexer);

// Pass the tokens to the parser
Java8Parser parser = new Java8Parser(tokens);
```

- **Utworzenie drzewa syntaktycznego:**

```
ParseTree parseTree = parser.classDeclaration();
```

- **Przechodzenie po wygenerowanym drzewie –** realizowane przez klasę z biblioteki ANTLR `ParseTreeWalker`:

```
ParseTreeWalker parseTreeWalker = new ParseTreeWalker();
Java8Listener listener = new JavaToPythonListener();
parseTreeWalker.walk(listener, parseTree);
```

  - Klasa jako parametr przyjmuje obiekt `JavaToPythonListener` implementujący interfejs `Java8Listener`, w którym zostały rozwinięte niektóre ze zdefiniowanych metod.
  - Sposób przechodzenia po drzewie opiera się na mechanizmie listenerów, które reagują na wydarzenia związane z przechodzeniem po drzewie syntaktycznym. Podstawowymi zdarzeniami są wejścia i wyjścia z danego typu węzła.

Klasa JavaToPythonListener – czyli przykłady zaimplementowanych listenerów:

- Deklaracja metody:

```
@Override
public void enterMethodDeclaration(Java8Parser.MethodDeclarationContext ctx) {
    System.out.println();
    System.out.print("def " + ctx.methodHeader().methodDeclarator().Identifier().getText());
}

@Override
public void exitMethodDeclaration(Java8Parser.MethodDeclarationContext ctx) {
    System.out.println();
}

@Override
public void enterMethodHeader(Java8Parser.MethodHeaderContext ctx) { System.out.print("("); }

@Override
public void exitMethodHeader(Java8Parser.MethodHeaderContext ctx) { System.out.println("):"); }
```

- Deklaracja argumentów w metodzie:

```
public void removeTypeParameter(StringBuilder stringBuilder, String formalParameterText, String splitter) {
    List<String> basicTypes = Arrays.asList("int", "double", "float", "short", "long", "byte", "boolean", "char", "String");
    for (String basicType : basicTypes) {
        if(formalParameterText.startsWith(basicType)) {
            stringBuilder.append(formalParameterText.substring(basicType.length()) + splitter);
        }
    }
}

@Override
public void enterFormalParameterList(Java8Parser.FormalParameterListContext ctx) {
    StringBuilder stringBuilder = new StringBuilder();
    for(Java8Parser.FormalParameterContext formalParameterContext: ctx.formalParameters().formalParameter()) {
        removeTypeParameter(stringBuilder, formalParameterContext.getText(), splitter: ", ");
    }
    System.out.print(stringBuilder.toString());
}

@Override
public void exitFormalParameterList(Java8Parser.FormalParameterListContext ctx) {
    StringBuilder stringBuilder = new StringBuilder();

    removeTypeParameter(stringBuilder, ctx.lastFormalParameter().formalParameter().getText(), splitter: "");

    System.out.print(stringBuilder.toString());
}
```

- Deklaracja zmiennych:

```
@Override
public void enterVariableDeclaratorList(Java8Parser.VariableDeclaratorListContext ctx) {
    System.out.print(ctx.variableDeclarator( i: 0).variableDeclaratorId().getText() + "=");
}

@Override
public void exitVariableDeclaratorList(Java8Parser.VariableDeclaratorListContext ctx) { System.out.println(); }
```

- Deklaracja klasy:

```
@Override
public void enterNormalClassDeclaration(Java8Parser.NormalClassDeclarationContext ctx) {
    System.out.println(ctx.CLASS().getText() + " " + ctx.Identifier().getText() + ":");
}
```

- Definicja preinkrementacji oraz postinkrementacji:

```
@Override
public void enterStatementExpression(Java8Parser.StatementExpressionContext ctx) {
    String text = ctx.getText();
    if(text.startsWith("++")){
        System.out.println(text.replace( target: "++",  replacement: "+=1"));
    } else if(text.startsWith("--")){
        System.out.println(text.replace( target: "--",  replacement: "-=1"));
    }
}
```

- Definicja metody print():

```
@Override
public void enterMethodInvocation(Java8Parser.MethodInvocationContext ctx) {
    if (ctx.getText().startsWith("System.out.println") || ctx.getText().startsWith("System.out.print")){
        System.out.print("print(");
    }
}

@Override
public void exitMethodInvocation(Java8Parser.MethodInvocationContext ctx) {
    if (ctx.getText().startsWith("System.out.println")){
        System.out.println(")");
    } else if (ctx.getText().startsWith("System.out.print")){
        System.out.println(", end='')");
    }
}
```

- Definicja instrukcji warunkowej:

```
@Override
public void enterStatementNoShortIf(Java8Parser.StatementNoShortIfContext ctx) { System.out.println(":"); }

@Override
public void enterIfStatement(Java8Parser.IfStatementContext ctx) { System.out.println(":"); }

@Override
public void enterElseStatement(Java8Parser.ElseStatementContext ctx) { System.out.println("else:"); }

@Override
public void enterIfThenStatement(Java8Parser.IfThenStatementContext ctx) { System.out.print("if "); }

@Override
public void enterIfThenElseStatement(Java8Parser.IfThenElseStatementContext ctx) { System.out.print("if "); }
```

- Definicja wyrażenia warunkowego:

```
@Override
public void enterConditionalExpression(Java8Parser.ConditionalExpressionContext ctx) {
    System.out.print(ctx.getText());
}

@Override
public void exitConditionalExpression(Java8Parser.ConditionalExpressionContext ctx) {
    String text = ctx.getText();
    if(text.contains("<") || text.contains(">")) System.out.println("):");
}
```

- Definicja instrukcji return:

```
@Override
public void enterReturnStatement(Java8Parser.ReturnStatementContext ctx) {
    System.out.print("return ");
}

@Override
public void exitReturnStatement(Java8Parser.ReturnStatementContext ctx) { System.out.println(); }
```

- Definicja pętli for *(w przypadku Pythona najbliższym odpowiednikiem jest while):*

```java
@Override
public void exitForInit(Java8Parser.ForInitContext ctx) { System.out.print("\nwhile("); }

@Override
public void exitBasicForStatement(Java8Parser.BasicForStatementContext ctx) {
    String forUpdateText = ctx.forUpdate().getText();
    if(forUpdateText.endsWith("++")){
        System.out.println(forUpdateText.replace( target: "++",  replacement: "+=1"));
    } else if(forUpdateText.endsWith("--")){
        System.out.println(forUpdateText.replace( target: "--",  replacement: "-=1"));
    } else if (!forUpdateText.contains("++") && !forUpdateText.contains("--")){
        System.out.println(forUpdateText);
    }
    System.out.println();
}
```

7.Testowe uruchomienie programu

Plik Wejściowy:

```java
public class ToBeTranslated {
    void arithmeticOperation(int a, double d, double b){
        double c = a + b + d;
        System.out.println(c);
        for (int i = 0; i < 3; i++){
            System.out.println(i);
        }
        System.out.print(d);
    }

    double returnValue(int a, int b, String value) {
        System.out.println(value);
        int c = a + b;
        return c;
    }

    void newMethod(int x, int y){
        if(x==y){
            System.out.println("statement is true");
        }


        if (x!=y){
            System.out.println("inside is statement");
        } else {
            System.out.print("inside else statement");
        }
    }
}
```

Wynik programu:

```
Run:  Main
/Library/Java/JavaVirtualMachines/amazon-corretto-11.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA.app/Contents/lib/idea_rt.jar=59681:/Applications/IntelliJ IDEA.app/Contents
class ToBeTranslated:

def arithmeticOperation(a, d, b):
c=a+b+d
print(c)
i=0

while(i<3):
print(i)
i+=1
print(d, end='')


def returnValue(a, b, value):
print(value)
c=a+b
return c


def newMethod(x, y):
if x==y:
print("statement is true")
if x!=y:
print("inside is statement")
else:
print("inside else statement", end='')


Process finished with exit code 0
```

8.Podsumowanie

Translator został wykonany zgodnie z architekturą współczesnych interpreterów oraz translatorów. Składa się z lexera i parsera.
W wyniku działania lexera i parsera otrzymywane jest drzewo syntaktyczne. Przechodzenie po drzewie pozwala na wykonywanie działań zgodnie z regułami translacji.

Literatura:

1. https://www.antlr.org - *główna strona generatora ANTLR*

2. https://github.com/antlr/antlr4 - *główna strona repozytorium generatora ANTLR*

3. https://www.antlr.org/api/maven-plugin/latest/index.html - *opis konfiguracji ANTLR Maven plugin*

4. https://www.cs.sjsu.edu/~mak/tutorials/InstallANTLR4.pdf - *opis instalacji oraz ustawiania aliasów w pliku konfiguracyjnym (na potrzeby testów Lexera oraz Parsera).*

5. https://tomassetti.me/antlr-mega-tutorial/- *blog zawierający opis funkcjonalności ANTLR.*

6. https://blog.knoldus.com/testing-grammar-using-antlr4-testrig-grun/ - *przykład użycia testowego narzędzia*

xw