Sprawozdanie

Temat : Translatory

Przedmiot : Języki formalne i kompilatory
Prowadzący: dr inż. Radosław Klimek
Autor: Piotr Klimiec
Grupa: środa 17:00

Wstęp:

Tematem projektu był napisanie translatora języka Java przy wykorzystaniu dostępnych na rynku parserów.

W pierwszej fazie skupiłem się na wybraniu i opisaniu zalet oraz minusów dostępnych parserów następnie wybrałem jeden z nich w celu realizacji zadania praktycznego.

Projekt pomógł zapoznać się w sposób praktyczny i teoretyczny z :

- Formalnym zapisem języków programowania
- Analizą oraz sposobem pisania gramatyk
- Zaznajomieniem procesu parsowania

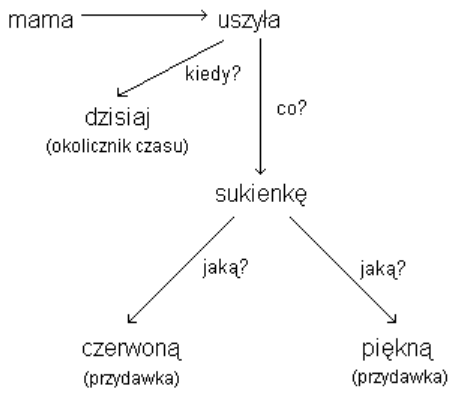
Opis narzędzi poprzedza krótki wstępem teoretycznym procesu translacji, parsowania, tokenizacji, gramatyk itd.

Trochę teorii:

Narzędzie służące do translacji jezyka programowania nazywamy translatorem. Aby przeprowadzić zadaną translację translator musi **być w staniu rozpoznać wszystkie zdania, wyrażenia oraz słowa danego języka w sposób jednoznaczny**. Na przykład operacja var = 100 musi być rozpoznania jako operacja przypisania , parametr var musi być rozpoznany jako Lvalue, 100 jako wartość przypisania. Po poprawnym rozpoznaniu poszczególnych struktur możemy nadać im odpowiednie znaczenie.

Program który potrafi rozpoznać język i nadać mu znaczenie nazwany jest parserem.

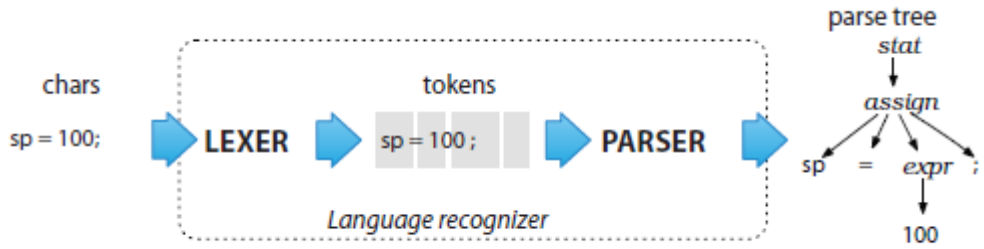
Mama uszyła dzisiaj piękną czerwoną sukienkę.



Parser na wejściu otrzymuje ciąg znaków który początkowo (dla parsera) nie ma żadnego znaczenia. To programista piszący program wie, że x =10 to instrukcja przypisania a x == 10 porównania. Aby komputer był to w zdanie zrozumieć dokładnie tak jak my proces parsowania rozbity jest na dwa etapy:

- **Tokenizacji** : czyli wyodrębnieniu z ciągu znaków poszczególnych fragmentów – tokenów. Program który wykonuje tą czynność nazywany jest **Lexerem**
- **Parsowania** właściwego: czyli rozpoznaniu z dostarczonych tokenów poszczególnych struktur takich jak instrukcja przypisania, instrukcja warunkowa itd.

Poniższy rysunek przedstawia proces parsowania:



Drzewa AST (Abstract Syntax Tree) są używane jako wewnętrzna reprezentacja procesu parsowania. Liśćmi takiego drzewa są tokeny uzyskane w procesie tokenizacji (wyszczególnienie elementów wejściowy), natomiast wewnętrzne węzły grupują swoje dzieci w coraz to bardziej ogólne struktury, aż do korzenia który jest najbardziej abstrakcyjną formułą opisująca nasz program.

Nazwy poszczególnych węzłów z reguły odpowiadają ich nazwie w dostarczonej gramatyce, dlatego tak ważne jest aby nazwy te niosły ze sobą znaczenie.

Gramatyka to nic innej jak formalny opis naszego języka na podstawie którego jesteśmy w stanie przy pomocy odpowiednich programów wygenerować parser który inaczej musielibyśmy napisać ręcznie – co jest dość czasochłonnych i podatnym na błędy procesem.

Przegląd dostępnych technologii:

Poniżej znajduje się lista oraz opis technologii umożliwiających generowanie parserów:

1) **ANTLR** to narzędzie służące do tworzenia kompilatorów oraz translatorów z opisu gramatyki zawierającego akcje w języku Java, C++, C# lub Python. W przeciwieństwie do narzędzi takich jak Bison czy SableCC, ANTLR generuje parser typu LL(k). Z tego powodu formalny opis parsera oraz leksera jest bardzo podobny, a generowany kod jest czytelny. Domyślnie ANTLR generuje lekser i parser w Javie, a plik z gramatyką ma rozszerzenie.g. ANTLR akceptuje trzy rodzaje gramatyk: parsery, leksery oraz drzewa parserów.

   Drzewa AST (Abstract Syntax Tree) są używane jako wewnętrzna reprezentacja strumienia wejściowego, zwykle tworzone są podczas parsowania. Ponieważ drzewa AST są dwuwymiarowe, mogą zarówno przechowywać strukturę wejścia (rozpoznaną przez parser) jak i symbole wejściowe.

2) **JLex** – generator analizatorów leksykalnych napisany w Javie dla Javy
   - dostępny kod źródłowy
   - Licencja open Source
   - Oparty na Lexie
   - Współpracuje z generatorem parserów CUP
   - Niezbyt bogata dokumentacja

3) **Gramatica** to generator parserów dla C# i Javy, wspierający gramatyki LL(k), który wyróżnia się od innych poprzez:
   - Separację kodu źródłowego i gramatyki
   - Szczegółowe informacje o błędach
   - Automatyczne error recovery
   - Przejrzysty kod pisanych parserów
   - Możliwość tworzenia parserów w trybie „run-time” (w locie)

   Parsowanie z użyciem Gramatica jest jednak wolniejsze w porównaniu z większością innych narzędzi, co jest największą wadą tego narzędzia.Generator parserów Grammatica jest udostępniony na zasadach licencji GNU LGPL.

4) **JavaCC** - generator parserów i skanerów. JavaCC czyta opis języka i generuje kod, napisany w Javie, który będzie analizował dany język.
Podobnie jak Yacc JavaCC generuje parser dla gramatyki w rozszerzonej notacji Backusa-Naura. Inaczej jednak niż w przypadku programu Yacc, JavaCC generuje parsery zstępujące (ang. top-down parser). Oznacza to, iż JavaCC umożliwia operowanie jedynie na gramatykach typu LL(k).

   Rozprowadzany na zasadach licencji BSD.

5) Coco / R to generator kompilatorów, który na podstawie przypisanej gramatyki języka źródłowego generuje skaner i parser dla tego języka. Skaner pracuje jako deterministyczny automat skończony.
Skaner obsługuje Unicode w kodowaniu UTF-8, z uwzględnieniem wielkości liter (lub bez). Oprócz symboli końcowych może również uznawać tokeny, które nie są częścią składni, ale mogą wystąpić gdziekolwiek w strumieniu wejściowym (np. dyrektywy kompilatora lub znaki końca linii). Skaner może czytać z dowolnego wejścia (nie tylko pliku).

   Parser wykorzystuje schodzenie rekurencyjne. W zasadzie gramatyka musi być LL(1), ale Coco/R może również obsługiwać noLL(1), za pomocą tzw. resolwera.

2. Porównanie znalezionych generatorów.

Narzędzie	Zalety	Wady
ANTLR	● Definicja leksera i gramatyki w jednym pliku (rozszerzenie *.g) ● Możliwość zdefiniowania analizatora drzewa ● Opis gramatyki w formie EBNF ● Automatyczne tworzenie drzewa składniowego ● generowany kod jest czytelny ● wspiera gramatyki typu LL(k),	● nie obsługuje preprocesora ● generowane drzewko wyrażeń w większości przypadków mało przydatne
JLex	● Dostępny kod źródłowy ● Oparty na Lexie ● Współpracuje z generatorem parserów CUP	● Uboga dokumentacja
Gramatica	● Separację kodu źródłowego i gramatyki ● Szczegółowe informacje o błędach ● Automatyczne error recovery ● Przejrzysty kod pisanych parserów ● Możliwość parsowania bez generowania kodu (w locie) ● wspiera gramatyki typu LL(k),	● parsowanie wolniejsze w stosunku do innych narzędzi ● mniejsza dostępność gotowych gramatyk w stosunku np. do ANTLR

JavaCC	● wspiera kodowanie Unicode 16 bitowe ● posiada dobrą integrację skanera z parserem ● wspiera EBNF(opis j. formalnych) ● wspiera budowę abstrakcyjnego drzewa składniowego (AST) ● jest dobrze wspierane (dedykowane grupy dyskusyjne) ● Dobre raportowanie błędów oraz debuggging	● kod szybko się rozrasta powodując duże trudności w wyszukiwaniu błędów ● integralność i poprawność AST jest pozostawiona programiście, w przypadku błędów, ● operowanie jedynie na gramatykach typu LL(k), powstanie zdegenerowane drzewo, błędy te są bardzo trudne do odnalezienia ● nie buduje tablic symboli -> parsery oparte na JavaCC
Coco / R	● Skaner obsługuje Unicode w kodowaniu UTF-8 ● wspiera gramatyki typu LL(k), ● Oprócz symboli końcowych może również uznawać tokeny, które nie są częścią składni(np. dyrektywy kompilatora lub znaki końca linii) ● Skaner może czytać z dowolnego wejścia (nie tylko pliku)	

Wybór narzędzia :

W celu implementacji projektu praktycznego zdecydowałem się na użycie generatora parserów – **ANTLR 4.0** oraz **3.***

Na podstawie dostarczonej gramatyki, której pisanie zostało w znaczny sposób uproszczone w nowej wersji ANTLRa generowany jest parser który który potrafi zbudować drzewo parsujące jak również mechanizmy do jego przechodzenia. Takie podejście w znaczy sposób uproszcza budowę narzędzie językowych, translatorów itd.

INSTALACJA :

ANTLR składa się z jednego plku jar który należy pobrać ze strony producenta. Co prawda generowane parsery mogą być pisane w różnych językach, jednak sam ANTLR został napisany w javie, dlatego do jego obsługi wymagane jest JDK.

Całość obsługiwane jest z linii komend, dlatego poleca się pracę z tym narzędziem z systemów unixowych.

Oprócz generowania parsera, program udostępnia również pomocne narzędzia jak wizualizacja drzewa AST – obsługa również z linii komend.

```
$ cd /usr/local/lib
$ wget http://antlr4.org/download/antlr-4.0-complete.jar
$ export CLASSPATH=".:/usr/local/lib/antlr-4.0-complete.jar:$CLASSPATH"
$ alias antlr4='java -jar /usr/local/lib/antlr-4.0-complete.jar'
$ alias grun='java org.antlr.v4.runtime.misc.TestRig'
```

Opis projektu :

Napisany translator jezyka Java posiada następujące funkcję:

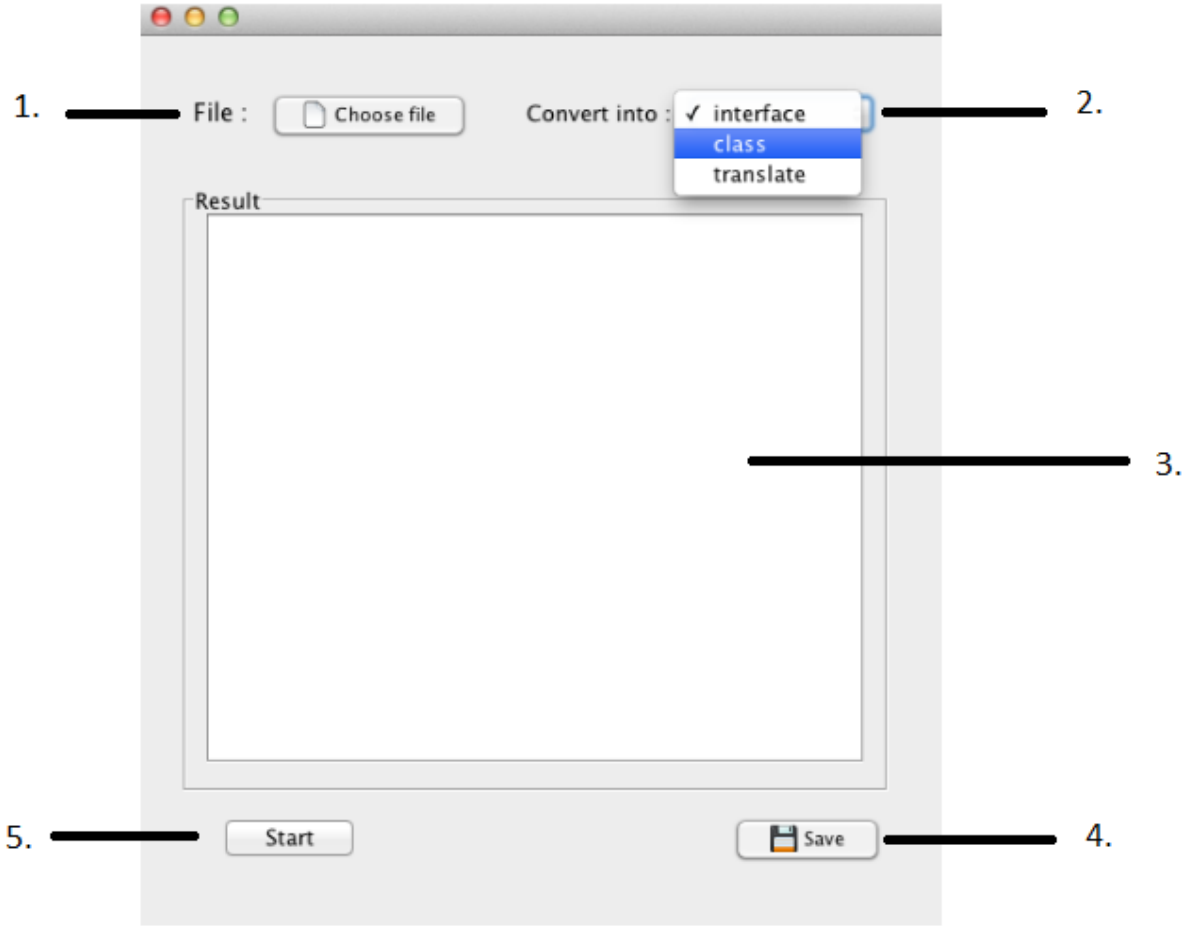
a) Tworzenie interfejsu na podstawie dostarczonej klasy.

b) Tworzenie klasy poprzez zaimplementowanie wszystkich metod dostarczonego interfejsu

c) Translacje do języka Pascal

Dwie pierwsze właściwości tego projektu stanowią realne i przydatne narzędzia których napisanie było by niemożliwe bez parsowania kodu javy a następnie jej obróbki.

Okno programu



1. Wybór pliku do parsowania : okno JFileChooser

2. Operacja która ma zostać wykonana na dostarczonym pliku

3. Pole wyniku : wizualna prezentacja rezultatu parsowania

4. Zapis rezultatu parsowania do pliku (brak implementacji w kodzie)

5. Rozpoczecie procesu parsowania, poprzedzone zerowaniem rezultatu

Całość została zrealizowana w oparciu o wzorzec **MVC**. Cała logika sterowania aplikacją znajduje się w Controlerze który posiada zarówno referencje do widoku jak i do poszczególnych modeli.

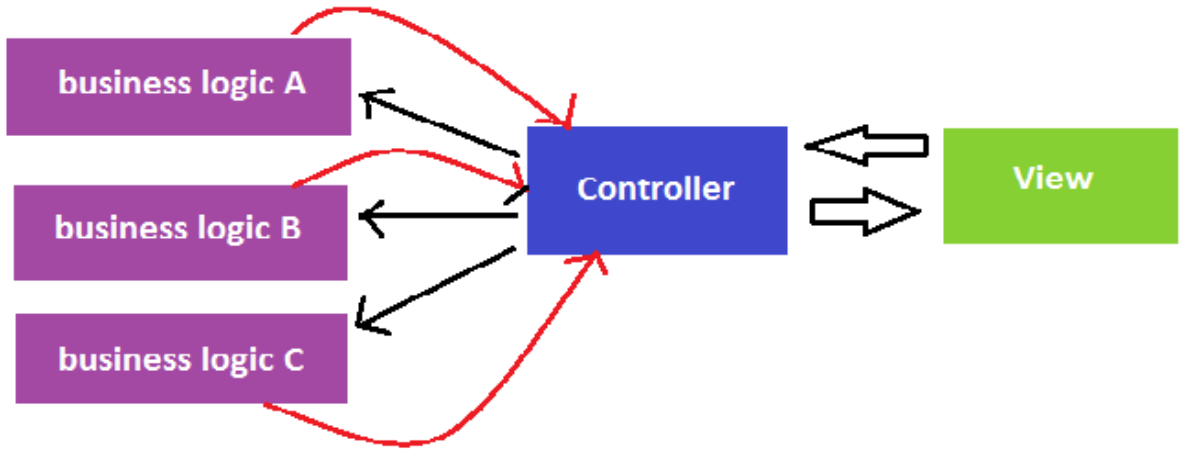
Widok komunikuje się z controlerem poprzez wywoływanie odpowiednich metod. Następnie polecenia te kierowane są do odpowiednich modeli, a rezultat obliczeń z modeli dostarczany jest do controlera poprzez mechanizm **KVO (key-value-observing)**.

Controler rejestruje siebie jako słuchacza zdarzeń w poszczególnych modelach i reaguje odpowiednio po otrzymaniu konkretnych wiadomości z modeli. Całość zbudowana jest o wzorzec obserwator. W tym celu wykorzystałęm gotowe mechanizmy jezyka java w postaci:

- interfejsu **PropertyChangeListener**

- klasy **PropertyChangeSupport**

Schemat programu



Gdzie:

- business logic A : model odpowiedzialny za translacje pliku klasy do interfejsu

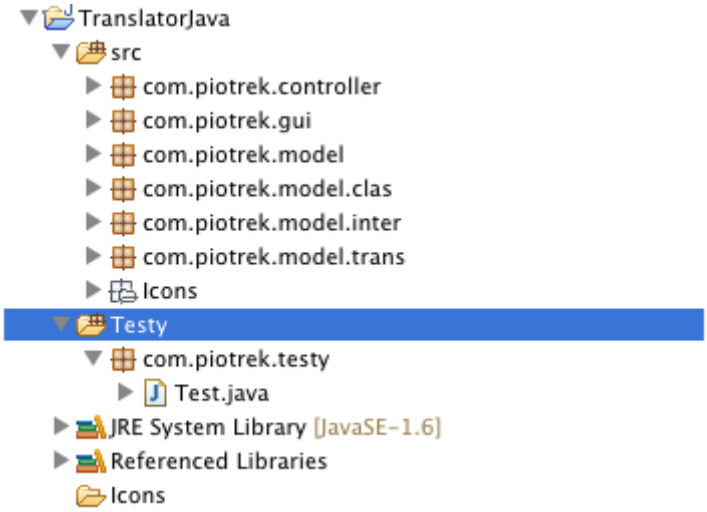
- business logic B : model odpowiedzialny za translacje interfejsu do klasy

- business logic C : model odpowiedzialny za translacje do języka Pascal

Uruchomienie projektu:

Projekt zbudowany jest z kilkunastu klas zgrupowanych w 7 pakietach.

Struktura projektu



Aby uruchomić program należy kliknąć w : folder Testy → pakiet com.piotrek.testy → Test.java

W wyniku tej sekwencji na ekraniu pojawi się okno głowne. Pliki testowe dla każdego rodzaju translacji dostępne są w folderze testy składającego się z 3 podfolderów:

>> ConvertClassToInterface : zawiera plik demonstrujący translacje z klasy do interfejsu

>> ConvertInterfaceToClass : zawiera plik demonstrujący translację z interfejsu do klasy

>> ConvertFromPascal : zawiera plik typu *.pas demonstrujący translację pomiędzy dialektem Pascala a jazykiem Java.

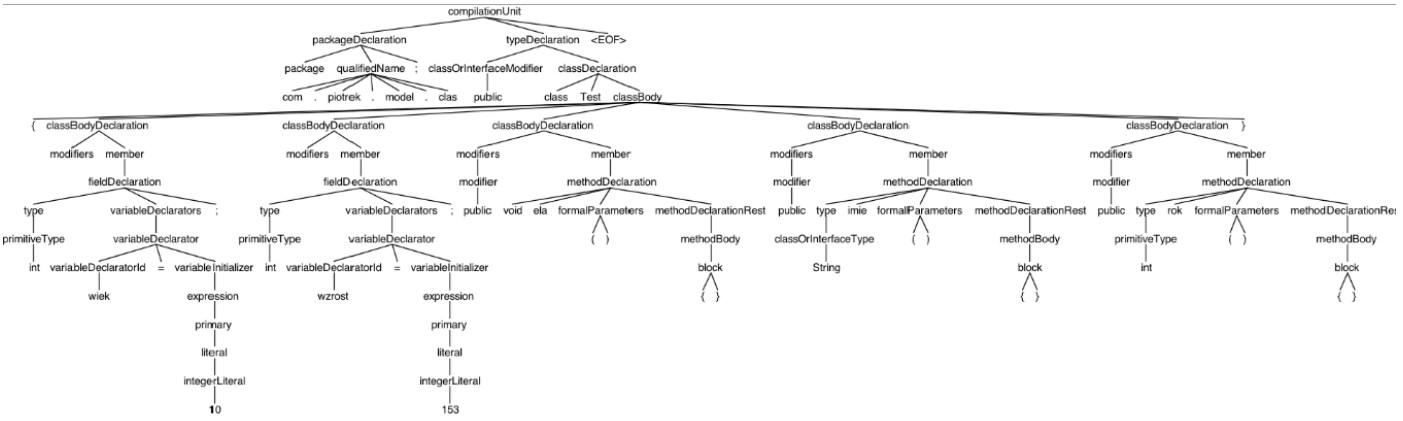
Translacja do interfejsu:

Jako dane wejściowe translator otrzymuje plik klasy. Celem translacji jest ‘wyłowienie’ wszystkich nagłówków metod oraz utworzenie z nich interfejsu.

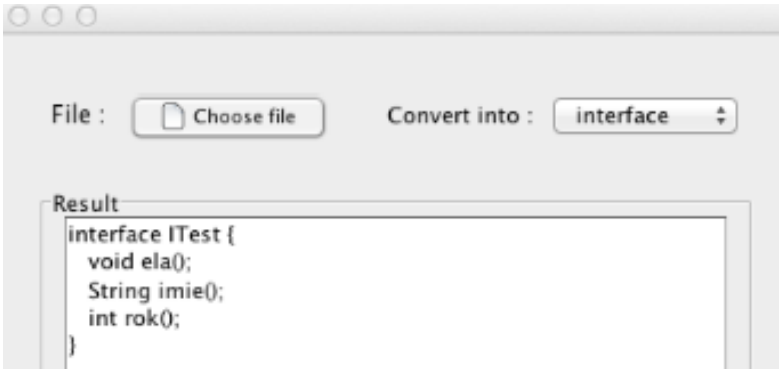
Dla danych wejściowych:

```
1  package com.piotrek.model.clas;
2
3  public class Test {
4
5    // Attributes
6    int wiek = 10;
7    int wzrost = 153;
8
9    // Methods
10   public void ela(){}
11   public String imie(){}
12   public int rok(){}
13
14 }
```

Nasze parser utworzy następujące drzewo:



A otrzymany wynik zostanie wyświetlony w oknie programu :



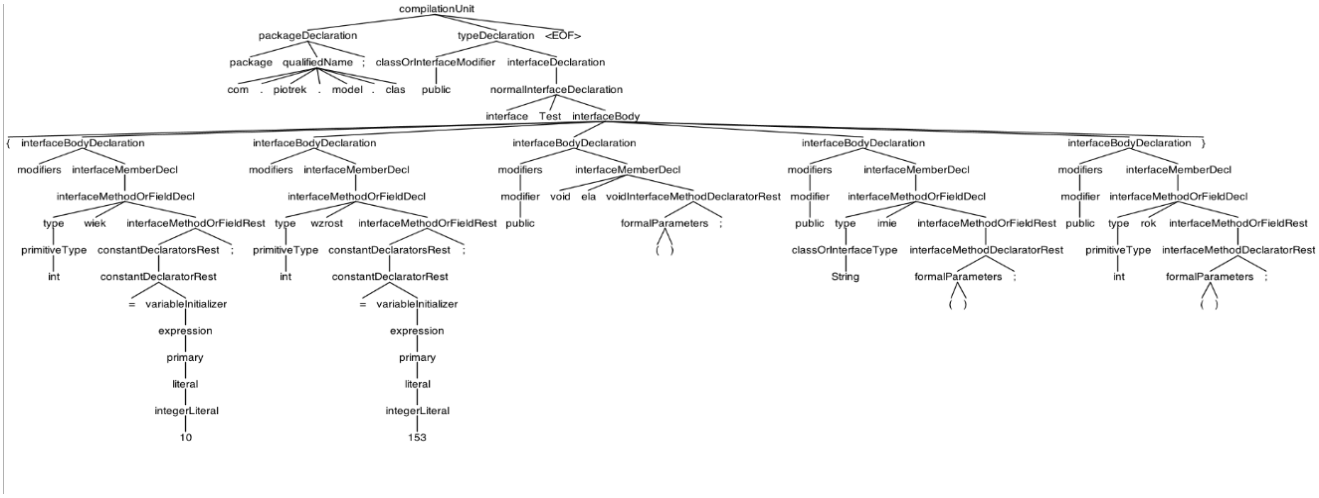
Translacje do klasy:

Jako dane wejściowe translator otrzymuje plik z opisem interfejsu. Celem translacji jest zachowanie struktury interfejsu, przy jednoczesnej implementacji wszystkich metod, poprzez umiejscowienie pustych nawiasów po deklaracjach funkcji

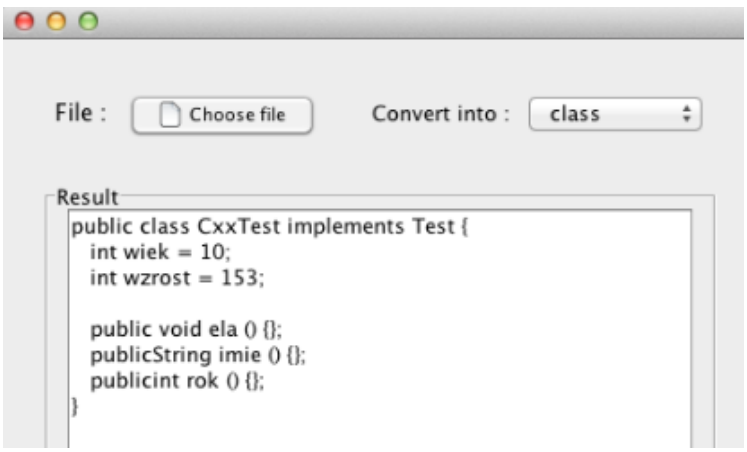
Dla danych wejściowych:

```
22  package com.piotrek.model.clas;
23
24  public interface Test {
25
26    // Attributes
27    int wiek = 10;
28    int wzrost = 153;
29
30    // Methods
31    public void ela();
32    public String imie();
33    public int rok();
34
35  }
```

Nasze parser utworzy następujące drzewo:



A otrzymany wynik zostanie wyświetlony w oknie programu:



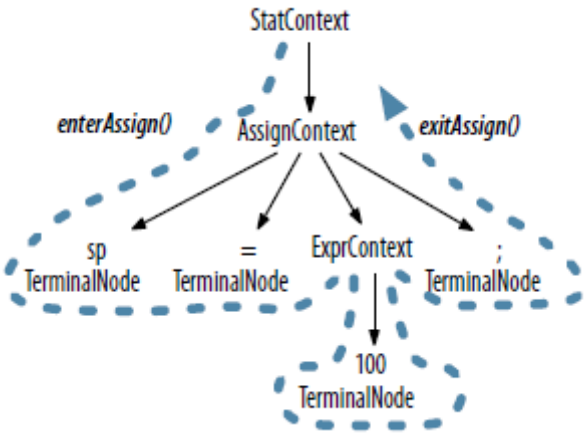
Opis translacji:

Zarówno w przypadku translacji to interfejsu jak i do klasy wykorzystujemy wewnętrzy mechanizm ANTLRa - **tree walker**.

Umożliwia on ma automatyczne przejście po drzewie, bez potrzeby pisania kodu samemu. Dzięki temu programista może skupić się na tym co chce osiągnąć a nie na tym jak to osiągnąć.

Dodatkowo mechanizm przechodzenia jest w stanie przy wjeściu/opuszczeniu każdego węzła wywołać metody które zostały wygenerowane przez parser a znajdują się w **parse-tree *listener* interface.**

Przykładowy schemat wywoływania metod przy wchodzeniu i opuszczaniu węzła



Mechanizm ten umożliwia znam dostęp do każdego elementu opisanego przez naszą gramatykę. W przypadku prosty translacji i modyfikacji ten mechanizm jest wystarczający. Dane wyjściowe „wyrzucane" są przez każdą metodę za pomocą instrukcji print.

Jednak w przypadku bardziej skompilkowanych translacji same printy nie wystarczają i znależy posłużyć się dodatkowym mechanizmem antlra jakim jest StringTemplate. Jest to swego rodzaju string z 'dziurami" które zostają stopniowo wypełniane. Jest to na tyle elastyczny mechanizm, ze umożliwia on translacje całych jezyków, stron www itd.

Jego użycie jest jednak bardziej czasochłonne oraz podatne na błędy. Mechanizm ten wykorzystaliśmy w naszym trzecim rodzaju tłumaczenia z Pascala do Javy.

Świetny opis użycia StringTemplate jest dostępny na

Link : http://cs.indhu.in/2009/09/automated-code-translation-using-antlr.html

Translacja Pascala:

Jako dane wejściowe translator otrzymuje plik *.pas. Celem translacji jest przekształcenia programu napisanego w Pascalu do Javy. W

Dla danych wejściowych:

```
1  program test;
2
3  const
4    x = 69;
5    y = 100;
6
7  var
8
9    i : integer;
10   test : real;
11   n : integer;
12
13 begin
14   i := 0;
15   n := 10;
16   w1 := 1.0;
17
18   repeat
19     begin
20         test := test + 1.15;
21         i:= i+1;
22     end
23   until i=n
24
25   if n == 10 then
26     writeln(n);
27     writeln();
28 end.
```

Wynik: