



# POLSKO-JAPOŃSKA WYŻSZA SZKOŁA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

**Katedra Inżynierii Oprogramowania**

Inżynieria Oprogramowania i Baz Danych

**Emil Wcisło**

Nr albumu 3122

## **Integracja obiektowego języka zapytań z językiem programowania Java**

Praca magisterska napisana  
pod kierunkiem:

dr inż. Piotr Habela

Warszawa, maj 2010

## Abstrakt

Niniejsza praca magisterska jest próbą rozbudowy popularnego języka programowania Java o obiektowy, mocny algorytmicznie, przyjazny dla użytkownika język zapytań. Język ten, stworzony w oparciu o przełomową dla języków zapytań koncepcję podejścia stosowego, realizuje przetwarzanie danych ulotnych w pamięci operacyjnej. Stanowi on odpowiedź na cieszący się uznaniem język LINQ, wprowadzony do języków programowania platformy Microsoft .NET. Zakończona sukcesem implementacja posiada również właściwości niedostępne w języku LINQ. Wśród nich należy wspomnieć o uniwersalności i minimalności składni, możliwości przetwarzania dowolnych obiektów Java, mocnej kontroli typologicznej w czasie kompilacji i wysokiej wydajności ewaluacji zapytań. W czasie tworzenia projektu została opracowana i zaimplementowana koncepcja bezszwowej integracji dwóch języków o odmiennej składni, semantyce i naturze działania. Zostały również stworzone unikalne rozwiązania wśród języków zapytań opartych na podejściu stosowym. Do nich należą: translacja zapytań na wysokowydajne natywne operacje środowiska wykonawczego Java, uniwersalny, przyjazny użytkownikowi operator sortowania, oraz generyczność kolekcji. Te oraz inne zaimplementowane użyteczne cechy pozwalają wykorzystać stworzony język zapytań jako rozwiązanie wielu problemów związanych z przetwarzaniem danych.

## Spis treści

Abstrakt.....	2
Spis treści .....	3
1 Wprowadzenie.....	8
2 Stan sztuki w momencie pisania pracy.....	10
2.1 LINQ.....	10
2.1.1 Opis .....	10
2.1.2 Krytyka .....	10
2.2 JaQuE .....	11
2.2.1 Opis .....	11
2.2.2 Krytyka .....	11
2.3 Queare.....	11
2.3.1 Opis .....	11
2.3.2 Krytyka .....	11
3 Język programowania Java .....	13
3.1 Wstęp .....	13
3.2 Model obiektowy .....	13
3.2.1 Klasy .....	13
3.2.2 Pakiety.....	14
3.2.3 Interfejsy .....	14
3.2.4 Dziedziczenie.....	14
3.2.5 Hermetyzacja .....	14
3.2.6 Kolekcje .....	14
3.2.7 Generyczność .....	15
3.2.8 Referencje .....	15
3.3 Cechy środowiska wykonawczego programów Java.....	16
3.3.1 Maszyna wirtualna .....	16
3.3.2 Automatyczne odśmiecanie pamięci .....	16
3.3.3 Refleksja .....	17
4 Podstawy teoretyczne budowy obiektowego języka zapytań .....	18
4.1 Podstawowe założenia .....	18
4.2 Modele danych w SBQL.....	18

4.2.1	Model AS0.....	19
4.2.2	Model AS1.....	20
4.2.3	Model AS2.....	20
4.2.4	Model AS3.....	20
4.3	Stos QRES .....	20
4.4	Stos ENVs i funkcja <i>nested</i> .....	20
5	Podstawowe problemy związane z rozszerzeniem języka Java o język zapytań.....	22
5.1	Niezgodność impedancji .....	22
5.1.1	Składnia.....	22
5.1.2	System typów.....	22
5.1.3	Semantyka i paradygmaty języków .....	23
5.1.4	Poziomy abstrakcji .....	23
5.1.5	Przetwarzanie kolekcji .....	24
5.2	Zasada korespondencji.....	24
5.3	Rozpoznawanie typów Java .....	25
5.4	Kompatybilność wsteczna .....	26
6	Opis języka SBQL4J .....	27
6.1	Główne założenia języka .....	27
6.2	Integracja z językiem Java .....	27
6.2.1	Motywacje .....	27
6.2.2	Zgodność modeli danych .....	28
6.2.3	Integracja składniowa .....	28
6.3	Architektura rozwiązania .....	29
6.3.1	Ogólny schemat architektury.....	29
6.3.1.1	Faza preprocesingu.....	29
6.3.1.2	Faza kompilacji.....	30
6.3.1.3	Faza uruchomienia.....	30
6.3.2	Budowa drzewa składni zapytania.....	32
6.3.3	Statyczna analiza zapytań .....	33
6.3.3.1	Motywacje .....	33
6.3.3.2	Ewaluacja statycznej analizy zapytań .....	33
6.3.3.3	Sygnatury zapytań .....	34

6.3.3.3.1	ValueSignature .....	34
6.3.3.3.2	BinderSignature .....	35
6.3.3.3.3	StructSignature .....	35
6.3.3.3.4	MethodSignature .....	35
6.3.3.3.5	ClassSignature .....	35
6.3.3.3.6	ConstructorSignature .....	36
6.3.3.3.7	PackageSignature .....	36
6.3.3.4	Kontrola typologiczna .....	36
6.3.4	Translacja zapytań SBQL4J na wyrażenia w języku Java .....	36
6.3.4.1	Tryb interpretera .....	36
6.3.4.2	Generacja kodu Java .....	37
6.3.4.2.1	Wariant z kodem tożsamym z kodem interpretera .....	38
6.3.4.2.2	Wariant z kodem bez stosu QRES .....	38
6.3.4.2.3	Wariant z wczesnym wiązaniem nazw z kodem bez stosów ENVŚ i QRES	38
6.4	Model danych JAS0 w SBQL4J .....	39
6.4.1	Opis ogólny .....	39
6.4.2	Obiekt .....	39
6.4.3	Referencja .....	39
6.4.4	Klasa .....	40
6.4.5	Metoda .....	40
6.4.6	Konstruktor .....	40
6.5	Opis operatorów języka .....	40
6.5.1	Operatory algebraiczne .....	40
6.5.1.1	Operatory porównania .....	40
6.5.1.2	Operatory porównania zakresowego .....	41
6.5.1.3	Operatory logiczne .....	42
6.5.1.4	Operator negacji logicznej .....	43
6.5.1.5	Operator sprawdzenia typu .....	43
6.5.1.6	Operatory arytmetyczne .....	44
6.5.1.7	Operator tworzenia bagu .....	45
6.5.1.7.1	Wariant z domyślną implementacją bagu .....	45

6.5.1.7.2	Wariant z wyborem implementacji bagu .....	45
6.5.1.8	Operator tworzenia sekwencji .....	46
6.5.1.8.1	Wariant z domyślną implementacją sekwencji .....	46
6.5.1.8.2	Wariant z wyborem implementacji sekwencji .....	47
6.5.1.9	Operator sumy .....	47
6.5.1.10	Operator zliczenia .....	48
6.5.1.11	Operator średniej wartości .....	48
6.5.1.12	Operator minimalnej wartości .....	49
6.5.1.13	Operator maksymalnej wartości .....	49
6.5.1.14	Operator egzystencjalny .....	50
6.5.1.15	Operator sumy zbiorów .....	50
6.5.1.16	Operator przecięcia zbiorów .....	51
6.5.1.17	Operator różnicy zbiorów .....	51
6.5.1.18	Operator zawierania się zbiorów .....	52
6.5.1.19	Operator iloczynu kartezjańskiego .....	52
6.5.1.20	Operator zakresowy .....	53
6.5.1.20.1	Wariant z wyborem pojedynczego elementu .....	53
6.5.1.20.2	Wariant z dolnym zakresem .....	53
6.5.1.20.3	Wariant z dolnym i górnym zakresem .....	54
6.5.1.21	Operator nazwy pomocniczej AS .....	54
6.5.1.22	Operator nazwy pomocniczej GROUP AS .....	55
6.5.1.23	Operator warunkowy .....	55
6.5.2	Operatory niealgebraiczne .....	56
6.5.2.1	Operator selekcji warunkowej .....	56
6.5.2.2	Operator nawigacji .....	57
6.5.2.3	Operator zależnego złączenia .....	58
6.5.2.4	Operator kwantyfikatora uniwersalnego .....	58
6.5.2.5	Operator kwantyfikatora egzystencjalnego .....	59
6.5.2.6	Operator sortowania .....	60
6.5.2.7	Operator tranzytywnego domknięcia .....	62
6.5.2.8	Operator tworzenia obiektu .....	62
6.5.2.8.1	Wariant z konstruktorem bez parametrów .....	62

6.5.2.8.2	Wariant z konstruktorem z parametrami.....	63
7	Kierunki rozwoju języka SBQL4J .....	65
7.1	SBQL4J jako uniwersalne API do baz danych i mechanizmów wyszukiwania .....	65
7.1.1	Wariant tłumaczenia wyrażeń SBQL4J na natywny język zapytań bazy.....	65
7.1.2	Wariant natywnych zapytań bazy połączonych z wyrażeniami SBQL4J z mocną kontrolą typologiczną – podzapytania niezależne semantycznie .....	65
7.2	Zapytania <i>ad-hoc</i> i kontrola typologiczna w czasie wykonania programu .....	66
7.3	Optymalizacja zapytań .....	66
7.4	Zintegrowane środowisko programistyczne (IDE) dla SBQL4J.....	66
8	Wykorzystane narzędzia.....	67
8.1	Skaner JFlex .....	67
8.2	Parser Cup .....	67
8.3	Kompilator OpenJDK .....	67
8.4	Środowisko programistyczne Eclipse .....	67
8.5	Biblioteka CGLib .....	68
8.6	Biblioteka ASM .....	68
8.7	Biblioteka Jalopy.....	68
8.8	Narzędzie skryptowe ANT .....	68
8.9	Serwer SVN.....	68
8.10	Witryna GoogleCode .....	69
9	Podsumowanie .....	70
	Bibliografia .....	71
	Spis ilustracji .....	72
	Dodatek A: Słownik użytej terminologii i skrótów.....	73
	Dodatek B: Słowa kluczowe języka SBQL4J .....	74
	Dodatek C: Gramatyka języka SBQL4J .....	75
	Dodatek D: Porównanie zapytań SBQL4J z zapytaniami w języku LINQ.....	81

## 1 Wprowadzenie

Po sukcesie rynkowym jakim było wprowadzenie języka zapytań LINQ dla platformy Microsoft .NET, powstało zapotrzebowanie na podobne koncepcyjnie rozwiązanie dla przemysłu informatycznego skupionego wokół technologii Java. Korzyści, jakie daje możliwość deklaratywnego przetwarzania danych w ramach języka programowania, stały się powszechnie docenione. Wielu naukowców i inżynierów związanych z obiektowymi bazami danych zaczęło postrzegać LINQ jako rozwiązanie integracyjne między językami programowania a bazami danych. Miało ono uprościć komunikację z bazami danych przy zachowaniu mocnej kontroli typów i zgodności z oryginalnym językiem programowania. Jednocześnie pojawiły się krytyczne opinie na temat enigmatycznej, nieortogonalnej składni LINQ, ograniczonej mocy algorytmicznej języka, niskiej wydajności w przetwarzaniu danych ulotnych i brak podstaw teoretycznych do budowy uniwersalnych optymalizatorów zapytań. Wątpliwości budził także sens tworzenia takiego interfejsu, gdy nie istniała żadna baza danych używająca LINQ jako języka zapytań.

W międzyczasie pojawiły się projekty usiłujące w mniej lub bardziej nieudolny sposób wprowadzić zapytania *a'la* LINQ do Javy. Autorzy projektów takich jak *Quaere*, *JaQue* czy *QueryDSL* swoją uwagę skupili na sposobie wyrażenia zapytań za pomocą standardowej składni języka Java. Nie zdawali oni sobie najwyraźniej sprawy z głębokiej różnicy między imperatywną naturą Javy a deklaratywnym kontekstem języków zapytań. Brak podstaw teoretycznych i odpowiednich środków technicznych użytych do ich budowy był przyczyną zupełnego niepowodzenia. Skutkiem tych działań były implementacje języków z enigmatyczną składnią, nawet z porównaniu z oryginalną implementacją LINQ. We wspomnianych rozwiązaniach zupełnie zaniedbano również takie kluczowe aspekty języka zapytań jak: uniwersalność, przyjazność dla użytkownika, brak lukru syntaktycznego, minimalność, mocna kontrola typów, modularność i precyzyjna semantyka, a przede wszystkim wydajność.

Przystępując do tworzenia implementacji będącej podstawą niniejszej pracy, autor zdawał sobie sprawę z korzyści jakie niesie rozszerzenie języka programowania o konstrukcje zapytań. Biorąc udział w dużych projektach informatycznych w roli programisty i projektanta, autor wielokrotnie spotkał się z koniecznością zaawansowanego przetwarzania skomplikowanych struktur danych w pamięci operacyjnej programu. Te wymagania były zazwyczaj realizowane za pomocą dużych bloków kodu zawierających pętle, instrukcje warunkowe, instrukcje tworzenia pomocniczych obiektów, przepisywania danych do tymczasowych kolekcji itp. Ponieważ taki kod często był rozwijany latami przez różnych programistów, stawał się on z czasem mało czytelny i uciążliwy w pielęgnacji. W celu poprawienia tych niedogodności powstał projekt SBQL4J, będący praktyczną realizacją niniejszej pracy magisterskiej. Jest on rozszerzeniem języka Java o konstrukcje języka zapytań, w podobny sposób jak zostało to zrealizowane w LINQ. Wzorowany na języku SBQL, SBQL4J nie posiada wymienionych

zasadniczych wad LINQ, jego składnia jest minimalna, ortogonalna i przyjazna dla użytkownika. Jego podstawą jest podejście stosowe opracowane przez prof. Kazimierza Subietę i dostosowane przez autora do zadań języka programowania ogólnego przeznaczenia, jakim jest język Java. Język ten stanowi udaną implementację abstrakcyjnego języka SBQL przy zachowaniu wysokiej wydajności i pełnej kompatybilności z szeroko używanym środowiskiem języka Java.

## 2 Stan sztuki w momencie pisania pracy

Niniejszy rozdział zawiera opis istniejących rozwiązań polegających na integracji języka zapytań z istniejącym wcześniej językiem programowania. Pokazane zostaną ich zasadnicze cechy i możliwości oraz krytyczne podsumowanie każdego z wymienionych projektów.

### 2.1 LINQ

#### 2.1.1 Opis

LINQ powstał w laboratoriach firmy Microsoft jako rozszerzenie języków platformy .NET. Jego głównym autorem jest Anders Hejlsberg, twórca min. języka C# i Delphi. Podstawową funkcjonalnością LINQ są zapytania na obiektach języka programowania. Posiada składnię z charakterystycznym lukrem SQL. Przykładem zapytania LINQ jest:

```
from p in products
select p.ProductName;
```

Język LINQ posiada ok. 37 operatorów, są to min. operatory selekcji, projekcji, op. agregujące, operator grupowania i inne. LINQ posiada rozszerzenia umożliwiające przeprowadzanie zapytań na dokumentach XML i relacyjnych bazach danych.

#### 2.1.2 Krytyka

Operatory języka w wielu przypadkach reprezentują zbliżone, nieortogonalne funkcjonalności, np. `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending` wprowadzające funkcjonalność sortowania w kolejności rosnącej i malejącej. Składania języka zawiera wiele elementów lukru syntaktycznego; np. powyższe zapytanie zawiera 8 elementów składniowych. Analogiczne zapytanie w SBQL wygląda następująco:

```
products.ProductName
```

Zawiera ono jedynie 3 elementy składniowe, a więc jest znacznie bardziej czytelne i łatwiejsze w pielęgnacji. Podejście zastosowane w LINQ utrudnia budowanie bardziej skomplikowanych zapytań, a w niektórych przypadkach całkowicie to uniemożliwia. Kolejnym problemem jest ograniczenie uniwersalności zapytań na obiektach. LINQ zakłada, że argumentem zapytania może być tylko obiekt implementujący interfejs `IEnumerable`, tymczasem język zapytań powinien mieć możliwość przetwarzania dowolnych obiektów. Język LINQ nie posiada mocnych podstaw teoretycznych pozwalających na budowanie uniwersalnych optymalizatorów zapytań. Język SBQL dzięki zastosowaniu podejścia stosowego znacznie wyprzedza go w tym aspekcie. Następną istotną kwestią jest wydajność przetwarzania zapytań. Mimo, że zapytania LINQ są ostatecznie tłumaczone na standardowe operacje oryginalnego języka programowania, odbywa się to z dodatkowymi kosztami. Według niektórych testów wydajności zapytania LINQ są nawet 2847 razy wolniejsze (Soni) w stosunku do optymalnego kodu napisanego bez użycia języka zapytań.

## 2.2 JaQuE

### 2.2.1 Opis

Jest to API do języka Java stworzone przez Konstantina Trigera. Biblioteka ta jest oparta na wyrażeniach *closures*, które mają być wprowadzone w przyszłych wydaniach Javy. Zapytania JaQuE mogą posiadać kilka podstawowych operatorów, takich jak selekcja, projekcja, grupowanie, operatory arytmetyczne i agregujące. Przykładowe zapytanie wygląda następująco:

```
from(data,
      where( { Integer i => i > 5 },
            orderBy( { Integer i1, Integer i2 => i1-i2 },
                    select( { Number i => 4 } ))));
```

### 2.2.2 Krytyka

Podstawowym błędem twórcy języka było oparcie się na nieistniejącej obecnie technologii w języku Java, w związku z czym JaQuE jest niemożliwy do użycia w stabilnym środowisku. Na krytykę zasługuje też zagmatwana składnia wyjątkowo przesłodzona lukrem syntaktycznym. Powyższe zapytanie realizuje bardzo prostą funkcjonalność selekcji z prostym warunkiem *where* i sortowaniem. Mimo to zapytanie zawiera aż 41 elementów składniowych. Takie zapytanie można wyrazić znacznie prościej i czytelniej w SBQL:

```
data as i
where i > 5
orderby i
```

## 2.3 Queare

### 2.3.1 Opis

Queare jest kolejną próbą wprowadzenia zapytań do języka Java. Jej autorzy położyli główny nacisk na wyrażenie zapytań za pomocą standardowej składni Javy. Przykładowe zapytanie:

```
from("n").in(numbers).
where(lt("n",5).
select("n");
```

Queare korzysta z właściwości importów statycznych i enumeracji obecnych w Javie. Dostępnych jest kilka operatorów typu projekcja, selekcja, grupowanie, sortowanie.

### 2.3.2 Krytyka

Ponieważ deklaratywna natura języka zapytań znacznie różni się od imperatywnej natury języka Java, wprowadzenia konstrukcji zapytań za pomocą standardowej składni odbyło się kosztem innych ważnych aspektów języka zapytań. Queare nie zapewnia mocnej kontroli typologicznej w czasie kompilacji, ponieważ nazwy używane w zapytaniach są

przekazywane jako obiekt typu *String*, a więc mogą być rozwiązane dopiero, gdy będzie znana wartość tych obiektów, czyli w czasie uruchomienia programu. Takie podejście implikuje szereg innych negatywnych cech. Niemożliwa jest statyczna analiza zapytań, a więc również ich optymalizacja przez przepisywanie. Podobnie niewykonalne jest tłumaczenie zapytania na optymalny kod środowiska wykonawczego programu, gdyż również do tego potrzebna jest analiza zapytania na możliwie wczesnym etapie. Błędem twórców Queare było skupienie się na wyrażeniu zapytań bez zmian w składni i semantyce języka Java, gdyż za pomocą tego typu prostych „sztuczek” nie da się zasypać głębokich różnic między językiem programowania i zapytań.

## 3 Język programowania Java

Implementacja, będąca podstawą tej pracy, polega na integracji języka Java z obiektowym językiem zapytań. Stąd też, do wyjaśnienia kluczowych aspektów tego przedsięwzięcia konieczne jest opisanie języka Java i jego właściwości. Niniejszy rozdział zawiera opis tego języka, streszczenie historii powstania i rozwoju, jego główne cechy i opis modelu danych.

### 3.1 Wstęp

Pierwsze wydanie platformy Java zostało opublikowane w 1995 roku, stworzone w laboratoriach firmy Sun Microsystems. Głównym autorem pierwszej wersji i następnych wydań był James Gosling. Od początku Java posiadała doceniane właściwości języka, takie jak prostota składni, mocna kontrola typologiczna i możliwość uruchomienia raz skompilowanego kodu na niemal dowolnej maszynie (*"Write Once, Run Anywhere"*). Twórcy Javy w wielu aspektach wzorowali się na języku C++, odrzucając z niego te właściwości, które często prowadziły do skomplikowania kodu i trudnych do zdiagnozowania problemów. Dzięki temu język zdobył ogromną popularność w wielu segmentach rynku informatycznego, począwszy od aplikacji na urządzenia przenośne (Java Micro Edition – JavaME), aplikacje desktopowe (Java Standard Edition – JavaSE) i aplikacje w architekturze klient-serwer (Java Enterprise Edition – JavaEE). Java stała się standardem, którego rozwój od 1998 roku przebiega za pomocą *Java Community Process (JCP)*. W ramach tego procesu powstają propozycje nowych cech języka (*Java Specification Requests – JSRs*). Propozycje te są ostatecznie poddawane pod głosowanie przez *JCP Executive Committee* w skład którego wchodzi eksperci i przedstawiciele zainteresowanych firm informatycznych. W ten sposób rozwój języka Java jest uporządkowany i zaplanowany, mimo że platforma pozostaje w dużym stopniu otwartym oprogramowaniem. W 2007 roku firma Sun Microsystems opublikowała kod źródłowy implementacji referencyjnej języka i środowiska wykonawczego Java. Wydarzenie to umożliwiło powstanie niniejszego projektu, gdyż korzysta on w dużym stopniu z kodu kompilatora Java.

### 3.2 Model obiektowy

Java jest językiem zorientowanym obiektowo, jest to jedyny styl pisania programów w tym języku. Stanowi to wyraźną różnicę w porównaniu do innych języków, np. C++, gdzie programista mógł wybrać między proceduralnym i obiektowym stylem pisania programów.

#### 3.2.1 Klasy

Miejszem definicji każdego z obiektów jest klasa. Jest to zbiór wszystkich wspólnych cech dla danej grupy obiektów. Definicja klasy zawiera opis struktury danego obiektu, będący zbiorem odwołań do innych obiektów, zwanych referencjami, posiadającymi określone nazwy. Klasa zawiera również listę metod, które można wywołać na danym obiekcie.

Klasa jest identyfikowana przez nazwę, prefiksowaną nazwą pakietu, do której ona należy.

### 3.2.2 Pakiety

Pakiety to hierarchiczna struktura w której umieszczone są definicje klas i interfejsów programu. Umożliwia uporządkowaną organizację kodu programu.

### 3.2.3 Interfejsy

Interfejs jest dodatkowym środkiem definicji obiektów. W odróżnieniu od klasy, w ramach definicji interfejsu można opisać jedynie listę nagłówek metod, które dany obiekt powinien implementować. Interfejs nie określa jednak implementacji metod, które powinny zostać zawarte w definicji klasy, która dziedziczy dany interfejs.

### 3.2.4 Dziedziczenie

Java dopuszcza dziedziczenie jednokrotne klasy z innej klasy. Klasa dziedzicząca przejmuje zarówno strukturę, jak i metody klasy dziedziczonej. Możliwe jest także wielokrotne dziedziczenie interfejsów przez klasę. Jest to namiastka wielokrotnego dziedziczenia klas, dostępnego np. w języku C++, przy uniknięciu skutków ubocznych, takich jak konflikty nazw struktury obiektu.

### 3.2.5 Hermetyzacja

Jest to mechanizm ukrywania wybranych elementów deklaracji klasy przez jej użytkownikami. Składniki deklaracji klasy, taki jak definicje pól i metod mogą posiadać odpowiedni modyfikator dostępu: *public*, *private* lub *protected*. Pierwszy z nich oznacza pełny dostęp klienta do danego elementu, *private* pozwala na dostęp jedynie z wnętrza deklarowanej klasy, natomiast *protected* umożliwia dostęp z wnętrza danej klasy oraz klas po niej dziedziczących. Brak modyfikatora dostępu oznacza hermetyzację w ramach pakietu. Wybrany składnik deklaracji klasy staje się wtedy dostępny z wnętrza wszystkich klas zgrupowanych w ramach tego samego pakietu.

Enkapsulacja pozwala ukryć przed użytkownikiem klasy te elementy, do których nie powinien mieć dostępu. Przykładem zastosowania jest ochrona przed tworzeniem zewnętrznych powiązań z składnikami klasy, które mogą ulec zmianie w trakcie rozwoju oprogramowania.

### 3.2.6 Kolekcje

W Javie typy masowe mogą mieć postać tablic lub kolekcji. Tablica jest typem wbudowanym o stałym rozmiarze. Kolekcja jest standardowym obiektem mogącym przechowywać nieokreśloną ilość innych obiektów. Pakiet *java.util* będący częścią standardowej biblioteki Javy zawiera interfejsy definiujące podstawowe typy kolekcji, takie jak *Collection*, *Set*, *Queue*, *List*, *Map*. Dostępnych jest wiele implementacji wymienionych interfejsów, co sprawia, że są one elastycznym narzędziem dającym się dostosować do konkretnych potrzeb.

### 3.2.7 Generyczność

Od wersji JavaSE 5 możliwe jest parametryzowanie typów. Jest to narzędzie podobne do mechanizmu szablonów z C++ umożliwiające tworzenie bardziej uniwersalnego kodu. Pozwala stworzyć klasę parametryzowaną typem, który jest określany dopiero w momencie użycia. Mechanizm ten jest najczęściej wykorzystywany podczas użycia kolekcji. Dzięki temu można określić typ obiektów, które ma przechowywać dana instancja kolekcji z kontrolą podczas kompilacji. Przykład użycia:

```
List<Student> studentList = new ArrayList<Student>();
studentList.add(new Student("Adam", "Kowalski", "s9835"));
Student s = studentList.get(0); //brak konieczności rzutowania do typu Student
```

### 3.2.8 Referencje

Obiekty w Javie są dostępne dla programisty przez referencje. Są to wskaźniki logiczne identyfikowane w kodzie programu przez nazwę. Dogłębne zrozumienie mechanizmu referencji w Javie jest szczególnie istotne dla określenia istotnych cech języka zapytań, takich jak nazwy obiektów i powiązania między nimi. Należy zwrócić uwagę, że w Javie same obiekty nie posiadają takiej cechy jak nazwa; dany obiekt może być dostępny przez wiele referencji o różnych nazwach. Referencje są definiowane w określonym kontekście programu (deklaracja klasy, metody, bloku kodu) i tylko w tym kontekście mogą być użyte. W przypadku, gdy występuje konieczność przekazania obiektu do innego, rozłącznego kontekstu, wartość referencji jest kopiowana. W nowym kontekście dany obiekt jest dostępny pod inną referencją, mogącą mieć inną nazwę niż oryginalna.

Przykładem takiej zmiany kontekstu jest wywołanie metody na innym obiekcie, niż obiekt obejmujący blok kodu w którym znajduje się wywołanie.

```
public class Pracownik extends Osoba {
    private String stanowisko;

    public void setStanowisko(String stan) {
        this.stanowisko = stan;
    }
    //...
}
```

Powyższa definicja klasy zawiera pole typu *String* o nazwie *stanowisko*, oraz metodę wypełniającą wartość tego pola – *setStanowisko*. Parametrem tej metody jest referencja do obiektu typu *String*, nazwana *stan*. Jedyną operacją metody jest przepisanie wartości referencji *stan* na inną referencję, będącą polem obiektu o nazwie *stanowisko*. Poniżej przedstawiony jest kod wywołania tej metody.

```
Pracownik prac = new Pracownik();
String s = "programista";
prac.setStanowisko(s);
```

W tym przykładzie tworzony jest nowy obiekt klasy `Pracownik`, następnie przypisany do referencji o nazwie `prac`. Następnie tworzony jest obiekt typu `String` o nazwie `s`, po czym wywoływana jest metoda obiektu `prac` `setStanowisko` z obiektem `s` jako parametrem. Na tym prostym przykładzie widać, że nazwa obiektu w Javie może zmienić się wielokrotnie w zależności od kontekstu użycia. W tym przypadku obiekt typu `String` o wartości „programista” występuje aż pod trzema nazwami – jako zmienna lokalna o nazwie `s`, parametr metody `stan`, oraz pole obiektu `stanowisko`.

Z powyższego wynika, że nazwa obiektu w Javie nie jest jego cechą, lecz własnością kontekstu w którym jest on użyty. Jest to udogodnienie zwiększające elastyczność języka programowania, gdyż w przeciwnym wypadku nazwanie obiektu wprowadziłoby dodatkową zależność we wszystkich kontekstach, w których mógłby być użyty. Tę cechę należy zapamiętać jako istotną przy określaniu abstrakcyjnego modelu danych języka zapytań.

### 3.3 Cechy środowiska wykonawczego programów Java

#### 3.3.1 Maszyna wirtualna

W odróżnieniu od wielu innych języków programowania, kod języka Java jest kompilowany do postaci pośredniej, tzw. kodu bajtowego (*bytecode*), który w czasie działania programu jest interpretowany przez specjalny program, maszynę wirtualną Java. Dzięki temu raz skompilowany program może być uruchomiony na dowolnej platformie posiadającej maszynę wirtualną Java. Kosztem takiego rozwiązania jest niższa wydajność, niż w przypadku kompilacji programu do kodu maszynowego. Aby zminimalizować tę stratę, maszyny wirtualne Java są wyposażone w kompilatory *just-in-time (JIT)*. Ich zadaniem jest kompilacja kodu pośredniego do kodu maszynowego bezpośrednio przed jego wykonaniem.

#### 3.3.2 Automatyczne odśmiecanie pamięci

Java nie posiada operatorów, które explicite niszczą struktury danych i zwalniają zajęta przez nie pamięć operacyjną. Obiekty Javy są automatycznie usuwane przez mechanizm odśmiecania pamięci (*garbage collector*). Jego działanie polega na sprawdzeniu, czy każdy z obiektów jest dostępny przez referencje z obiektów, o których wiadomo że mogą być użyte na ścieżce wykonania kodu. Te obiekty, które nie spełniają tego warunku są usuwane i zajęta przez nie pamięć jest zwalniana. Ten mechanizm mimo ogromnej zalety automatycznego zarządzania pamięcią bez ingerencji programisty, ma również wady. W środowisku Java najważniejszą z nich jest brak przewidywalności, kiedy mechanizm odśmiecania zostanie uruchomiony. Na czas jego wykonanie programu musi być zawieszony, co znacznie utrudnia użycie Javy w pewnych zastosowaniach, jak np. systemy czasu rzeczywistego.

### 3.3.3 Refleksja

Java posiada mechanizm odzyskiwania informacji o typie w czasie wykonania programu (*ang. run-time type information, RTTI*). W ten sposób możliwe jest uzyskanie informacji o strukturze i metodach danego obiektu. Poza tym możliwy jest generyczny dostęp do wartości pól obiektu i tworzenie dynamicznych wywołań metod. Umożliwia to tworzenie bardzo uniwersalnego kodu programu, jednak odbywa się to kosztem wydajności. Dostęp do właściwości obiektu przez refleksję jest kilka- do kilkudziesięciu razy wolniejszy niż przez standardowe wywołanie.

## 4 Podstawy teoretyczne budowy obiektowego języka zapytań

Niniejszy rozdział zawiera podstawowe informacje na temat koncepcji podejścia stosowego (Stack-Based Approach - SBA) będącej podstawą implementacji języka zapytań SBQL4J. Znajdują się też tu informacje dotyczące abstrakcyjnego języka zapytań opartego na SBA – Stack-Based Query Language (SBQL). Zarówno SBA jak i SBQL zostały szerzej opisane w książce (Subieta, 2004).

### 4.1 Podstawowe założenia

Główną osią podejścia stosowego jest założenie, że języki zapytań są odmianą języków programowania. W związku z tym, koncepcje i pojęcia rozpowszechnione w językach programowania mają swoje odzwierciedlenie w językach zapytań. Ze względu na ich uniwersalność, możliwe jest stworzenie języka zapytań operującego na dowolnym modelu danych.

### 4.2 Modele danych w SBQL

Język SBQL definiuje cztery abstrakcyjne modele danych, zróżnicowane pod względem złożoności i możliwości ekspresji bytów świata rzeczywistego. Są to modele o nazwach: AS0, AS1, AS2, AS3. Każdy kolejny z nich rozwija uprzedni o nowe cechy, wprowadzając nowe sposobności modelowania.

Cechami wspólnymi wymienionych modeli są:

- Relatywizm obiektów – zasada ta mówi, że obiekt złożony może składać się z pod-obiektów, które są bytami tej samej kategorii, co obiekt nadrzędny. Obiekty podrzędne mogą być traktowane samodzielnie na dowolnym poziomie hierarchii podrzędności.
- Zasada wewnętrznej identyfikacji – zgodnie z nią, każdy byt programistyczny, który może być użyty niezależnie od innych, posiada własny, unikalny identyfikator. Stosując się do tej zasady, każdy obiekt posiada wewnętrzny identyfikator, nie mający odzwierciedlenia w modelowanym bycie świata rzeczywistego. Służy on wyłącznie do określania tożsamości obiektów w pamięci komputera. Jest on nadawany automatycznie przez system zarządzania składem obiektów, nie może być użyty bezpośrednio w zapytaniach.
- Zewnętrzna nazwa obiektów – jest to cecha nadająca nieformalną semantykę obiektowi. Umożliwia dostęp do niego za pomocą zapytań. Nazwa jest nadawana przez użytkownika, np. programistę, projektanta lub administratora (w przypadku systemów zarządzania bazami danych). Nazwa nie musi być unikalna.
- Wartość atomowa – niepodzielna z punktu widzenia użytkownika. Może być nią np. liczba (rzeczywista lub złożona), łańcuch znaków, wartość logiczna, obiekt binarny itp.

We wszystkich modelach występują następujące zbiory:

- $I$  – wewnętrzne identyfikatory obiektów
- $N$  – zewnętrzne nazwy obiektów
- $V$  – wartości atomowe

#### 4.2.1 Model ASO

W modelu ASO występują trzy rodzaje obiektów:

- Obiekt atomowy – można go opisać uporządkowaną trójką  $\langle i, n, v \rangle$ , gdzie:
  - $i$  jest identyfikatorem,
  - $n$  jest nazwą zewnętrzną,
  - $v$  jest wartością atomową.

i4732, pensja, 8960.50

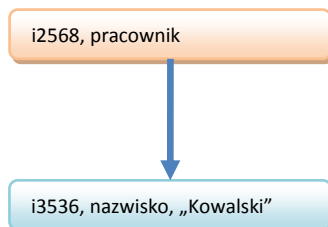
Rysunek 1 - Przykładowy obiekt atomowy

- Obiekt złożony  $\langle i, n, T \rangle$ , gdzie:
  - $i$  jest identyfikatorem,
  - $n$  jest nazwą zewnętrzną,
  - $T$  jest zbiorem dowolnych obiektów



Rysunek 2 - przykładowy obiekt złożony

- Obiekt referencyjny  $\langle i, n, i_2 \rangle$ , gdzie:
  - $i$  jest identyfikatorem,
  - $n$  jest nazwą zewnętrzną,
  - $i_2$  jest pointerem (referencją) do innego obiektu



Rysunek 3 - przykładowy obiekt referencyjny

Skład modelu ASO definiuje para uporządkowana  $\langle S, R \rangle$ , gdzie:

- $S$  jest zbiorem obiektów
- $R$  jest zbiorem identyfikatorów obiektów głównych (korzeniowych)

### 4.2.2 Model AS1

Model AS1 jest rozszerzeniem AS0 o pojęcia klasy i dziedziczenia między klasami. Klasy są rodzajem obiektów przechowujących wspólne cechy stałe dla danej grupy obiektów. Takimi inwariantami mogą być np. definicja struktury obiektu, metody, nazwa typu, pola statyczne. Inną cechą modelu AS1 jest możliwość zdefiniowania relacji dziedziczenia między klasami. Dopuszczalne jest dziedziczenie wielokrotne.

Skład modelu AS1 definiuje czwórka  $\langle S, R, KK, OK \rangle$ , gdzie:

- $S$  jest zbiorem obiektów
- $R$  jest zbiorem identyfikatorów obiektów głównych (korzeniowych)
- $KK \subseteq I \times I$  jest relacją wyznaczającą związki dziedziczenia między klasami
- $OK \subseteq I \times I$  jest relacją wyznaczającą przynależność obiektów do klas

### 4.2.3 Model AS2

Model AS2 rozszerza AS1 o pojęcia dziedziczenia między obiektami. Jest ono możliwe, na podobnej zasadzie jak dziedziczenie między klasami. Wystąpienie takiego związku w terminologii języka SBQL nazywa się rolą.

Skład modelu AS1 definiuje uporządkowana piątka  $\langle S, R, KK, OK, OO \rangle$ , gdzie:

- $S$  jest zbiorem obiektów
- $R$  jest zbiorem identyfikatorów obiektów głównych (korzeniowych)
- $KK \subseteq I \times I$  jest relacją wyznaczającą związki dziedziczenia między klasami
- $OK \subseteq I \times I$  jest relacją wyznaczającą przynależność obiektów do klas
- $OO \subseteq I \times I$  jest relacją wyznaczającą związki dziedziczenia między obiektami

### 4.2.4 Model AS3

Nową cechą modelu AS3 jest pojęcie hermetyzacji. Do jej realizacji został wprowadzony nowy inwariant dla klas – *lista eksportowa*. Zawiera ona nazwy *publicznych* składowych klasy i obiektów, mających być dostępnymi z zewnątrz. Jest to uproszczony mechanizm hermetyzacji znany z popularnych języków programowania, takich jak Java.

## 4.3 Stos QRES

Stos rezultatów (Query Result Stack - QRES) jest odpowiedzialny za tymczasowe przechowywanie rezultatów zapytań. Są to zarówno wyniki pośrednie, jak i rezultat końcowy zapytania. Stos QRES nie jest niezbędny do działania języka zapytań, poprawia on nieco czytelność implementacji języka zapytań. W projekcie SBQL4J nie występuje w niektórych trybach przetwarzania zapytań, ze względu na optymalizację wydajności przetwarzania.

## 4.4 Stos ENVs i funkcja *nested*

Stos środowiskowy (Environmental Stack – ENVs) jest strukturą danych reprezentującą środowisko przetwarzania programu podczas uruchomienia. Koncepcja stosu

środowiskowego jest powszechnie znana i używana w większości języków programowania. Składa się on z sekcji, z których każda reprezentuje kontekst środowiska czasu wykonania (tzw. rekord aktywacji). Ich kolejność jest istotna. Każda sekcja zawiera zbiór *binderów* łączących nazwy z bytami programistycznymi. Stos ENVS spełnia następujące zadania:

- Wiązanie nazw z bytami programistycznymi czasu wykonania
- Kontrola zakresu nazw
- Przechowywanie dynamicznych bytów programu (np. parametry aktualnie przetwarzanych metod, ślady powrotu z funkcji, zmienne pomocnicze przy przetwarzaniu operatorów)

Stos ENVS posiada swój odpowiednik wykorzystywany przed uruchomieniem programu, podczas analizy statycznej. Zasady jego działania są identyczne, choć jest on wykorzystywany do innych celów.

Funkcja *nested* określa „wnętrze” obiektu. Jej użycie powoduje podwyższenie stosu ENVS o nową sekcję zawierającą *bindery* uzyskane z określonego bytu czasu wykonania, który był argumentem funkcji *nested*.

W projekcie SBQL4J został zaimplementowany zarówno stos ENVS jak i jego statyczny odpowiednik. Jednak nie występuje on w czasie wykonania zapytań w jednym z trybów języka. W celu poprawienia wydajności, w tym trybie jego zadania spełnia natywny stos środowisk języka Java.

## 5 Podstawowe problemy związane z rozszerzeniem języka Java o język zapytań

W tym rozdziale zostaną przedstawione najważniejsze problemy dotyczące integracji języka Java z nowopowstałym językiem zapytań. Problemy te zostały zdiagnozowane i w znakomitej większości rozwiązane podczas prac projektowych i implementacyjnych rozwiązania. Najczęściej wynikają one z odmienności charakteru obydwu rozwiązań mających stanowić całość. Są wśród nich również problemy techniczne, wynikające ze specyfiki środowiska Java. Poniższy rozdział zawiera definicję tych problemów, oraz propozycje ich rozwiązania zastosowane w implementacji omawianego projektu.

### 5.1 Niezgodność impedancji

Tym terminem określa się problemy wynikające z zanurzenia języka zapytań w język programowania. Przykładem takiego użycia są zapytania SQL do relacyjnych baz danych zanurzone w język programowania Java. Niezgodności te zostały opisane w (Subieta, 2004) Problemy te dotyczą następujących aspektów:

#### 5.1.1 Składnia

W przypadku zanurzonych języków zapytań programista musi używać dwóch różnych składni i dostosowywać się do odmiennych stylów pisania kodu. Wymusza to wydłużony czas uczenia się programisty.

Rozwiązaniem tego problemu jest możliwe zbliżenie składni języka zapytań do rozszerzanego języka programowania. Ponieważ język zapytań tworzony jest od podstaw, istnieje możliwość dopasowania składni operatorów mających swoje odpowiedniki w oryginalnym języku programowania. W ten sposób programista jest w stanie intuicyjnie używać dobrze znanych sobie elementów języka w nowym środowisku. W zaimplementowanym języku zapytań dotyczy to takich operatorów jak: arytmetyczne (+, -, \*, /, %), porównania (==, !=), porównania zakresowego (>, >=, <, <=), logiczne (!, &&, ||), tworzenia obiektu (*new*), porównania typu (*instanceof*). W podobny sposób do języka Java odbywa się wywołanie metod i dostęp do atrybutów obiektu. W tym rozwiązaniu poznania wymagają nowe operatory języka, takie jak *where*, *join*, *order by* i inne, jednak stanowią one wartość dodaną do środowiska tworzenia oprogramowania, a więc motywacja do ich nauki powinna być wystarczająca.

#### 5.1.2 System typów

Problem ten występuje np. przy próbie łączenia relacyjnych baz danych z obiektowym językiem programowania. Oba te światy posiadają swoje systemy typów, zupełnie odmienne od siebie. W bazie danych występuje typ relacja, którego nie ma w obiektowych językach programowania. I na odwrót, jako parametru zapytania nie da się bezpośrednio przekazać złożonych obiektów języka programowania, które nie mają odpowiednika w SQL.

Problem ten nie występuje w przypadku języka zapytań tworzonego dla konkretnego języka programowania. W stworzonej implementacji został wykorzystany oryginalny system typów języka Java, co umożliwia kompatybilność wsteczną programów. Zarówno parametrami zapytań jak i ich wynikami są standardowe obiekty Java, zdefiniowane zgodnie z ich niezmiennym systemem typologicznym. Należy zwrócić uwagę, że zastosowane podejście przy tworzeniu języka zapytań (*Stack-Based Approach*) traktuje dane na wyższym poziomie abstrakcji niż konkretny system typologiczny. W istocie system typów Javy został potraktowany jako pewna odmiana ogólnego modelu ASO, dzięki czemu możliwe jest pełne wykorzystanie możliwości płynących z zastosowania SBA.

### 5.1.3 Semantyka i paradygmaty języków

Problem dotyczy odmiennej koncepcji języków. Języki zapytań są z reguły deklaratywne, a więc kod napisany w takim języku wyznacza cel, a nie środki potrzebne do jego uzyskania. Inaczej jest w językach programowania, takich jak Java. Kod takiego języka jest ciągiem instrukcji, a więc bazuje na koncepcji imperatywnej. Problemem jest bezszwowe połączenie obydwu stylów.

Podchodząc do tego problemu autor wyszedł z założenia, że zapytanie, jako całość, jest szczególnym przypadkiem wyrażenia imperatywnego. Najbliższym jego odpowiednikiem w semantyce Javy jest wywołanie funkcji. Funkcja ta może przyjmować dowolną liczbę parametrów i zwracać dowolny wynik. Podobnie, w zapytaniu zanurzone w języku programowania możemy użyć dowolnych danych z kontekstu języka programowania, tak jak może ono zwrócić dowolny wynik. Zasadnicza różnica polega na sposobie definiowania funkcji i zapytania. O ile język programowania nie dopuszcza konstrukcji funkcji anonimowych (takich jak wyrażenia lambda w języku C#), musi być ona zdefiniowana przed jej wywołaniem. Składa się ona zazwyczaj z szeregu wyrażeń imperatywnych, podobnie jak inne konstrukcje języka programowania. Inaczej jest w przypadku zapytań, które mogą być zdefiniowane w miejscu użycia. Rozwiązaniem tego problemu jest translacja zapytania na wywołanie wygenerowanej funkcji, która zawiera instrukcje konieczne do uzyskania pożądanego wyniku, określonego w tekście zapytania. Dzięki takiemu podejściu możliwe jest bezszwowe łączenie obydwu koncepcji języków.

### 5.1.4 Poziomy abstrakcji

Ze względu na deklaratywny styl, języki zapytań są konstrukcjami, których wyrażenia są na wysokim poziomie abstrakcji. Programista piszący zapytanie nie musi zajmować się pisaniem algorytmów prowadzących do zamierzonego celu, abstrahuje od wielu szczegółów implementacyjnych i organizacyjnych dotyczących przetwarzanych danych. Z kolei programowanie w języku Java wymaga zwrócenia uwagi na wiele więcej szczegółów, ale też oferuje większe możliwości przetwarzania danych. W pracy (Subieta, 2004) prof. Subieta zwraca uwagę na problem, który pojawia się, gdy występuje konieczność dostępu do danych zarówno przez wyrażenia języka zapytań, jak i języka

programowania. Problem ten dotyczy izolacji użytkownika języka zapytań od szczegółów implementacyjnych znajdujących się „pod spodem”.

Odpowiedzią na ten problem jest dbałość o elastyczność języka zapytań osiągnięta przez zastosowanie generycznych operatorów, oraz możliwość użycia w zapytaniach standardowych konstrukcji języka programowania. Przykładem generycznego operatora jest operator `==`, gdzie algorytm porównujący obiekty jest zdefiniowany w samych obiektach porównywanych w standardowej metodzie *equals*, którą posiada każdy obiekt Java. Innymi przykładami są zaimplementowane operatory porównania zakresowego, korzystają one z metody *compare*, zdefiniowanej w standardowym interfejsie *java.lang.Comparable*, który jest wymagany dla obiektów podlegających porównaniu zakresowemu. Inną ważną cechą języka zapytań jest możliwość mieszania konstrukcji języka zapytań z wyrażeniami zdefiniowanymi w języku programowania, takimi jak wywołania funkcji, metod i konstruktorów obiektów. W wielu przypadkach funkcja, która konsumuje wynik zapytania może zastąpić operator języka zapytań, którego implementacja wymagałaby zmian składniowych i semantycznych w samym języku. Wymienione środki pozwalają wykorzystać synergię wynikającą z połączenia dwóch odmiennych od siebie języków.

### 5.1.5 Przetwarzanie kolekcji

Szczególnym przypadkiem różnicy w poziomach abstrakcji języków są kolekcje. W języku Java mają one postać standardowych obiektów. Ich dodatkowe własności, takie jak możliwość użycia ich jako argumentu operatora iterującego *for*, wyznacza odpowiedni interfejs który muszą implementować. Przetwarzając kolekcje w języku Java programista musi używać takich środków jak pętle czy specjalne obiekty – iteratory. Mimo tych utrudnień takie rozwiązanie gwarantuje dużą elastyczność, dzięki możliwości dopasowania implementacji kolekcji do aktualnych wymagań. Inaczej wygląda to w języku zapytań, takim jak SBQL, gdzie semantyka kolekcji nie jest tożsama z semantyką obiektu. Zastosowany wysoki poziom abstrakcji traktuje licznosc obiektów jako ich dodatkową cechę, podobnie jak typ zastosowanej kolekcji w przypadku licznosci większej od 1. Programista języka zapytań nie buduje explicite wyrażen iteracji po kolekcjach. Są one wbudowane w semantykę operatorów języka zapytań, takich jak selekcja, projekcja i złączenie.

Ponieważ zapytania są ostatecznie tłumaczone na szereg wyrażen imperatywnych, jest możliwie przejście z niższego poziomu abstrakcji na wyższy. Na podstawie informacji o typie obiektu można jasno określić zasady rozpoznawania kolekcji w mechanizmie zapytań.

## 5.2 Zasada korespondencji

Cechą istotną podczas rozszerzania istniejącego języka o nowe własności jest zasada korespondencji. W książce (Subieta, 2004) została ona zdefiniowana następująco:

***Zasada korespondencji mówi, że wraz z wprowadzeniem do języka pewnej cechy X należy precyzyjnie określić inne relewantne cechy języka w taki sposób, aby cecha X współdziałała z już istniejącymi konstrukcjami, została wtopiona w istniejące lub zmodyfikowane mechanizmy nazywania, typowania, zakresu i wiązania oraz miała zapewnioną uniwersalną obsługę.***

W naszym przypadku problem staje się niebagatelny, jeśli uświadomimy sobie jego skalę. Nowopowstały język zapytań liczy sobie ok. 35 operatorów, natomiast gramatyka języka Java składa się z ponad 240 nie-terminali<sup>1</sup>. Nietrudno sobie wyobrazić, jak skompilowałoby składnię języka bezpośrednie włączenie nowych funkcjonalności do składni Javy. Każdy z nowych operatorów musiałby mieć jasno zdefiniowany mechanizm współdziałania z każdą istniejącą konstrukcją języka Java. Znacząco skomplikowałoby to rozszerzony język i wymusiłoby dłuższy czas nauki.

Rozwiązaniem tego problemu jest koncepcyjne rozdzielenie składni języka zapytania od składni Javy. Zapytania powinny być zdefiniowane w oddzielnym kontekście od pozostałych wyrażeń języka. W ten sposób całe zapytanie jest nowym elementem języka programowania, co znacząco zmniejsza ilość nowych powiązań. Jak już wspomniano, z punktu widzenia semantyki Javy zapytanie jest specjalnym przypadkiem wywołania funkcji. Przy tych założeniach zapytanie może użyte wszędzie tam, gdzie jest możliwe jego wstawienie.

### 5.3 Rozpoznawanie typów Java

Wspomnieliśmy wcześniej, że do zanurzenia języka zapytań konieczna jest informacja o typach języka programowania. Aby zapewnić kontrolę typologiczną oraz wykorzystać mechanizmy optymalizacyjne dla zapytań należy posiadać informację o wszystkich typach obiektów użytych w zapytaniu. Jednak niuanse implementacyjne języka Java powodują pewne problemy w osiągnięciu tego celu. W przypadku wczesnego wiązania konieczna jest analiza kodu programu w którym wykorzystane są zapytania, oraz wszystkich źródeł i bibliotek zależnych. Do tego celu niezbędny jest mechanizm kompilatora Java, z którego można uzyskać wszystkie te informacje w momencie analizowania zapytania. Samodzielna implementacja kompilatora Java jest zadaniem o bardzo dużym nakładzie pracy, wystarczy wspomnieć, że oryginalny kompilator firmy Sun Microsystems zbudowany jest z kodu o objętości ponad 200 tysięcy linii.

Zaimplementowane rozwiązanie korzysta w całości z wczesnego wiązania nazw. Typy obiektów użytych w zapytaniu są rozpoznawane przez mechanizm kompilatora Java zintegrowanego z analizatorem zapytań. Kompilator ten jest modyfikacją oryginalnego kompilatora firmy Sun Microsystems udostępnionego na licencji open-source. Dzięki temu zapewniona jest mocna kontrola typów na wczesnym etapie, oraz możliwe okazało się zastosowanie skutecznej optymalizacji zapytań, min. dzięki tłumaczeniu ich do

---

<sup>1</sup> Taką złożoność ma gramatyka języka Java w wersji 5 wyrażona w składni generatora parserów CUP (Ananian, 2002)

natywnego kodu Javy. Jednak aby w przyszłości umożliwić analizę zapytań w trakcie wykonania programu, będą potrzebne dodatkowe środki techniczne. Aby analizować zapytania tworzone w trakcie działania programu będzie konieczne rozszerzenie mechanizmu kontroli typologicznej o współpracę z jednym z mechanizmów RTTI (runtime type information) – np. z refleksją Java.

#### 5.4 Kompatybilność wsteczna

Rozszerzenie języka programowania o nowe cechy wymaga szczególnych zmian w składni i semantyce tego języka. Problem pojawia się, jeśli zmiany te dotyczą wszystkie fazy tworzenia oprogramowania. Na szczególnie narażone na koszty są zmiany w środowisku uruchomienia programu. W przypadku zmian w samej maszynie wirtualnej Java tracona jest zgodność z istniejącymi standardami, szeroko zaakceptowanymi we współczesnym przemyśle informatycznym. W ten sposób wzrasta znacznie ryzyko projektowe związane z użyciem rozszerzonego języka, co może prowadzić do zniechęcenia dużej części architektów oprogramowania i programistów.

Zastosowane w implementacji rozwiązanie pozostawia niezmienione dotychczasowe fazy tworzenia oprogramowania, tj. fazę kompilacji i uruchomienia. Kod zawierający zapytania jest przetwarzany w nowej fazie poprzedzającej kompilację – fazie preprocesingu. Wynikiem jej działania jest wygenerowany kod Java w 100% kompatybilny z standardowymi kompilatorami, a więc możliwy do uruchomienia w istniejących środowiskach wykonawczych Java.

## 6 Opis języka SBQL4J

Język SBQL4J jest językiem zapytań, zbudowanym w oparciu o stosową architekturę przetwarzania zapytań (SBA – Stack Based Architecture). Składnią i semantyką przypomina on SBQL – abstrakcyjną implementację języka zapytań opartego na SBA. Niniejszy rozdział zawiera opis architektury rozwiązania, opis składni i semantyki języka oraz głównych aspektów działania kontrolera typów, interpretera i generatora kodu.

### 6.1 Główne założenia języka

Podczas prac implementacyjnych były brane pod uwagę następujące założenia projektowe:

- Składnia i semantyka języka będą podobne do składni i semantyki języka SBQL
- Składnia będzie posiadać następujące cechy: minimalność, ortogonalność, relatywizm, uniwersalność, bezpieczeństwo
- Semantyka będzie sterowana składnią
- Język będzie używał mocnej kontroli typów i późnego wiązania, dla celów optymalizacyjnych będzie zastosowane wczesne wiązanie tam, gdzie jest to możliwe
- Jednorodne traktowanie danych (typy proste, obiekty)
- Przetwarzane dane będą obiektami języka Java
- Przetwarzanie będzie uwzględniało cechy języka Java (dziedziczenie, polimorfizm, hermetyzacja)
- Wynik zapytań będzie wystąpieniem typu języka Java
- W zapytaniach będzie możliwe użycie istniejących metod i funkcji języka Java
- Język zapytań będzie działać w sposób wydajny, w celu optymalizacji możliwie największa ilość podstawowych operacji nowego języka będzie natywnymi operacjami środowiska wykonawczego języka Java
- Będą dostępne niektóre operatory imperatywne, np. konstruktory obiektów
- Brak konstrukcji definiujących typy danych (klasy, metody)
- Brak cech charakterystycznych dla systemów zarządzania bazami danych (trwałość obiektów, transakcje, izolacja zasobów, optymalizacja przez indeksy itp.)

### 6.2 Integracja z językiem Java

W odróżnieniu od wielu innych implementacji SBQL, SBQL4J nie jest samodzielnym językiem zapytań. Stanowi on rozszerzenie języka programowania Java i może on być użyty tylko z wyrażeniami tego języka.

#### 6.2.1 Motywacje

Celem autora było stworzenia języka zapytań, który możliwie najpełniej uzupełnia imperatywne konstrukcje języka Java. Między nimi zachodzi synergia – Java posiada bogaty zestaw bibliotek i jest szeroko obecna we współczesnym przemyśle

informatycznym. Z kolei SBQL4J umożliwia pisanie zapytań, których implementacja za pomocą standardowych konstrukcji języka Java byłaby znacznie dłuższa i bardziej skomplikowana. Co za tym idzie – byłaby ona mniej przejrzysta i trudniejsza w pielęgnacji. SBQL4J pozwala przy tym korzystać z bogactwa bibliotek, narzędzi i *frameworków* stworzonych dla języka Java, co stawia jego praktyczną użyteczność na bardzo wysokim poziomie, niedostępnym dla nowych języków działających w zupełnie nowym środowisku. Dzięki takiemu podejściu SBQL4J jest przyjazny dla szerokiego grona użytkowników – programistów i architektów – dla których użycie go wiąże się ze znacznie mniejszym ryzykiem projektowym niż zastosowanie nowego, akademickiego języka programowania.

### 6.2.2 Zgodność modeli danych

W celu zachowania zgodności z Javą, SBQL4J nie definiuje własnego modelu danych. Przetwarzanie zapytań opiera się o model danych języka Java, który jest traktowany jako pewien wariant modelu AS0. Z naukowego punktu widzenia jest to pewne ograniczenie, ponieważ nie pozwala wykorzystać nowatorskich cech zastosowanych w modelach danych SBQL, takich jak np. dynamiczne role. Przystępując do pracy autor zdawał sobie sprawę z tego ograniczenia, jednak zalety płynące z bezszwowej integracji z wiodącym językiem programowania przekonały go o sensowności takiego rozwiązania.

### 6.2.3 Integracja składniowa

Podczas implementacji, autor przyjął założenie, że składnia Javy i SBQL4J powinny zostać rozdzielone w dwóch rozłącznych kontekstach. Oto przykład fragmentu kodu z użyciem zapytania SBQL4J:

```
Integer[] n = new Integer[] {4, 2, 6, 20, 4, 23, 1, 5, 7};
List<Integer> result = #{ n where self > 7 };
System.out.println(result);
```

Kolorem niebieskim oznaczono kod zgodny ze standardową składnią języka Java. Kod w kolorze czerwonym reprezentuje zapytanie SBQL. Widać, że zapytanie jest wydzielone z kontekstu kodu Javy za pomocą znaku hash (#) i klamer otaczających tekst zapytania.

Do takiego rozwiązania, polegającego na rozdzieleniu kontekstów Java i SBQL4J, skierowały autora następujące powody:

- Imperatywna natura wyrażeń języka Java znacznie różni się od deklaratywnej natury zapytań SBQL4J, a co za tym idzie, operatory o takiej samej składni (np. operator kropki) mają w tych językach zupełnie inną semantykę.
- Zgodnie z zasadą korespondencji każdy nowy element języka musi mieć określone współdziałanie z wszystkimi dotychczasowymi elementami. W przypadku bezpośredniego włączenia zapytań do składni Javy oznaczałoby to konieczność definiowania takiej relacji z każdym elementem zapytania, co wprowadziłoby niepotrzebne zamieszanie i utrudniłoby rozwój języka. Z punktu widzenia składni

Javy całe zapytanie jest nowym elementem językowym, co zmniejsza ilość nowych powiązań.

- Aby umożliwić swobodny rozwój języka, należy mieć swobodę w definiowaniu jego składni. Wpisanie składni języka w nowy kontekst niezależny od dotychczasowej składni stanowi rozwiązanie tego problemu.

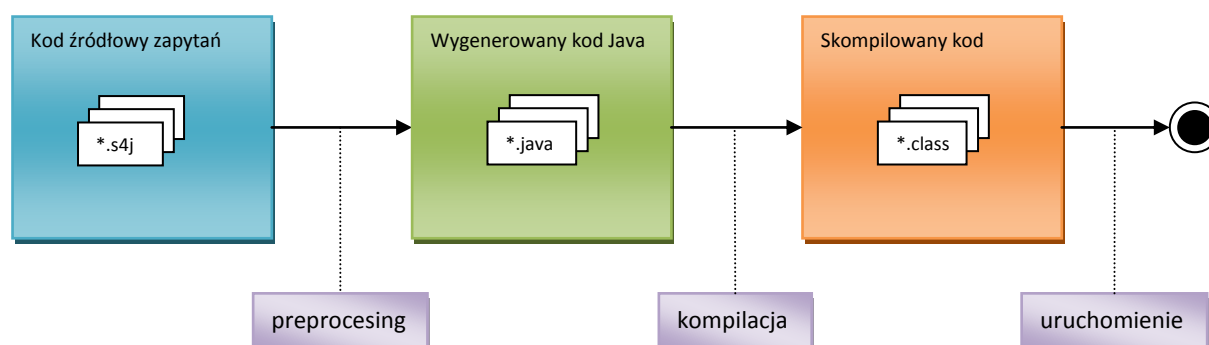
Powyższe rozwiązanie zakłada, że zmienne z kontekstu Javy mogą być bezpośrednio użyte w kontekście zapytania. Niepotrzebne są dodatkowe środki umożliwiające przekazywanie parametrów *explicite*, które zastosowano np. w zapytaniach SQL z użyciem bibliotek JDBC.

### 6.3 Architektura rozwiązania

Celem tego podrozdziału jest opisanie głównych założeń działania języka SBQL4J, jak również jego najważniejszych modułów.

#### 6.3.1 Ogólny schemat architektury

Rysunek 4 przedstawia fazy tworzenia działającego programu z użyciem zapytań:



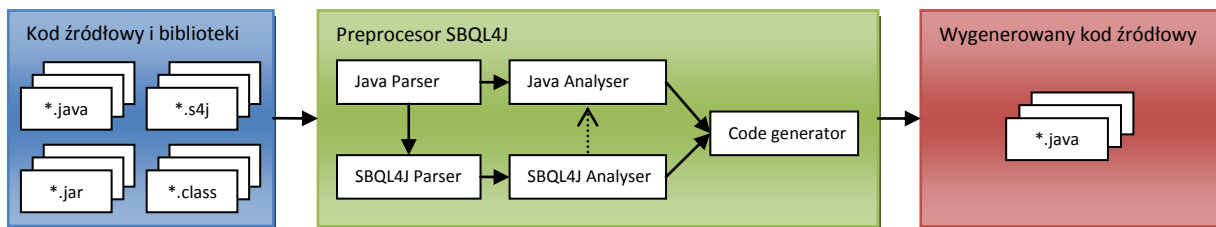
Rysunek 4 - architektura działania języka SBQL4J

Jak widać, proces tworzenia programu jest bardzo podobny do tworzenia programu w standardowych językach programowania. W istocie, program z zapytaniami SBQL4J różni się od zwykłego programu w języku Java tylko dodatkową fazą preprocesingu. Pozostałe fazy - kompilacji i uruchomienia są takie same dla Javy i SBQL4J. Co więcej, program z zapytaniami może być skompilowany za pomocą dowolnego kompilatora Java i uruchomiony w dowolnym środowisku maszyny wirtualnej Java (o ile są one zgodne z wersją Java 6, lub nowszą). Dzięki temu udało się zachować kompatybilność z istniejącymi standardami przemysłowymi, co znacząco wpływa na praktyczną użyteczność języka SBQL4J.

##### 6.3.1.1 Faza preprocesingu

W ramach tej fazy, kod źródłowy zapytań oraz kod programu Java zostaną przeanalizowane i zostaną wychwycone błędy składniowe i semantyczne. Początkowo kod jest analizowany przez nieco zmodyfikowany kompilator Javy. Różni się on od standardowego kompilatora tym, że rozpoznaje szczególny typ wyrażień, jakim są

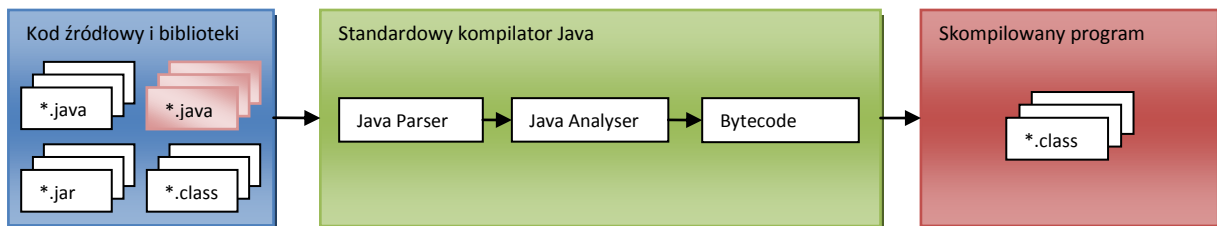
zapytania SBQL4J. Dla każdego z zapytań zostanie utworzony i uruchomiony kontroler typów (*type checker*), który sprawdzi ich poprawność. Kontroler ten współpracuje z analizatorem kompilatora Javy, dzięki czemu może określić poprawność nazw i typów pochodzących z kontekstu Javy, użytych w zapytaniu. W związku z tym, w fazie preprocesingu biorą udział zarówno pliki źródłowe z zapytaniem (z rozszerzeniem .s4j), jak również wszystkie źródła i biblioteki które są potrzebne do ich poprawnej kompilacji. Zaletą tego rozwiązania jest wczesne wykrycie błędów popełnianych przez programistę – zarówno błędów w samych zapytaniach, jak również błędów kompilacji w kontekście kodu Javy. Kontroler typów określi również typ wyniku zapytania, o ile on istnieje. Wynikiem końcowym tej fazy jest wygenerowany kod źródłowy w języku Java. W miejscu zapytań znajdą się standardowe wyrażenia tego języka, które w środowisku uruchomieniowym wykonają ewaluację zapytania.



Rysunek 5 - opis działania preprocesora

### 6.3.1.2 Faza kompilacji

Jak wspomniano wcześniej, ta faza nie różni się niczym od standardowej kompilacji kodu języka Java. Na wejściu znajdują się pliki źródłowe programu wraz ze źródłami wygenerowanymi w fazie preprocesingu. Są one powtórnie parsowane i analizowane, a wynikiem tego procesu jest skompilowany kod w postaci plików z rozszerzeniem .class zawierających *bytecode* przeznaczony do uruchomienia w maszynie wirtualnej Java.



Rysunek 6 - opis kompilacji kompilatorem Java

W tej fazie w trakcie działania programu zapytania są uruchamiane i przetwarzają konkretne dane. W zależności od rodzaju wygenerowanego kodu przetwarzanie to może odbywać się w różny sposób:

- W trybie INTERPRETER zapytanie przekazywane jest w postaci napisu. Jest ono ponownie parsowane i uruchamiany jest interpreter, który pracuje na drzewie składniowym stworzonym przez parser. Obiekty i klasy Java, które w fazie analizy zostały oznaczone jako biorące udział w zapytaniu, są inicjalnie przekazywane do

interpretera. Podczas ewaluacji zapytania zajmują najniższą sekcję stosu ENVS i dzięki temu mogą być wykorzystane.

W trybie generowania kodu zapytanie jest tłumaczone na szereg wyrażeń w języku Java. Odbywa się to w fazie analizy, a więc w czasie uruchomienia nie ma potrzeby ponownego rozbioru składniowego zapytania. Dla każdego z zapytań generowana jest odpowiednia klasa, której konstruktor przyjmuje obiekty Java biorące udział w zapytaniu. Zawiera ona metodę *executeQuery()* której wywołanie uruchamia przetwarzanie zapytania i ostatecznie zwraca wynik. Sygnatura tej metody zależy od typu rezultatu zapytania, który jest określany w fazie analizy. Tak więc zapewniona jest pełna kontrola typologiczna – w fazie analizy sprawdzana jest zarówno poprawność zapytania, jak też dokładnie określany jest typ rezultatu.

Kod wygenerowany w wyniku translacji zapytania może być różny w zależności od trybu wybranego przez użytkownika:

- W trybie SIMPLE wygenerowany kod odpowiada wywołaniom w trybie interpretera. Wygenerowana klasa zawiera obiekty reprezentujące stosy ENVS i QRES. Zaletą w stosunku do trybu interpretera jest brak konieczności parsowania tekstu zapytania w trakcie działania programu, co skutkuje większą wydajnością. Drugą zaletą jest mniejsze zużycie pamięci wynikające z faktu, że kod interpretera znajduje się w jednej metodzie i nowe sekcje na stosie maszyny wirtualnej Java otwierane są stosunkowo rzadko. Jest to usprawnienie w stosunku do interpretera, gdzie przetwarzanie każdego elementu drzewa składni zapytania odbywa się w oddzielnym wywołaniu metody. W przypadku bardziej skomplikowanych zapytań może prowadzić do znacznego podwyższenia stosu maszyny wirtualnej i w konsekwencji większego zapotrzebowania na pamięć i mniejszej ogólnej wydajności programu.
- Tryb NO\_QRES jest podobny do trybu SIMPLE. Różni się przede wszystkim brakiem stosu QRES. Wyniki podzapytań są przechowywane w nowozainicjalizowanych zmiennych. Skutkuje to nieco większą wydajnością i przejrzystością wygenerowanego kodu. Dodatkowym usprawnieniem jest mniejsza ilość instrukcji warunkowych. Dotyczy to min. sytuacji gdy należy zamienić pojedynczy wynik na kolekcję jednoelementową i odwrotnie. Ponieważ informacja o liczności wyników podzapytań wyliczana jest w fazie analizy, wygenerowany kod wykonuje automatyczną koercję tylko tam, gdzie jest to konieczne.
- W trybie NO\_STACKS zastosowano najbardziej wyrafinowane techniki optymalizacyjne, co powoduje, że jest on najbardziej wydajnym trybem wykonania zapytań SBQL4J. Zostało to osiągnięte przez dużo lepsze wykorzystanie natywnego środowiska wykonawczego Javy do ewaluacji zapytania. W uprzednio opisanych trybach obiekty Java są opakowane przez

*wrappery*, które zawierają funkcjonalności charakterystyczne dla semantyki obiektów SBQL. Zawierają one min. metodę *nested*, która dla obiektu Javy zwraca listę binderów do atrybutów tego obiektu. Takie rozwiązanie umożliwia min. wiązanie pól i metod obiektu po nazwie w czasie wykonania programu za pomocą stosu ENVS. W trybie NO\_STACKS wiązanie nazw, a dokładnie zamiana nazw użytych w zapytaniu na nazwy z modelu Javy, odbywa się wcześniej, podczas generacji kodu. Dzięki temu dostęp do atrybutów i metod obiektów jest znacznie szybszy, co znacząco wpływa na ogólną wydajność zapytań (wzrasta ona blisko 50-krotnie w stosunku do trybu interpretera). Poza tym wygenerowany kod jest krótszy i bardziej czytelny dla użytkownika.

W dalszych podrozdziałach zostaną bardziej szczegółowo opisane techniki generowania kodu.

### 6.3.2 Budowa drzewa składni zapytania

Podczas analizy zapytań oraz uruchomienia w trybie interpretera zapytanie jest reprezentowane w programie w postaci drzewa składniowego (*abstract syntax tree* – *AST*). Do jego utworzenia ze źródła tekstowego zapytania został wykorzystany parser utworzony przy użyciu generatora parserów *CUP*. Jest to generator typu *LALR* (*Look Ahead Left to right, identifying the Rightmost production*). Współpracuje on ze skanerem utworzonym za pomocą biblioteki *JFlex* – generatorem skanerów dla języka Java.

Dla budowy drzewa składni zapytania przyjęto następujące założenia:

- Każdy operator języka lub podzapytanie jest reprezentowany przez element drzewa
- Każdy element drzewa będzie obiektem klasy dziedziczącej po abstrakcyjnej klasie *Expression*
- Obiekty elementów drzewa będą posiadać referencje do elementu nadrzędnego, oraz elementów bezpośrednio podrzędnych
- Operatory języka wywołujące metodę *nested* dla co najmniej jednego ze swoich argumentów będą podklasami *NestedExpression*
- Operatory języka cechujące się unikalną semantyką w stosunku do innych wyrażeń będą reprezentowane przez osobne podklasy *Expression* (np. operatory *where, join, order by*)
- Operatory języka cechujące się podobną semantyką będą reprezentowane przez wspólną podklasę *Expression*, zawierającą obiekt klasy dziedziczącej po abstrakcyjnej klasie *Operator* będący dyskryminatorem typu wyrażenia (np. operatory arytmetyczne)
- Każdy element drzewa będzie zawierał obiekt abstrakcyjnej klasy *Signature* reprezentujący informacje gromadzone podczas analizy zapytań

- Elementy drzewa będą zawierać indeks wskazujący na początek tekstowej reprezentacji danego wyrażenia w tekście zapytania

### 6.3.3 Statyczna analiza zapytań

Statyczna analiza zapytania polega na uzyskaniu informacji o zapytaniu na podstawie jego treści i opisu modelu danych użytych w zapytaniu (meta-modelu). Ma ona miejsce w fazie preprocesingu, przed właściwą kompilacją programu.

#### 6.3.3.1 Motywacje

Do zastosowanie tej techniki skłoniły autora dwa kluczowe cele:

- Uzyskanie mocnej kontroli typologicznej w trakcie kompilacji  
Dzięki temu użytkownik będzie powiadomiony o błędach typów możliwie wcześniej przed uruchomieniem właściwego programu. Do tego potrzebna jest dokładna analiza zapytania, ze szczególnym uwzględnieniem typów zwracanych przez podzapytania i sprawdzenia ich zgodności z użytymi operatorami. Szerzej ten proces będzie opisany w dalszych rozdziałach.
- Translacja zapytań do optymalnego kodu Javy  
Do realizacji tego celu również konieczny jest dokładny opis zapytania, w szczególności informacje dotyczące nazw wiązanych w trakcie ewaluacji.

#### 6.3.3.2 Ewaluacja statycznej analizy zapytań

Przetwarzanie zapytania rozpoczyna się, gdy kompilator Javy w miejscu, w którym oczekuje wyrażenia, odnajduje tekst zapytania ograniczony klamrami z symbolem hash (`{ ... }`). Na jego podstawie budowane jest drzewo składni, opisane w punkcie 6.3.2. Następnie inicjalizowany jest obiekt typu *JavaNameResolver*. Za jego pomocą mechanizm analizujący zapytanie rozpoznaje byty programistyczne środowiska Javy, dzięki czemu możliwe jest określenie modelu danych użytych w zapytaniu. Obiekt ten jest adapterem mechanizmu rozpoznawania nazw kompilatora Javy. Kolejnym krokiem jest utworzenie obiektu klasy *TypeChecker*, za pomocą którego będzie przeprowadzona analiza.

Klasa *TypeChecker* implementuje wzorzec wizytatora. Jest to wzorzec czynnościowy, jego zastosowanie polega na rozdzieleniu algorytmu od struktury danych na której operuje. Powodem zastosowanie tego wzorca była potrzeba poprawy czytelności oraz modularności kodu. Wzorzec ten ułatwia również dodawanie nowych funkcjonalności, które opierają się na przetwarzaniu drzewa składniowego zapytania. W projekcie zastosowano również inne klasy operujące na tego typu drzewie, które implementują ten wzorzec. Tak działa min. klasa implementująca interpreter. Klasa *TypeChecker* implementuje interfejs *TreeVisitor*, zawierający metody wirtualne *visit[...]* dla każdego z typów elementów drzewa składniowego. Z kolei każdy element drzewa zawiera metodę *accept(TreeVisitor)*, w której następuje wywołanie metody interfejsu *TreeVisitor* odpowiedniej dla danego elementu drzewa. Następnie w metodzie *visit[...]* zostają

wywołane metody *accept(TreeVisitor)* dla elementów podrzędnych w danej gałęzi drzewa, po czym wykonywany jest algorytm funkcjonalny.

Działanie obiektu *TypeChecker* polega na oznaczeniu każdego z elementów drzewa składniowego odpowiednią sygnaturą, oraz sprawdzenie zgodności typów dla każdego z operatorów użytych w zapytaniu. W przypadku znalezienia nieprawidłowości w zapytaniu zostaje zasygnalizowany odpowiedni błąd.

### 6.3.3.3 Sygnatury zapytań

W trakcie statycznej analizy każdy element w drzewie składniowym zapytania zostaje opisany odpowiednią sygnaturą. Podobnie jak obiekt w semantyce SBQL sygnatura posiada metodę *nested*. Zwraca ona listę sygnatur będącymi statycznymi odpowiednikami obiektów zwróconych przez metodę *nested* w czasie wykonania zapytania. Ze względu na różnorodność kontekstów w jakim może być użyty dany element drzewa, wyróżniamy wiele rodzajów sygnatur. Mogą one posiadać również pewną ilość atrybutów wspólnych dla wszystkich typów:

- Nazwa typu rezultatu
- Typ kolekcji
 

Jeśli dany element drzewa zwraca jakiś wynik, jest określony dla niego jeden z następujących typów kolekcji: BAG, SEQUENCE lub ELEMENT. Implikuje on również licznosc wyniku, dla typów BAG i SEQUENCE wynosi ona 0..\*, dla typu ELEMENT jest to 0..1.
- Nazwa rezultatu
 

Jest ona używana podczas generowania kodu. Zazwyczaj określa ona nazwę zmiennej do której przypisany jest wynik danego podzapytania, wtedy musi on być unikalny.

Więcej informacji na temat analizy zapytań z użyciem sygnatur zawiera praca (Stencel, 2006).

Poniżej zaprezentowano typy sygnatur użytych w statycznej analizie zapytań SBQL4J:

#### 6.3.3.3.1 ValueSignature

Jest to statyczny odpowiednik obiektu złożonego. Sygnatura ta reprezentuje wartość, jaką zwraca podzapytanie. Może być to literał, wynik działania operatora lub zmienna z kontekstu Javy. Sygnatura *ValueSignature* zawiera dodatkowe atrybuty:

- Typ obiektu określony przez kompilator Java
- Listę pól publicznych
- Listę metod publicznych
- Flagę określającą, czy dana wartość była oryginalnie użyta jako typ tablicowy w kontekście Javy

Metoda *nested* zwraca listę sygnatur metod i pól.

#### 6.3.3.3.2 BinderSignature

Reprezentuje statyczny binder. Zawiera atrybuty:

- Nazwę
- Sygnaturę wartości *bindera*
- Flagę określającą, czy dany binder został utworzony jako nazwa pomocnicza
- Obiekt *NestedInfo*.

W przypadku, gdy dana sygnatura *bindera* została utworzona w metodzie *nested* innej sygnatury, obiekt ten zawiera informacje potrzebne do odtworzenia operacji wiązania nazwy. Są to:

- Sygnatura z której pochodzi dana sygnatura *bindera*
- Indeks pola struktury w przypadku, gdy sygnatura *bindera* pochodzi z metody *nested* sygnatury struktury
- Element drzewa składniowego zapytania, w którym została wykonana metoda *nested*

Informacje te zostaną wykorzystane podczas generowania kodu Javy, w celu zastąpienia wiązania nazw za pomocą stosu ENVS odpowiednim wyrażeniem zgodnym z semantyką języka Java.

Metoda *nested* zwraca kopię danej sygnatury.

#### 6.3.3.3.3 StructSignature

Reprezentuje statyczną strukturę. Zawiera listę sygnatur reprezentujących pola danej struktury. Metoda *nested* zwraca listę sumę wyników metod *nested* dla wszystkich pól struktury.

#### 6.3.3.3.4 MethodSignature

Reprezentuje metodę Java, którą można wywołać w zapytaniu. Jest rozszerzeniem sygnatury *ValueSignature*. Zawiera dodatkowe atrybuty:

- Nazwę metody
- Klasę będącą źródłem danej metody
- Sygnaturę struktury opisującą parametry danej metody
- Obiekt *NestedInfo* pełniący tę samą funkcję co w przypadku *BinderSignature*

Typ zwracany przez metodę jest opisany przez nadklasę *ValueSignature*.

#### 6.3.3.3.5 ClassSignature

Reprezentuje klasę Java określoną w kodzie jako literał. Należy odróżnić sytuację, w której klasa występuje w tekście programu jako literał od sytuacji, gdzie jest użyta jako nośnik informacji o typie w czasie wykonania (run-time type information, RTTI) jako

obiekt typu *java.lang.Class*. W tym drugim przypadku jest zwyczajnym obiektem Java, reprezentowanym podczas analizy zapytań przez *ValueSignature*. Przykłady:

```
new java.util.Date() //ClassSignature
Date.valueOf("2000-01-01") //ClassSignature
Date.class //ValueSignature
Class.forName("java.util.Date") //ValueSignature
```

Sygnatura *ClassSignature* jest rozszerzeniem *ValueSignature*. Podobnie jak ona, zawiera listę pól i metod, jednak są to jedynie statyczne inwarianty. Oprócz tego zawiera też listę konstruktorów właściwych dla danej klasy. Metoda *nested* zwraca listę sygnatur wspomnianych statycznych metod i pól oraz konstruktorów.

#### 6.3.3.3.6 ConstructorSignature

Reprezentuje konstruktor obiektu. Rozszerza *ValueSignature*. Podobnie jak *MethodSignature* zawiera sygnaturę struktury reprezentującą listę parametrów oraz obiekt *NestedInfo* gromadzący informacje o pochodzeniu konstruktora.

#### 6.3.3.3.7 PackageSignature

Reprezentuje pakiet Javy.

### 6.3.3.4 Kontrola typologiczna

Sprawdzenie zgodności typów polega na ustaleniu dla każdego operatora użytego w zapytaniu czy jego argumenty posiadają odpowiednie sygnatury. W przeciwnym wypadku zgłaszany jest błąd. W celu kontynuacji tego procesu w przypadku błędu operator oznaczany jest domyślną sygnaturą właściwą dla niego. Dzięki temu jest możliwość wychycenia większej ilości błędów, niż w przypadku zatrzymania kontroli na pierwszej niezgodności typów. Mechanizm zgłaszania błędów jest zintegrowany z odpowiadającym mu standardowym mechanizmem kompilatora Java. Dzięki temu użytkownik otrzymuje informacje o błędach w zapytaniach w taki sam sposób, jak w przypadku standardowych błędów kompilacji kodu Java.

## 6.3.4 Translacja zapytań SBQL4J na wyrażenia w języku Java

Aby uruchomić zapytania w standardowej maszynie wirtualnej Java, konieczne jest jego przetłumaczenie na wyrażenia tego języka. Może się to odbyć na wiele sposobów, różnych pod względem architektury i wydajności. W poniższym podrozdziale przedstawione zostaną te z nich, które zostały zaimplementowane w języku SBQL4J.

### 6.3.4.1 Tryb interpretera

W tym trybie wywołanie zapytania jest zamieniane na odpowiadające mu wywołanie kodu klasy interpretera. Parametrami tego wywołania jest tekst zapytania, oraz wszystkie obiekty i klasy z kontekstu kodu Javy, których nazwy zostały użyte w zapytaniu i rozpoznane podczas statycznej analizy. Zapytanie jest następnie parsowane, czego wynikiem jest drzewo składniowe AST. Inicjowane są stosy QRES i ENVŚ mające postać odpowiednich obiektów Java. Pierwsze dwie sekcje stosu ENVŚ wypełniają inicjalnie *bindery* do klas i obiektów będących parametrami zapytania. Wszystkie byty języka Java

są w tym trybie „opakowane” przez odpowiednie adaptery, umożliwiające traktowanie ich w charakterze obiektów SBQL. Uruchamiany jest interpreter, który w postfiksowej kolejności przegląda drzewo składniowe i wykonuje odpowiednie operacje na podstawie każdego jego elementów. Ostatecznie wynik zapytania jest zdejmowany ze stosu QRES, „wypakowywany” z adaptera i zwracany do kontekstu, z którego wywołano zapytanie.

Tryb interpretera, jako jedyny z obecnie zaimplementowanych w SBQL4J, umożliwia tworzenie zapytań dynamicznych w czasie działania programu. Jednak wadą w tym rozwiązaniu jest brak wczesnej kontroli typologicznej. Użytkownik musi sam zadbać o to, aby wszystkie nazwy były użyte prawidłowo oraz argumenty operatorów były właściwych typów. W przeciwnym wypadku wystąpi błąd czasu wykonania w trakcie ewaluacji zapytania.

Pod względem wydajności tryb interpretera jest najwolniejszy z wszystkich zaimplementowanych sposobów ewaluacji zapytań. Wynika to z konieczności parsowania zapytań przy ich każdym uruchomieniu, co jest kosztowną operacją. Dodatkowe opóźnienia wynikają z użycia adapterów dla każdego z przetwarzanych obiektów. Kosztowne jest też dynamiczne wiązanie nazw z bytów Javy. W ramach projektu stworzono architekturę umożliwiającą rozdzielenie sposobu rozwiązywania nazw od reszty aspektów przetwarzania zapytań. Zostały zaimplementowane dwa rozwiązania tego celu:

- Wiązanie nazw za pomocą refleksji Javy – jak już wspomnieliśmy, jest to standardowy mechanizm RTTI w języku Java. Jego wadą jest niska wydajność, pojedyncza operacja związania nazwy jest w ten sposób kilkanaście- do kilkudziesięciu razy wolniejsza od standardowego wywołania.
- Wiązanie nazw za pomocą biblioteki *CGLib* – biblioteka ta dostarcza funkcjonalności RTTI działając wydajniej niż refleksja Javy

#### 6.3.4.2 Generacja kodu Java

Oprócz trybu interpretera w języku SBQL4J dostępne są trzy inne warianty translacji zapytań. Ich wspólną cechą jest generowanie kodu nowej klasy, w której występuje metoda wyliczająca wynik zapytania. Kod tej metody jest różny dla każdego z wariantów. Zaletą tego podejścia jest możliwość przekształcenia drzewa składni zapytania w celu optymalizacji w trakcie statycznej analizy. W trybie interpretera jest to niemożliwe (chyba że po takiej translacji drzewo AST byłoby z powrotem tłumaczone na tekst zapytania, co nie zostało zrealizowane w projekcie).

Generatory kodu są klasami implementującymi interfejs *TreeVisitor*, podobnie jak inne klasy tego typu realizują swoją funkcjonalność odwiedzając każdy element drzewa składniowego zapytania. W momencie odwiedzenia każdego z elementów drzewa generowany jest fragment kodu. Najczęściej jest to wykonanie pewnej operacji w języku Java i przypisanie jej wyniku do nowej zmiennej.

#### 6.3.4.2.1 Wariant z kodem tożsamym z kodem interpretera

Przetwarzanie zapytań w tym wariacie jest najbardziej zbliżone do działania interpretera. Wygenerowana klasa posiada stosy QRES i ENVŚ biorące udział w wykonaniu zapytania. Kod metody jest tożsamy z kodem wykonanym przez interpreter podczas przeglądania drzewa AST.

#### 6.3.4.2.2 Wariant z kodem bez stosu QRES

Ten wariant różni się od poprzedniego kilkoma optymalizacjami. Nie występuje już w nim stos QRES, wyniki podzapytań są podstawiane do osobnych zmiennych. Ponieważ nazwy tych zmiennych najczęściej muszą być unikalne w ramach zapytania, przed generowaniem kodu elementy drzewa składniowego są dodatkowo dekorowane unikalnymi nazwami. Służy do tego klasa *ExpressionResultNameDecorator*. Dodatkowo, nie jest generowany kod koercji wyników podzapytań, gdzie nie jest to konieczne. Decyzja o koercji jest podejmowana podczas analizy zapytania na podstawie liczności wyników podzapytań zawartych w sygnaturach.

#### 6.3.4.2.3 Wariant z wczesnym wiązaniem nazw z kodem bez stosów ENVŚ i QRES

W tym wariacie przetwarzanie zapytań odbywa się z największym dotychczas wykorzystaniem środowiska maszyny wirtualnej Java. Wiązanie nazw odbywa się za pomocą stosu maszyny wirtualnej Java. W tym celu generowany jest odpowiedni kod, w którym nazwy związane z użyciem funkcji *nested* zamieniane są na odpowiednie wyrażenia ścieżkowe. W istocie jest to pewien rodzaj wczesnego wiązania nazw, z punktu widzenia języka zapytań. Jednak w odróżnieniu od języków programowania z wczesnym wiązaniem, gdzie nazwy zamieniane są na odwołania do konkretnych adresów pamięci, SBQL4J zamienia je na nazwy w języku Java. Ich rozwiązaniem zajmuje się maszyna wirtualna Javy dopiero w czasie uruchomienia programu. Dzięki temu zachowane są cechy charakterystyczne dla obiektowego języka programowania z późnym wiązaniem nazw takie jak np. dziedziczenie i polimorfizm. Odbywa się to z minimalnym kosztem wydajnościowym dzięki ścisłej integracji ze środowiskiem wykonawczym programu. SBQL4J jest dzięki temu prawdopodobnie jedną z najszybszych implementacji SBQL jakie dotychczas powstały, jeśli weźmiemy pod uwagę sumę operacji w skali mikro. Oczywiście jest to tylko jeden z istotnych aspektów wpływających na ogólną wydajność języka. Co najmniej równie istotnym jest zastosowanie wydajnego optymalizatora całych zapytań, a więc operacji w skali makro.

W celu przekształcenia wiązania nazw dostarczonych przez funkcję *nested* na wyrażenia ścieżkowe Java konieczne jest dokładne odtworzenie tej funkcji w momencie wiązania nazwy w trakcie analizy zapytania. W tym celu, podczas analizy gromadzone są dodatkowe informacje mające to umożliwić. Są to:

- Nazwy zmiennych pomocniczych użytych w wyrażeniach do iteracji po obiektach, na których wywoływana jest funkcja *nested*
- Do każdej sygnatury *bindera* utworzonej w wyniku działania funkcji *nested* są to:

- Referencja do sygnatury macierzystej
- W przypadku, gdy sygnaturą macierzystą jest sygnatura struktury – indeks pola z której pochodzi dany *binder*
- Referencja do elementu drzewa składniowego w którym wywołano funkcję *nested*
- Flaga informująca, czy dany *binder* został utworzony jako nazwa pomocnicza (w wyniku działania operatora *as* lub *group as*)

Na podstawie tych informacji, oraz typu wiążanego bindera (wartość, metoda lub klasa) generowany jest kod wyrażenia ścieżkowego. Odbywa się to w momencie odwiedzenia generatora kodu w elemencie drzewa w którym wiązana jest dana nazwa.

## 6.4 Model danych JAS0 w SBQL4J

Niniejszy podrozdział opisuje model danych języka SBQL4J na wyższym poziomie abstrakcji, w podobny sposób jak abstrakcyjne modele języka SBQL.

### 6.4.1 Opis ogólny

Model JAS0 jest odmianą modelu AS0 wzbogaconą o klasy, metody i konstruktory. Nie są modelowane związki obiekt-klasa oraz związki dziedziczenia klasa-klasa. Nie było to konieczne, gdyż tymi cechami zajmuje się maszyna wirtualna Javy. Z punktu widzenia języka zapytania dany obiekt posiada wszystkie cechy swojej klasy, oraz klas po których ona dziedziczy. Nie występuje zróżnicowanie na obiekty złożone i proste. Obiekt złożony może posiadać wartość atomową (np. obiekty wbudowanych typów takich jak *java.lang.Integer*, *java.lang.String*), jednak nie jest to istotne z punktu widzenia zewnętrznego systemu typów. Pewne operatory, np. algebraiczne, mogą oczekiwać odpowiednich typów posiadających wartość atomową, lecz oczekiwanie odpowiedniego typu jest cechą wspólną dla wszystkich operatorów i w tym przypadku nie wymaga dodatkowych środków. Nie jest modelowana hermetyzacja, nie ma podziału na cechy publiczne i prywatne. Zapytania mogą operować jedynie na publicznych właściwościach bytów programistycznych. Również w żaden sposób nie jest zarządzana tożsamość obiektu, podobnie jak w przypadku innych nieobecnych w modelu cech; odpowiedzialność za to przejmuje środowisko wykonawcze Java.

### 6.4.2 Obiekt

Jest parą uporządkowaną  $\langle F, M \rangle$ , gdzie:

- F jest zbiorem referencji do pól obiektu (pod-obiektów)
- M jest zbiorem metod obiektu

Obiekt jest reprezentowany podczas analizy statycznej jako *ValueSignature*.

### 6.4.3 Referencja

Można ją opisać parą uporządkowaną  $\langle n, o \rangle$ , gdzie:

- n jest nazwą zewnętrzną
- o jest obiektem

Referencja jest identyfikatorem logicznym wiążącym nazwę z konkretnym obiektem. W trakcie analizy jest reprezentowana przez *BinderSignature*.

#### 6.4.4 Klasa

Jest trójką uporządkowaną  $\langle F, M, C \rangle$ , gdzie:

- F jest zbiorem referencji do statycznych pól klasy (pod-obiektów wspólnych dla wszystkich obiektów danej klasy)
- M jest zbiorem metod statycznych klasy (funkcji)
- C jest zbiorem konstruktorów

Sygnaturą reprezentującą klasę jest *ClassSignature*.

#### 6.4.5 Metoda

Posiada nazwę, typ zwracany i listę parametrów. Podczas analizy jest reprezentowana przez *MethodSignature*.

#### 6.4.6 Konstruktor

Jest specjalną metodą tworzącą obiekt, przynależy do klasy. Zawiera listę parametrów, a typem zwracanym jest typ klasy, z której dany konstruktor pochodzi. Sygnaturą reprezentującą konstruktor jest *ConstructorSignature*.

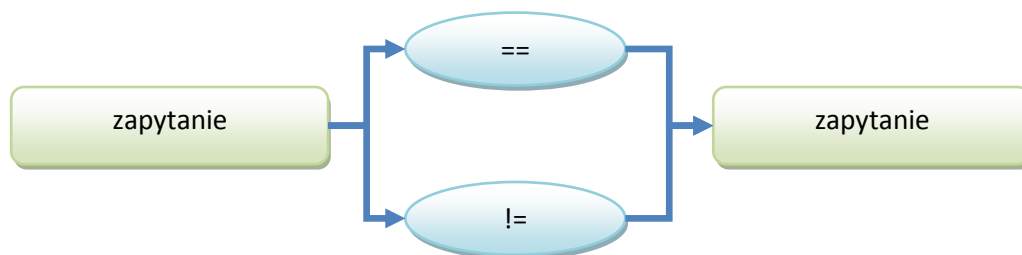
### 6.5 Opis operatorów języka

Poniższy podrozdział zawiera spis operatorów języka, opis ich składni, semantyki i tablicę kontroli typologicznej.

#### 6.5.1 Operatory algebraiczne

Cechą wspólną operatorów algebraicznych jest to, że kolejność argumentów nie odgrywa roli. Ich przetwarzanie odbywa się wyłącznie z użyciem stosu QRES. Stos ENVS nie jest używany podczas ich ewaluacji.

##### 6.5.1.1 Operatory porównania



#### Semantyka:

Operatory porównania służą do stwierdzenia, czy wynik zapytania po lewej stronie równa się (operator ==), bądź nie równa się (operator !=) wynikowi zapytania po prawej

stronie. Argumentami mogą być dowolne obiekty Java, do ich porównania wykorzystywana jest metoda `equals(Object obj)` z klasy `java.lang.Object`, będącej podstawą hierarchii dziedziczenia dla każdego obiektu Java. Metoda ta jest zdefiniowana w obiektach typu `String` i w typach liczbowych. Zaleceniem twórców języka Java jest aby każda klasa miała swoją definicję tej metody. W przypadku gdy tak nie jest, porównanie polega na sprawdzeniu czy obydwa referencje wskazują na ten sam obiekt. Obydwa operatory zwracają obiekt klasy `java.lang.Boolean`, będący wartością logiczną `true` albo `false`.

#### Przykłady:

```
„Napis1” == „Napis2”
3.36 != 10
```

#### Kontrola typologiczna:

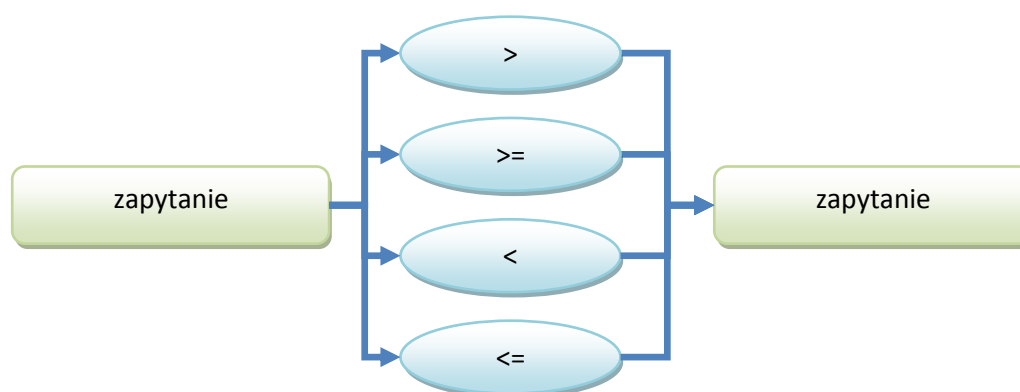
Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	1..1	1..1	1..1
Typ kolekcji	element	element	element
Typ obiektu	<code>java.lang.Object</code>	<code>java.lang.Object</code>	<code>java.lang.Boolean</code>

#### Generowany kod Java dla jednego z przykładów:

```
/** query="Napis1" == "Napis2"
 */
public Boolean executeQuery() {
    Boolean _queryResult = OperatorUtils.equalsSafe("Napis1", "Napis2");

    return _queryResult;
}
```

### 6.5.1.2 Operatory porównania zakresowego



#### Semantyka:

Operatory sprawdzają, czy pomiędzy argumentami zachodzi odpowiednia relacja porównania (>, >=, <, <=). Wynikiem jest wartość logiczna `java.lang.Boolean`. Argumenty mogą być typu liczbowego, jak również dowolnego typu implementującego interfejs `java.lang.Comparable`. W ten sposób możliwe jest porównywanie typów podstawowych innych niż liczbowe (np. napisy, daty) oraz dowolnych innych typów zdefiniowanych przez użytkownika. **Stanowi to znaczną różnicę jakościową w stosunku do**

dotychczasowych prototypów języków opartych na SBQL, gdyż pozwala użyć operatorów porównania zakresowego w znacznie szerszym kontekście niż dotychczas.

#### Przykłady:

```
8.6 > 3
date1 <= date2
```

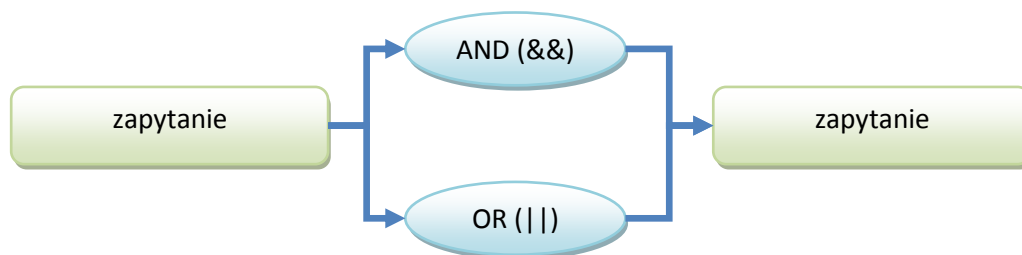
#### Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	1..1	1..1	1..1
Typ kolekcji	element	element	element
Typ obiektu	java.lang.Number	java.lang.Number	java.lang.Boolean
	java.lang.Comparable	java.lang.Comparable	java.lang.Boolean

#### Generowany kod Java dla jednego z przykładów:

```
/** query='date1 >= date2'
 */
public Boolean executeQuery() {
    Boolean _queryResult = date1.compareTo(date2) <= 0;
    return _queryResult;
}
```

### 6.5.1.3 Operatory logiczne



#### Semantyka:

Wynikiem działania operatorów jest wartość logiczna wyliczona na podstawie argumentów będących wynikami lewego i prawego zapytania. Argumenty te muszą być pojedynczymi wartościami logicznymi (*java.lang.Boolean*). Operatory mają dwojaką składnię. Operator mnożenia logicznego można wyrazić za pomocą słowa kluczowego „AND” lub w stylu C++: „&&”. Dla operatora sumy logicznej jest to odpowiednio „OR” i „||”.

#### Przykłady:

```
true OR false
unitsOnStock > 0 AND unitPrice >= 3.0
```

#### Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	1..1	1..1	1..1

Typ kolekcji	element	element	element
Typ obiektu	java.lang.Boolean	java.lang.Boolean	java.lang.Boolean

Generowany kod Java dla jednego z przykładów:

```

/** query='true OR false'
 */
public Boolean executeQuery() {
    Boolean _queryResult = true || false;

    return _queryResult;
}

```

#### 6.5.1.4 Operator negacji logicznej



Semantyka:

Operator zwraca wartość logiczną będącą odwrotnością wyniku zapytania leżącego po prawej stronie operatora. Zapytanie to musi zwrócić wartość typu *java.lang.Boolean*. Operator można wyrazić za pomocą słowa „NOT” lub „!”.

Przykłady:

```

NOT true
!order.isExpired()

```

Kontrola typologiczna:

Sygnatura argumentu	Zapytanie	Wynik
Liczność	1..1	1..1
Typ kolekcji	element	element
Typ obiektu	java.lang.Boolean	java.lang.Boolean

#### 6.5.1.5 Operator sprawdzenia typu



Semantyka:

Operator sprawdza, czy wynik zapytania po lewej stronie operatora jest określonego typu. Typ ten jest określany przez wynik prawego zapytania. Obydwa argumenty muszą być pojedynczymi elementami, a wynikiem jest wartość logiczna.

Przykłady:

```

"Napis" instanceof String
objects as o where o instanceof java.lang.Double

```

Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	1..1	1..1	1..1

Typ kolekcji	element	element	element
Typ obiektu	java.lang.Object	java.lang.Class	java.lang.Boolea n

### Generowany kod Java dla jednego z przykładów:

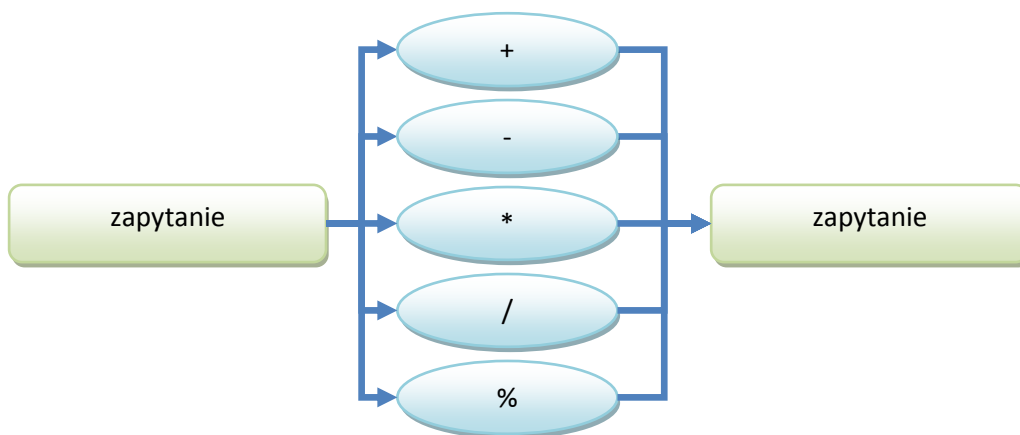
```

/** query="Napis" instanceof String'
 */
public Boolean executeQuery() {
    Boolean _queryResult = "Napis" instanceof java.lang.String;

    return _queryResult;
}

```

### 6.5.1.6 Operatory arytmetyczne



#### Semantyka:

Operatory wykonują odpowiednie równania arytmetyczne na zadanych argumentach. Dla wszystkich mechanizm ewaluacji jest podobny. Po wykonaniu podzapytań lewego i prawego ich wyniki są konsumowane przez operator, który wykonuje na nich odpowiednią operację arytmetyczną (dodawanie, odejmowanie, mnożenie, dzielenie, modulo) i zwraca jej wynik. W zależności od typów argumentów operator „+” zamiast dodawania może wykonać konkatencję napisów. Dzieje się tak, gdy przynajmniej jeden z argumentów jest typu napisowego, a więc *java.lang.String*. Pozostałe operatory działają jedynie na typach liczbowych, a więc dziedziczących po *java.lang.Number*. Z wyjątkiem wspomnianej konkatencji napisów, wynikiem operacji jest silniejszy z typów numerycznych będących argumentami. Są to, w kolejności od najsilniejszego do najslabszego: *Double*, *Float*, *Long*, *Integer*, *Short*, *Byte*.

#### Przykłady:

```

12.5 / 4
„napis” + 5

```

#### Kontrola typologiczna dla operatora „+”:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	1..1	1..1	1..1

Typ kolekcji	element	element	element
Typ obiektu	java.lang.Number	java.lang.Number	java.lang.Number
	java.lang.Object	java.lang.String	java.lang.String
	java.lang.String	java.lang.Object	java.lang.String

Kontrola typologiczna dla operatorów „-”, „\*”, „/”, „%”:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	1..1	1..1	1..1
Typ kolekcji	element	element	element
Typ obiektu	java.lang.Number	java.lang.Number	java.lang.Number

Generowany kod Java dla jednego z przykładów:

```

/** query="napis" + 5'
 */
public String executeQuery() {
    return ("napis" + 5);
}

```

### 6.5.1.7 Operator tworzenia bagu

#### 6.5.1.7.1 Wariant z domyślną implementacją bagu



Semantyka:

Wynikiem działania operatora jest kolekcja typu *bag*. Tworzona jest jej domyślna implementacja oparta na kolekcji typu *java.util.ArrayList*.

Przykłady:

```

bag(1, 2.53, 5, 8, 4)
bag(customers.orders.total)

```

Kontrola typologiczna:

Sygnatura argumentu	Zapytanie	Wynik
Liczność	1..1	1..1
	0..*	0..*
Typ kolekcji	element / bag / sequence	bag
Typ obiektu	java.lang.Object	java.lang.Object
	pl.wcislo.sbc14j.java.model.runtime.Struct	java.lang.Object

#### 6.5.1.7.2 Wariant z wyborem implementacji bagu



Semantyka:

W tym wariantcie operatora bag użytkownik ma możliwość wyboru implementacji kolekcji. Lewe zapytanie powinno zwrócić klasę implementującą podstawowy interfejs

kolekcji w Javie tj. *java.util.Collection*. W ten sposób możliwa jest specjalizacja bagów bez zmian w składni i semantyce języka, np. podanie jako lewego argumentu klasy *java.util.HashSet* spowoduje utworzenie kolekcji typu zbiór z automatycznym usuwaniem duplikatów. Specjalizacja jest też rodzajem optymalizacji zapytań. Poszczególne implementacje różnią się od siebie czasem wykonania podstawowych operacji, takich jak wstawianie, usuwanie, wyszukiwanie elementu po indeksie, iteracja po elementach kolekcji.

#### Przykłady:

```
bag<HashSet>(1, 2.53, 5, 8, 4)
bag<org.apache.commons.collections.bag.SynchronizedSortedBag>(customers.orders.total)
```

#### Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	1..1	1..1	1..1
	1..1	0..*	0..*
Typ kolekcji	element	element/bag/sequence	bag<T>
Typ obiektu	java.lang.Class<T>	java.lang.Object	java.lang.Object
	java.lang.Class<T>	pl.wcislo.sbg14j.java.model.runtime.Struct	java.lang.Object

### 6.5.1.8 Operator tworzenia sekwencji

#### 6.5.1.8.1 Wariant z domyślną implementacją sekwencji



#### Semantyka:

Wynikiem działania operatora jest kolekcja typu *sequence*. Tworzona jest jej domyślna implementacja oparta na kolekcji typu *java.util.ArrayList*.

#### Przykłady:

```
sequence(1, 3, 5, 8, 12)
sequence(customers.orders.total)
```

#### Kontrola typologiczna:

Sygnatura argumentu	Zapytanie	Wynik
Liczność	1..1	1..1
	0..*	0..*
Typ kolekcji	element / bag / sequence	sequence
Typ obiektu	java.lang.Object	java.lang.Object
	pl.wcislo.sbg14j.java.model.runtime.Struct	java.lang.Object

### 6.5.1.8.2 Wariant z wyborem implementacji sekwencji



#### Semantyka:

Podobnie jak w przypadku operatora *bag*, użytkownik ma możliwość wyboru implementacji sekwencji. Lewe zapytanie powinno zwrócić klasę kolekcji zachowującej kolejność elementów czyli implementującej interfejs *java.util.List*.

#### Przykłady:

```
Sequence<LinkedList>(1, 3, 5, 8, 12)
sequence<org.apache.commons.collections.list.TreeList>(customers.orders.total)
```

#### Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	1..1	1..1	1..1
	1..1	0..*	0..*
Typ kolekcji	element	element/bag/sequence	sequence<T>
Typ obiektu	java.lang.Class<T>	java.lang.Object	java.lang.Object

### 6.5.1.9 Operator sumy



#### Semantyka:

Ewaluacja operatora polega na wyliczeniu podzapytania, które powinno zwrócić kolekcję lub pojedynczy obiekt liczbowy. Następnie obliczana jest suma wszystkich elementów tego wyniku. Podobnie jak w przypadku operatorów arytmetycznych, typ wyniku jest najmocniejszym typem liczbowym spośród typów elementów podzapytania.

#### Przykłady:

```
sum(bag(1, 2.53, 5, 8, 4))
sum(customers.orders.total)
```

#### Kontrola typologiczna:

Sygnatura argumentu	Zapytanie	Wynik
Liczność	0..*	1..1
	1..1	1..1
Typ kolekcji	bag / sequence / element	element
Typ obiektu	java.lang.Number	java.lang.Number

### 6.5.1.10 Operator zliczenia



#### Semantyka:

Ewaluacja operatora polega na wyliczeniu podzapytania, które powinno zwrócić kolekcję lub pojedynczy obiekt dowolnego typu. Następnie zliczana jest liczba elementów tego wyniku, która jest wynikiem działania operatora.

#### Przykłady:

```
count(bag(true, true, false, true, false))
count(company.employs)
```

#### Kontrola typologiczna:

Sygnatura argumentu	Zapytanie	Wynik
Liczność	0..*	1..1
	1..1	1..1
Typ kolekcji	bag / sequence / element	element
Typ obiektu	java.lang.Object	java.lang.Integer

### 6.5.1.11 Operator średniej wartości



#### Semantyka:

Ewaluacja operatora polega na wyliczeniu podzapytania, które powinno zwrócić kolekcję lub pojedynczy obiekt liczbowy. Następnie wyliczana jest średnia z elementów tego wyniku, która jest wynikiem działania operatora.

#### Przykłady:

```
avg(sequence(4, 2.5, 9.2, 8))
avg(dept.employs.salary)
```

#### Kontrola typologiczna:

Sygnatura argumentu	Zapytanie	Wynik
Liczność	1..*	1..1
	1..1	1..1
Typ kolekcji	element	element
	bag / sequence	element
Typ obiektu	java.lang.Number	java.lang.Double

### 6.5.1.12 Operator minimalnej wartości



#### Semantyka:

Ewaluacja operatora polega na wyliczeniu podzapytania, które powinno zwrócić kolekcję lub pojedynczy obiekt liczbowy. Następnie znajdowana jest najmniejsza wartość z elementów tego wyniku, która jest zwracana jako wynik działania operatora.

#### Przykłady:

```
min(bag(9, 3.5, 6, 4, 10, 1.45))
min(emp.salary)
```

#### Kontrola typologiczna:

Sygnatura argumentu	Zapytanie	Wynik
Liczność	1..*	1..1
	1..1	1..1
Typ kolekcji	element	element
	bag / sequence	element
Typ obiektu	java.lang.Number	java.lang.Number

### 6.5.1.13 Operator maksymalnej wartości



#### Semantyka:

Ewaluacja operatora polega na wyliczeniu podzapytania, które powinno zwrócić kolekcję lub pojedynczy obiekt liczbowy. Następnie znajdowana jest największa wartość z elementów tego wyniku, która jest zwracana jako wynik działania operatora.

#### Przykłady:

```
max(bag(9, 3.5, 6, 4, 10, 1.45))
max(emp.salary)
```

#### Kontrola typologiczna:

Sygnatura argumentu	Zapytanie	Wynik
Liczność	1..*	1..1
	1..1	1..1
Typ kolekcji	element	element
	bag / sequence	element
Typ obiektu	java.lang.Number	java.lang.Number

### 6.5.1.14 Operator egzystencjalny



#### Semantyka:

Ewaluacja operatora polega na sprawdzeniu liczności argumentu. Wynikiem działania operatora jest wartość logiczna. Jest to *true* w przypadku gdy liczność argumentu jest większa od zera. W przeciwnym wypadku jest to *false*.

#### Przykłady:

```
exists emp.salary
exists bag(1, 3, 5)
```

#### Kontrola typologiczna:

Sygnatura argumentu	Prawe zapytanie	Wynik
Liczność	0..*	1..1
Typ kolekcji	bag / sequence / element	element
Typ obiektu	java.lang.Object	java.lang.Boolean

### 6.5.1.15 Operator sumy zbiorów



#### Semantyka:

Ewaluacja operatora polega na wyliczeniu lewego i prawego podzapytania, które powinny zwrócić dowolny wynik. Następnie wykonywana jest suma mnogościowa, która jest zwracana jako wynik działania operatora. Zwracany typ jest najbliższą wspólną klasą w hierarchii dziedziczenia obydwu typów argumentów. Wynikowy typ kolekcji jest wyliczany na podstawie następującej reguły: jeżeli jednym z argumentów jest *bag*, lub obydwa argumenty są pojedynczymi elementami wynikiem jest *bag*. W przeciwnym wypadku wynikiem jest sekwencja, a obydwa argumenty są łączone z zachowaniem ich kolejności.

#### Przykłady:

```
bag(2, 3, 7, 9) union bag(4, 5, 8, 10)
pracownicy union studenci
```

#### Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	0..*	0..*	0..*
Typ kolekcji	bag	sequence / element	bag
	sequence / element	bag	bag
	sequence	element	sequence
	element	sequence	sequence

	sequence	sequence	sequence
	element	element	bag
Typ obiektu	T1	T2	? super T1, T2

### 6.5.1.16 Operator przecięcia zbiorów



#### Semantyka:

Operator wykonuje przecięcie mnogościowe na wynikach podzapytań. Typ wyniku prawego zapytania musi być podtypem lub takim samym typem jak wynik lewego zapytania. Typ kolekcji wyniku jest identyczny z typem kolekcji lewego zapytania.

#### Przykłady:

```

bag(5, 2, 7, 1) intersect bag(6, 5, 1, 8)
studenci intersect pracownicy
  
```

#### Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	1..1	0..*	0..1
	0..*	0..*	0..*
Typ kolekcji	bag / sequence	element	bag / sequence
	element	element	element
Typ obiektu	T	? extends T	T

### 6.5.1.17 Operator różnicy zbiorów



#### Semantyka:

Operator wykonuje różnicę mnogościową na wynikach podzapytań. Typ wyniku prawego zapytania musi być podtypem lub takim samym typem jak wynik lewego zapytania. Typ kolekcji wyniku jest identyczny z typem kolekcji lewego zapytania.

#### Przykłady:

```

bag(5, 2, 7, 1) minus bag(6, 5, 1, 8)
studenci minus pracownicy
  
```

#### Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	1..1	0..*	0..1
	0..*	0..*	0..*
Typ kolekcji	bag / sequence	element	bag / sequence
	element	element	element
Typ obiektu	T	? extends T	T

### 6.5.1.18 Operator zawierania się zbiorów



#### Semantyka:

Operator wykonuje zawieranie mnogościowe na wynikach podzapytań. Typ wyniku prawego zapytania musi być podtypem lub takim samym typem jak wynik lewego zapytania. Typ kolekcji wyniku jest identyczny z typem kolekcji lewego zapytania.

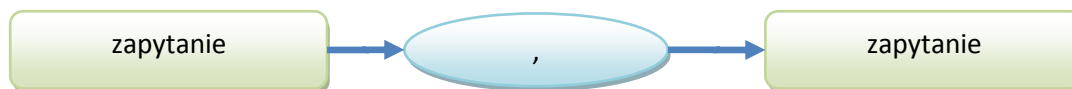
#### Przykłady:

```
bag(5, 2, 7, 1) in bag(6, 5, 1, 8)
studenci in pracownicy
```

#### Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	0..*	0..*	1..1
Typ kolekcji	bag / sequence	bag / sequence	element
Typ obiektu	T	? extends T	java.lang.Boolean

### 6.5.1.19 Operator iloczynu kartezyjskiego



#### Semantyka:

Operator wykonuje iloczyn kartezyjski na wynikach podzapytań.

#### Przykłady:

```
(1, 2, 3, 4)
customers, orders
```

#### Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	0..*	0..*	0..*
	1..1	1..1	1..1
Typ kolekcji	bag / sequence	element	bag / sequence
	element	bag / sequence	bag / sequence
	element	element	element
Typ obiektu	java.lang.Object	java.lang.Object	pl.wcislo.sbcql4j. java.model.runtime. e.Struct

## 6.5.1.20 Operator zakresowy

### 6.5.1.20.1 Wariant z wyborem pojedynczego elementu



#### Semantyka:

Operator zwraca element z kolekcji będącej wynikiem lewego zapytania o indeksie określonym przez prawe zapytanie. Indeks powinien być typu liczbowego *java.lang.Integer*. Typ wyniku jest taki sam jak typ wyniku lewego zapytania.

#### Przykłady:

```
customers[5]
orders[count(orders) - 1]
```

#### Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	0..*	1..1	1..1
Typ kolekcji	bag / sequence	element	element
Typ obiektu	T	java.lang.Integer	T

### 6.5.1.20.2 Wariant z dolnym zakresem



#### Semantyka:

Wynikiem działania operatora jest podzbiór elementów kolekcji będącej wynikiem lewego zapytania, ograniczonej dolnym indeksem określonym przez prawe zapytanie. Indeks powinien być typu liczbowego *java.lang.Integer*. Typ wyniku jest taki sam jak typ wyniku lewego zapytania.

#### Przykłady:

```
customers[2..*]
orders[n+1..*]
```

#### Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	0..*	1..1	0..*
Typ kolekcji	bag	element	bag
	sequence	element	sequence
Typ obiektu	T	java.lang.Integer	T

### 6.5.1.20.3 Wariant z dolnym i górnym zakresem



#### Semantyka:

W tym wariacie operatora zakresowego wynikiem jest podzbiór kolekcji będącej wynikiem lewego zapytania ograniczony dolnym i górnym indeksem. Indeksy te są określane przez odpowiednio środkowe i prawe podzapytanie, powinny być typu liczbowego *java.lang.Integer*. Typ wyniku jest taki sam jak typ wyniku lewego zapytania.

#### Przykłady:

```
customers[3..5]
bag(1, 5, 7, 3)[1..2]
```

#### Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Środkowe / Prawe zapytanie	Wynik
Liczność	0..*	1..1	0..*
Typ kolekcji	bag	element	bag
	sequence	element	sequence
Typ obiektu	T	java.lang.Integer	T

### 6.5.1.21 Operator nazwy pomocniczej AS



#### Semantyka:

Ewaluacja operatora polega na opatrzeniu każdego elementu zapytania nazwą pomocniczą określoną po prawej stronie operatora. W przypadku generowania natywnego kodu Javy następuje przepisanie wyniku zapytania do nowej zmiennej. Zazwyczaj nazwa ta jest wyłącznie nazwą pomocniczą i nie ma odwzorowania w wygenerowanym kodzie, chyba, że utworzony binder jest konsumowany ostatecznie przez operator wykonujący iloczyn kartezjański. W takim przypadku możliwe jest odwołanie się do elementów utworzonej struktury za pomocą nazwy pomocniczej, np.:

```
Struct s = #{ 10 as liczba, 20 as liczba2 };
Integer liczba = (Integer) s.get("liczba");
```

Wynik jest takiego samego typu jak typ zapytania, podobnie jak typ kolekcji.

#### Przykłady:

```
10 as liczba
bag(1, 5, 7, 3) as n
```

#### Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Wynik
---------------------	----------------	-------

Liczność	0..*	0..*
	1..1	1..1
Typ kolekcji	bag / sequence	bag / sequence
	element	element
Typ obiektu	T	T

### 6.5.1.22 Operator nazwy pomocniczej GROUP AS



#### Semantyka:

Ewaluacja operatora GROUP AS jest podobna jak w przypadku operatora AS, z tą różnicą, że dla całego wyniku zapytania tworzony jest dokładnie jeden binder z nazwą określoną po prawej stronie zapytania.

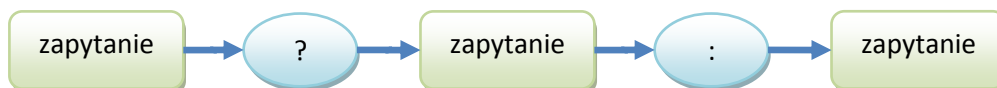
#### Przykłady:

```
bag(1, 5, 7, 3) group as n
customers.orders group as o
```

#### Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Wynik
Liczność	0..*	1..1
	1..1	1..1
Typ kolekcji	bag / sequence	element
	element	element
Typ obiektu	T	T

### 6.5.1.23 Operator warunkowy



#### Semantyka:

Podczas ewaluacji operatora warunkowego najpierw wykonywane jest lewe podzapytanie. Zwraca ono wartość logiczną, na podstawie której podejmowana jest decyzja, które podzapytanie będzie wykonane w następnej kolejności. W przypadku wyniku *true* wykonywane jest środkowe podzapytanie i zwracany jest jego wynik. W przeciwnym wypadku wynikiem działania operatora jest wynik prawego podzapytania.

#### Przykłady:

```
exists salary ? salary : 0
i % 2 == 0 ? "parzysty" : "nieparzysty"
```

#### Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Środkowe zapytanie	Prawe zapytanie	Wynik
---------------------	----------------	--------------------	-----------------	-------

Liczność	1..1	0..*	0..*	0..*
Typ kolekcji	element	element / bag / sequence	element / bag / sequence	element / bag / sequence
Typ obiektu	java.lang.Boolean	T1	T2	? super T1, T2

## 6.5.2 Operatory niealgebraiczne

Operatory niealgebraiczne różni od algebraicznych użycie podczas ich ewaluacji stosu ENVS. Nie można ich działania opisać standardowymi algebraami.

### 6.5.2.1 Operator selekcji warunkowej



#### Semantyka:

Wynikiem działania operatora *where* jest podzbiór wyniku zapytania Q1 leżącego po lewej stronie operatora. Ewaluacja przebiega następująco: najpierw wykonywane jest zapytanie Q1, następnie dla każdego z elementów wyniku tego zapytania wywoływana jest funkcja *nested* i wykonywane zapytanie Q2 leżące po prawej stronie operatora. Wynikiem tego ostatniego powinna być wartość logiczna, jeżeli jest to *true*, przetwarzany element wyniku zapytania Q1 dodawany jest do wyniku.

#### Przykłady:

```

napisy where length() > 5
products where unitsInStock == 0
  
```

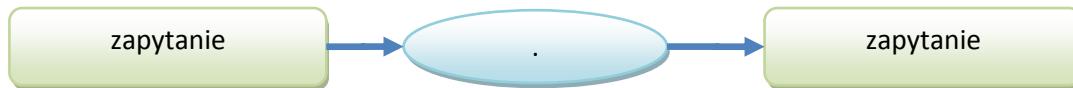
#### Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	0..1	0..1	0..1
	0..*	0..1	0..*
Typ kolekcji	element	element	element
	bag	element	bag
	sequence	element	sequence
Typ obiektu	T	java.lang.Boolean	T

### Generowany kod Java dla jednego z przykładów:

```
public java.util.List<Product> executeQuery() {
    java.util.List<Product> _queryResult = new java.util.ArrayList<Product>();
    for (Product _whereEl : products) {
        Boolean _equalsResult = OperatorUtils.equalsSafe(_whereEl.unitsInStock, 0);
        if (_equalsResult) {
            _queryResult.add(_whereEl);
        }
    }
    return _queryResult;
}
```

### 6.5.2.2 Operator nawigacji



#### Semantyka:

Ewaluacja operatora jest następująca: najpierw wykonywane jest podzapytanie leżące po lewej stronie operatora. Następnie na każdym z elementów jego wyniku wykonywana jest funkcja *nested*, po czym wykonywane jest prawe podzapytanie. Jego wynik dodawany jest do wyniku operatora. W przypadku wygenerowanego kodu Java, gdzie w czasie wykonania nie występuje stos ENVs i funkcja *nested*, operator nawigacji zamieniany jest na odpowiednie wyrażenie Java będące odpowiednikiem funkcji *nested* i prawego zapytania.

#### Przykłady:

```
"Napis".length()
products.category
```

#### Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	0..*	0..*	0..*
	0..1	0..*	0..*
	0..*	0..1	0..*
	0..1	0..1	0..1
Typ kolekcji	bag / sequence	element	bag / sequence
	element	bag / sequence	bag / sequence
	element	element	element
Typ obiektu	java.lang.Object	T	T

### Generowany kod Java dla jednego z przykładów:

```
/** query='products.category'
 */
public java.util.List<String> executeQuery() {
    java.util.List<String> _queryResult = new java.util.ArrayList<String>();
    for (Product _dotEl : products) {
        _queryResult.add(_dotEl.category);
    }
    return _queryResult;
}
```

### 6.5.2.3 Operator zależnego złączenia



#### Semantyka:

Ewaluacja operatora polega na wykonaniu zapytania leżącego po lewej stronie operatora, następnie na każdym z elementów jego wyniku wykonywana jest funkcja *nested*. Kolejnym krokiem jest wykonanie prawego zapytania i wykonanie iloczynu kartezyjskiego na jego wyniku oraz przetworzonym elemencie wyniku lewego zapytania. W przypadku wygenerowanego kodu Javy wywoływana jest funkcja wykonująca iloczyn kartezyjski odpowiedni dla liczności argumentów. Wynikiem działania operatora jest obiekt reprezentujący strukturę lub odpowiednia kolekcja (*bag* lub sekwencja) takich obiektów.

#### Przykłady:

```
customers join orders
products as p join p.unitPrice as price
```

#### Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	0..*	0..*	0..*
	0..1	0..*	0..*
	0..*	0..1	0..*
	0..1	0..1	0..1
Typ kolekcji	bag / sequence	element	bag / sequence
	element	bag / sequence	bag / sequence
	element	element	element
Typ obiektu	java.lang.Object	java.lang.Object	pl.wcislo.sq14j. java.model.runtime. e.Struct

#### Generowany kod Java dla jednego z przykładów:

```

/** query='customers join orders'
 */
public java.util.List<Struct> executeQuery() {
    java.util.List<Struct> _queryResult = new java.util.ArrayList< Struct>();
    for (Customer _joinEl : customers) {
        _queryResult.addAll(OperatorUtils.cartesianProductCC(_joinEl,
            _joinEl.orders, "", ""));
    }
    return _queryResult;
}

```

### 6.5.2.4 Operator kwantyfikatora uniwersalnego



Semantyka:

Ewaluacja operatora jest następująca: najpierw wykonywane jest lewe zapytanie. Jeżeli liczność jego wyniku wynosi zero, operator zwraca wartość logiczną *true*. W przeciwnym wypadku dla każdego z elementów wyniku wywoływana jest funkcja *nested*, następnie wykonywane jest prawe zapytanie, które powinno zwrócić wartość logiczną. Jeżeli jest to *false*, operator zwraca ją jako wynik. W przeciwnym wypadku przetwarzany jest kolejny element wyniku lewego zapytania. Gdy nie ma już więcej elementów do przetworzenia, operator zwraca wartość logiczną *true*.

Przykłady:

```
all (bag(1, 5, 2, 6, 2) as n) (n % 2 == 1)
all products unitPrice < 100.0
```

Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	0..*	1..1	1..1
Typ kolekcji	bag / sequence	element	element
	element	element	element
Typ obiektu	java.lang.Object	java.lang.Boolean	java.lang.Boolean

**6.5.2.5 Operator kwantyfikatora egzystencjalnego**Semantyka:

Ewaluacja operatora jest podobna do operatora kwantyfikatora uniwersalnego. Jeżeli liczność jego wyniku wynosi zero, operator zwraca wartość logiczną *false*. W przeciwnym wypadku dla każdego z elementów wyniku wywoływana jest funkcja *nested*, następnie wykonywane jest prawe zapytanie, które powinno zwrócić wartość logiczną. Jeżeli jest to *true*, operator zwraca ją jako wynik. W przeciwnym wypadku przetwarzany jest kolejny element wyniku lewego zapytania. Gdy nie ma już więcej elementów do przetworzenia, operator zwraca wartość logiczną *false*.

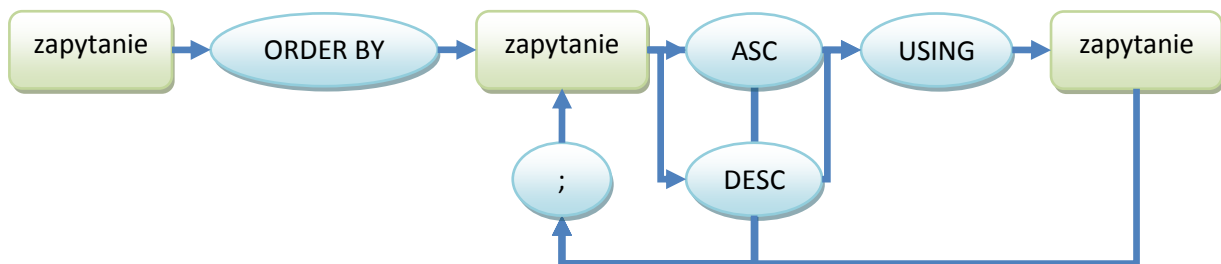
Przykłady:

```
any (bag(1, 5, 2, 6, 2) as n) (n < 0)
any products unitsInStock == 0
```

Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	0..*	1..1	1..1
Typ kolekcji	bag / sequence	element	element
	element	element	element
Typ obiektu	java.lang.Object	java.lang.Boolean	java.lang.Boolean

### 6.5.2.6 Operator sortowania



#### Semantyka:

Działanie operatora polega na sortowaniu kolekcji będącej wynikiem lewego podzapytania według wybranych kryteriów. Java dostarcza mechanizmy umożliwiające sortowanie dowolnych obiektów. Funkcja określająca kolejność może być zdefiniowana w sortowanym obiekcie (wtedy taki obiekt powinien implementować interfejs *java.lang.Comparable*). Możliwe jest również zdefiniowanie wielu różnych kryteriów sortowania obiektów danego typu niezależnie od definicji sortowanego obiektu. W tym celu tworzy się specjalne obiekty, typu *java.util.Comparator<T>*, gdzie T jest typem sortowanego obiektu, zawierające funkcję kolejności dla danego typu obiektu. W ten sposób rozwiązano problem opisany w (Subieta, 2004) na str. 262 związany z sortowaniem napisów ze znakami narodowymi w sposób bardziej ogólny, nie ingerujący bezpośrednio w semantykę języka. Do standardowych bibliotek Javy dołączone są definicje obiektów sortujących napisy dla wielu języków narodowych. Operator *order by* korzysta z obydwu wymienionych technik określania kolejności obiektów. Środkowe zapytanie powinno zwrócić klucz sortowania, może ono wykorzystać nazwy zwrócone przez funkcję *nested* wywołaną uprzednio na elemencie sortowanej kolekcji. Dla każdego z kluczy można określić kolejność sortowania słowami ASC lub DESC (jest to opcjonalne, domyślny jest ASC). Dodatkowo dla klucza można określić wspomniany obiekt typu *Comparator*. Jest on opcjonalny, gdy klucz zawiera w sobie definicję kolejności (implementuje interfejs *Comparable*); w przeciwnym wypadku jest wymagany. Możliwe jest również zdefiniowanie wielu kluczy sortowania oddzielonymi średnikami. Kod Java wygenerowany na podstawie zastosowanego operatora zawiera definicję nowego obiektu typu *Comparator* parametryzowanego typem sortowanej kolekcji, w którym zawarte są definicje sortowania po wszystkich użytych kluczach wraz z użytymi parametrami. Następnie wywoływana jest funkcja sortowania z standardowej biblioteki Javy *java.util.Collections.sort(List list, Comparator c)*. Funkcja ta gwarantuje stabilność, tj. elementy wyliczone jako równe pozostają w oryginalnym porządku. Funkcja ta gwarantuje również wydajność na poziomie  $n \log(n)$ , choć w naszym przypadku może być ona niższa ze względu na możliwość wywołania funkcji i metod o dowolnej złożoności, zarówno w zapytaniach zwracających klucze sortowania, jak i w metodach porównujących zdefiniowanych w komparatorach dołączonych do kluczy.

Ostatecznie wynikiem działania operatora jest posortowana kolekcja.

**Przykłady:**

```
products order by unitPrice
words as w order by w.length(); w using comp
```

**Kontrola typologiczna:**

Sygnatura argumentu	Lewe zapytanie	Środkowe zapytanie	Prawe zapytanie (opcjonalne)	Wynik
Liczność	0..*	1..1	1..1	1..1
Typ kolekcji	bag / sequence	element	element	sequence
Typ obiektu	T	java.lang.Comparable	(nie występuje)	T
	T	java.lang.Object	java.util.Comparator	T

**Generowany kod Java dla jednego z przykładów:**

```
/** query='words as w
        order by w.length(); w using comp'
 */
public java.util.List<String> executeQuery() {
    java.util.List<String> _asResult_w = words;
    java.util.List<String> _queryResult = new java.util.ArrayList<String>();
    _queryResult.addAll(_asResult_w);
    Comparator<String> _comparator0 = new Comparator<String>() {
        public int compare(String _leftObj, String _rightObj) {
            if (_leftObj == null) {
                return -1;
            }
            int res = 0;
            Integer _leftParam0;
            {
                String _dotEl = _leftObj;
                Integer _mth_lengthResult = _dotEl.length();
                _leftParam0 = _mth_lengthResult;
            }
            Integer _rightParam0;
            {
                String _dotEl = _rightObj;
                Integer _mth_lengthResult = _dotEl.length();
                _rightParam0 = _mth_lengthResult;
            }
            if (_leftParam0 != null) {
                res = _leftParam0.compareTo(_rightParam0);
            } else {
                return -1;
            }
            if (res != 0) {
                return res;
            }
            String _leftParam1;
            _leftParam1 = _leftObj;
            String _rightParam1;
            _rightParam1 = _rightObj;
            if (comp != null) {
                res = comp.compare(_leftParam1, _rightParam1);
            } else {
                res = 0;
            }
            return res;
        }
    };
    Collections.sort(_queryResult, _comparator0);
    return _queryResult;
}
```

### 6.5.2.7 Operator tranzytywnego domknięcia



#### Semantyka:

Ewaluacja operatora jest następująca: wykonywane jest podzapytanie po lewej stronie, następnie na każdym z elementów jego wyniku wywoływana jest funkcja *nested* i wykonywane prawe zapytanie. Jego wynik dodawany jest do wyniku lewego zapytania.

#### Przykłady:

```
engine close by part
new File("C:\\") close by listFiles()
```

#### Kontrola typologiczna:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	0..*	0..*	0..*
Typ kolekcji	bag / sequence / element	bag / sequence / element	bag
Typ obiektu	T	T	T

#### Generowany kod Java dla jednego z przykładów:

```
/** query='new File("C:\\") close by listFiles()'
 */
public java.util.List<File> executeQuery() {
    File _constrResult = new File("C:\\");
    java.util.List<File> _queryResult = new ArrayList<File>();
    _queryResult.add(_constrResult);
    int _il = 0;
    while (_il < _queryResult.size()) {
        File _closeByEl = _queryResult.get(_il);
        java.util.List<File> _mth_listFilesResult =
            ArrayUtils.toList(_closeByEl.listFiles());
        _queryResult.addAll(_mth_listFilesResult);
        _il++;
    }
    return _queryResult;
}
```

### 6.5.2.8 Operator tworzenia obiektu

#### 6.5.2.8.1 Wariant z konstruktorem bez parametrów



#### Semantyka:

Operator tworzy nowy obiekt klasy zwróconej przez zapytanie. Klasa ta musi posiadać konstruktor bezargumentowy.

#### Przykłady:

```
new Address()
new java.util.Date()
```

### Kontrola typologiczna:

Sygnatura argumentu	Zapytanie	Wynik
Liczność	1..1	1..1
Typ kolekcji	element	element
Typ obiektu	java.lang.Class<T>	T

### Generowany kod Java dla jednego z przykładów:

```
/** query='new java.util.Date()'
 */
public Date executeQuery() {
    Date _queryResult = new java.util.Date();

    return _queryResult;
}
```

### 6.5.2.8.2 Wariant z konstruktorem z parametrami



### Semantyka:

Operator tworzy nowy obiekt klasy zwróconej przez lewe zapytanie. Klasa ta musi posiadać konstruktor z odpowiednią liczbą argumentów odpowiadających typologicznie przekazanym parametrom. Prawe zapytanie powinno zwrócić strukturę parametrów przekazanych do konstruktora.

### Przykłady:

```
new Address(city, street, postCode)
new java.util.Date(System.currentTimeMillis())
```

### Kontrola typologiczna dla konstruktora z jednym parametrem:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	1..1	Tak jak oczekiwany parametr konstruktora	1..1
Typ kolekcji	element		element
Typ obiektu	java.lang.Class<T>		T

### Kontrola typologiczna dla konstruktora wieloma parametrami:

Sygnatura argumentu	Lewe zapytanie	Prawe zapytanie	Wynik
Liczność	1..1	1..1	1..1
Typ kolekcji	element	element	element
Typ obiektu	java.lang.Class<T>	pl.wcislo.sbc14 j.java.model.ru ntime.Struct	T

### Generowany kod Java dla jednego z przykładów:

```
/** query='new Address(city, street, postCode)'
 */
public Address executeQuery() {
    pl.wcislo.sq4j.java.model.runtime.Struct _commaResult =
        OperatorUtils.cartesianProductSS(city, street, "", "");
    pl.wcislo.sq4j.java.model.runtime.Struct _commaResult1 =
        OperatorUtils.cartesianProductSS(_commaResult, postCode, "", "");
    Address _queryResult = new Address(
        (String) _commaResult1.getValue(0),
        (String) _commaResult1.getValue(1),
        (String) _commaResult1.getValue(2)
    );

    return queryResult;
}
```

## 7 Kierunki rozwoju języka SBQL4J

W tym rozdziale zostaną przedstawione propozycje rozwoju projektu SBQL4J o nowe, uznane przez autora za interesujące, funkcjonalności.

### 7.1 SBQL4J jako uniwersalne API do baz danych i mechanizmów wyszukiwania

Język SBQL4J umożliwia wykonywanie zapytań na obiektach Javy. Dużym udogodnieniem byłoby umożliwienie odpytywania innych źródeł danych. Takimi źródłami mogłyby być: bazy danych, repozytoria i dokumenty XML, usługi katalogowe, mechanizmy wyszukiwarek internetowych. Mogłoby odbywać się to z uwzględnieniem mocnej kontroli typów, podobnie jak ma to miejsce obecnie. Warunkiem do tego byłoby oczywiście dostarczenie meta-modelu danego źródła w czasie analizy zapytania. Podobny mechanizm rozszerzeń został zaimplementowany dla języka LINQ. Rozpatrując architekturę tego rozwiązania, należy rozważyć następujące warianty:

#### 7.1.1 Wariant tłumaczenia wyrażeń SBQL4J na natywny język zapytań bazy

W tym wariantcie semantyka języka nie uległaby zmianie. Zapytania do nowych źródeł wyrażane byłyby za pomocą dotychczasowej, jednolitej składni SBQL4J. Zapytania te byłyby ostatecznie tłumaczone na natywne operacje uzyskania danych ze źródła, podobnie jak ma to miejsce w przypadku tłumaczenia zapytań SBQL4J na kod Javy. W ten sposób możliwe byłoby nawet integracja z bazami relacyjnym, dzięki zastosowaniu mechanizmu przepisywania zapytań SBQL na SQL opisanego w (Wiślicki, 2008). Podobne rozwiązanie polegające na przepisywaniu funkcjonuje obecnie w LINQ. Należy jednak stwierdzić, że może okazać się ono nieadekwatne do wszystkich źródeł danych. Dla niektórych z nich mogą okazać się przydatne nowe operatory języka, których nie ma w standardowej składni. Z kolei rozszerzenie składni SBQL4J o nowy operator wiąże się ze sporymi zmianami i nie mogłoby być dostarczone w ramach dodatkowego API, w odróżnieniu od implementacji mechanizmu zapytań dla nowego źródła danych.

#### 7.1.2 Wariant natywnych zapytań bazy połączonych z wyrażeniami SBQL4J z mocną kontrolą typologiczną – podzapytania niezależne semantycznie

W tym wariantcie zastosowano by architekturę pozwalającą na niezależne parsowanie i przetwarzanie fragmentów zapytania, odnoszących się do nowych źródeł danych. Tego typu rozwiązanie pozwalałoby na rozszerzenie języka zapytań o nowe konstrukcje, np. operatory. Byłoby to w istocie wprowadzenie możliwości rozbudowy SBQL4J o kolejne pod-języki, podobnie jak w ramach projektu język Java został rozbudowany o język zapytań. Dużą zaletą byłaby wyjątkowa elastyczność architektury dająca szerokie możliwości integracyjne. Oczywiście, integracja składniowa jest tylko jednym z elementów szeroko rozumianej integracji systemów informatycznych. Więcej informacji na ten temat w kontekście języka SBQL zawiera praca (Lentner, 2008).

## 7.2 Zapytania *ad-hoc* i kontrola typologiczna w czasie wykonania programu

Obecnie zapytania SBQL4J do poprawnego działania wymagają analizy w czasie kompilacji, min. w celu określenia zmiennych z kontekstu języka Java użytych w zapytaniu. Mechanizm kontroli typów jest związany z mechanizmem kompilatora Java, co uniemożliwia analizę zapytań powstałych w trakcie działania programu. Z kolei umożliwienie działania takim zapytaniom znacznie poszerzyłoby zakres potencjalnych zastosowań języka. Możliwe byłyby zapytania *ad-hoc* tworzone przez użytkownika w trakcie działania programu, lub generowane przez program. W celu realizacji tego zadania należałoby min. rozszerzyć mechanizm kontroli typów o współpracę z mechanizmem RTTI w celu uzyskania informacji o typach obiektów w trakcie wykonania programu.

## 7.3 Optymalizacja zapytań

W ramach prac nad językiem SBQL opracowano wiele mechanizmów optymalizacji zapytań. Niektóre z nich zostały opisane w pracy (Płodzień, 2000). W celu poprawy wydajności wskazane jest zastosowanie przynajmniej niektórych z nich. Implementacja SBQL4J jest do tego przygotowana, ponieważ jest w niej rozwinięty mechanizm analizy zapytań i kontroli typologicznej, co umożliwia zastosowanie wielu technik optymalizacji zapytań przez przepisywanie.

## 7.4 Zintegrowane środowisko programistyczne (IDE) dla SBQL4J

Obecnie nie istnieje żaden edytor, który umożliwiłby tworzenie zapytań SBQL4J w sposób przyjazny dla użytkownika. Celowe wydaje się stworzenie takiego edytora, najlepiej w ramach jednego z popularnych zintegrowanych środowisk programistycznych (Integrated Development Environment, IDE). Taki edytor byłby rozszerzeniem edytora plików źródłowych Java, ze względu na integrację składniową Java z SBQL4J.

## 8 Wykorzystane narzędzia

W tym rozdziale zostaną krótko opisane najważniejsze narzędzia użyte do realizacji projektu SBQL4J.

### 8.1 Skaner JFlex

Jest to generator analizatorów leksykalnych (lub inaczej generator skanerów) stworzony dla języka Java. Jego autorem jest Gerwin Klein. Stanowi on odpowiednik popularnych generatorów skanerów Lex/Flex dla języka C/C++. Jego cechą od pierwszych wersji jest ścisła integracja z generatorem parserów CUP. Podczas prac implementacyjnych JFlex został on użyty do stworzenia składni zapytań SBQL4J.

JFlex jest dostępny na licencji open-source na stronie domowej projektu, pod adresem: <http://jflex.de/>

### 8.2 Parser Cup

Rozwijany od 1996 roku projekt CUP stanowi implementację generatora parserów LALR dla języka Java. Pierwszymi autorami biblioteki byli Scott Hudson, Frank Flannery i C. Scott Ananian. Od 1999 roku opieką nad projektem zajmuje się Monachijski Uniwersytet Techniczny. Projekt jest wzorowany na popularnym generatorze parserów YACC. Gramatyka zapytań SBQL4J została stworzona za jego pomocą.

CUP jest projektem open-source, strona domowa projektu znajduje się pod adresem:

<http://www2.cs.tum.edu/projects/cup/>

### 8.3 Kompilator OpenJDK

Począwszy od 2006 roku firma Sun Microsystems, twórca języka Java, udostępnia swoją implementację języka w postaci projektu open-source. Jeden z podprojektów OpenJDK – Compiler został wykorzystany w implementacji SBQL4J jako źródło informacji o typach podczas statycznej analizy zapytań.

Strona domowa projektu znajduje się pod adresem:

<http://openjdk.java.net/>

### 8.4 Środowisko programistyczne Eclipse

Eclipse jest platformą napisaną w Javie do tworzenia aplikacji okienkowych. Na jej podstawie stworzono zintegrowane środowisko programistyczne (ang. integrated development environment, IDE). Eclipse zostało stworzone w laboratoriach firmy IBM, a następnie zostało udostępnione i jest rozwijane na licencji open-source. Może być użyte do tworzenia aplikacji w języku Java, oraz w innych językach programowania. Istotną cechą platformy jest tzw. mechanizm „wtyczek” pozwalający na łatwy rozwój możliwości środowiska, min. na wsparcie programowania w językach innych niż Java. Środowisko Eclipse zostało użyte w celu usprawnienia prac implementacyjnych projektu SBQL4J.

Strona domowa projektu:

<http://www.eclipse.org/>

### 8.5 Biblioteka CGLib

CGLib jest biblioteką generowania bajtkodu Javy w czasie wykonania programu. Umożliwia dynamiczne rozszerzanie klas i interfejsów. W projekcie SBQL4J biblioteka ta została wykorzystana do wiązania nazw z obiektów Java, jako bardziej wydajna alternatywa mechanizmu refleksji. Ten sposób wiązania nazw jest wykorzystany podczas przetwarzania nazw za pomocą interpretera.

Strona domowa projektu:

<http://cglib.sourceforge.net/>

### 8.6 Biblioteka ASM

ASM jest niskopoziomową biblioteką generowania bajtkodu. W porównaniu do CGLib cechuje się małym rozmiarem i większą wydajnością.

Strona domowa projektu:

<http://asm.ow2.org/>

### 8.7 Biblioteka Jalopy

Jalopy jest biblioteką służącą do formatowania kodu Java. Za jej pomocą formatowany jest wygenerowany kod zapytań SBQL4J.

Strona domowa projektu:

<http://jalopy.sourceforge.net/>

### 8.8 Narzędzie skryptowe ANT

ANT jest narzędziem skryptowym napisanym w Javie, pozwalającym min. na zautomatyzowanie procesu budowania aplikacji. W ramach projektu SBQL4J zostało stworzone rozszerzenie, umożliwiające prekompilację źródeł z zapytaniami za pomocą skryptu ANT.

Strona domowa projektu:

<http://ant.apache.org/>

### 8.9 Serwer SVN

SVN, zwany też *Subversion*, jest systemem kontroli wersji umożliwiającym współdzielenie plików. Działa on w architekturze klient-serwer, pozwala na bezpieczne przechowywanie plików źródłowych programu oraz monitorowanie zmian. Jest on wzorowany na systemie CVS, popularnym systemie kontroli wersji w ostatnich latach. W projektach, w których uczestniczy wielu programistów, jest on niezastąpiony przy pracy

nad wspólnym kodem źródłowych. W projekcie SBQL4J system SVN został użyty do przechowywania bezpiecznej kopii źródeł tworzonej implementacji, oraz monitorowania zmian. Ostatecznie kod projektu został udostępniony do wglądu pod adresem: <https://sbql4j.googlecode.com/svn>.

Jako klient do serwera SVN została użyte rozszerzenie środowiska Eclipse – Subversive. Jej strona domowa to:

<http://www.eclipse.org/subversive>

## 8.10 Witryna GoogleCode

Witryna GoogleCode (<http://code.google.com/>) umożliwia publikację i zarządzanie projektem informatycznym. Udostępnia ona darmowy serwer SVN, pozwala na publikację dowolnych plików (np. skompilowanych programów), oraz dokumentacji w postaci stron WWW w stylu Wikipedii. Pozwala również na jednoczesną pracę nad projektem wielu użytkowników. Administrator projektu może zaprosić do niego dowolne osoby i przypisać im zadania i uprawnienia w projekcie. Dostępny jest także moduł zgłaszania błędów.

Projekt SBQL4J za zgodą władz uczelni został opublikowany na licencji open-source, dostępny jest w witrynie GoogleCode pod adresem: <http://code.google.com/p/sbql4j/>

## 9 Podsumowanie

Rozbudowa popularnego języka programowania o konstrukcje zapytań jest zadaniem niosącym duże korzyści i otwierającym nowe perspektywy programistom i architektom oprogramowania. Jest wyzwaniem technicznym stawiającym przed twórcą szereg problemów i zmuszającym do podjęcia wielu trudnych decyzji projektowych. W wyniku niniejszej pracy stworzono udaną implementację tego zadania, przy okazji znaleziono rozwiązania wielu problemów dotyczących integracji różnych od siebie języków. Stworzona implementacja jest w pełni funkcjonalna i rozwiązuje wiele problemów, z którymi dotychczas musieli borykać się programiści. Język SBQL4J wnosi nie tylko nowe elementy do języka Java, ale w unikalny sposób pozwala łączyć ze z istniejącymi konstrukcjami języka. Powstała w ten sposób synergia pozwala na uzyskanie niespotykanej dotychczas przejrzystości, wydajności, niezawodności, rozszerzalności i pielęgnacyjności oprogramowania związanego z przetwarzaniem skomplikowanych struktur danych. Stworzone rozwiązanie jest interesującą alternatywą dla języka LINQ – docenianego i popularnego języka dla technologii .NET.

Język SBQL4J jest w stanie spopularyzować podejście stosowe w dziedzinie języków zapytań. W odróżnieniu od wielu poprzednich prototypów opartych na tym podejściu, SBQL4J może być z łatwością wykorzystany w istniejących projektach opartych istniejących produktach informatycznych. Ryzyko projektowe wynikające z jego użycia zostało maksymalnie zminimalizowane, choćby ze względu na pełną kompatybilność z istniejącymi środowiskami wykonania programów Java oraz przejrzystość kodu wygenerowanego na podstawie zapytań.

Stworzona implementacja rozszerza koncepcję SBQL o nowe cechy, min. o innowacyjne podejście do sortowania obiektów. Projekt ma bardzo duży potencjał rozwoju, szczególnie w dziedzinie integracji różnych źródeł danych, co zasygnalizowano w poprzednich rozdziałach.

## Bibliografia

**Ananian, Scott. 2002.** CUP grammar for the Java programming language. [Online] 2002. <http://www2.cs.tum.edu/projects/cup/>.

**Lentner, Michał. 2008.** Integracja danych i aplikacji przy użyciu wirtualnych repozytoriów. *praca doktorska*. Warszawa : Polsko-Japońska Wyższa Szkoła Technik Komputerowych, Katedra Inżynierii Oprogramowania, 2008.

**Płodzień, Jacek. 2000.** Optimization Methods in Object Query Languages. *praca doktorska*. Warszawa : Instytut Podstaw Informatyki, Polska Akademia Nauk, 2000.

**Soni, Rahul.** LINQ Performance - LINQ to Collection. [Online] <http://www.dotnetcraps.com/dotnetcraps/post/LINQ-Performance-Part-1-LINQ-to-Collection.aspx>.

**Stencel, Krzysztof. 2006.** *Półmocna kontrola typów w językach programowania baz danych*. Warszawa : Wydawnicwo PJWSTK, 2006.

**Subieta, Kazimierz. 2004.** *Teoria i konstrukcja obiektowych języków zapytań*. Warszawa : Wydawnictwo PJWSTK, 2004.

**Wiślicki, Jacek. 2008.** An object-oriented wrapper to relational databases with query optimisation. *Praca doktorska*. Łódź : Politechnika Łódzka, Wydział Elektrotechniki, Elektroniki, Informatyki i Automatyki, Katedra Informatyki Stosowanej, 2008.

## Spis ilustracji

Rysunek 1 - Przykładowy obiekt atomowy .....	19
Rysunek 2 - przykładowy obiekt złożony .....	19
Rysunek 3 - przykładowy obiekt referencyjny .....	19
Rysunek 4 - architektura działania języka SBQL4J .....	29
Rysunek 5 - opis działania preprocesora .....	30
Rysunek 6 - opis kompilacji kompilatorem Java .....	30

## **Dodatek A: Słownik użytej terminologii i skrótów**

AST - abstract syntax tree  
bytecode - kod pośredni Java  
class - klasa  
collection - kolekcja  
ENVS - Environmental Stack  
expression - wyrażenie  
garbage collector - odśmieczacz pamięci  
JavaEE - Java Enterprise Edition  
JavaME - Java Micro Edition  
JavaSE - Java Standard Edition  
JCP - Java Community Process  
JIT - just-in-time  
JSR - Java Specification Request  
LALR - Look Ahead Left to right, identifying the Rightmost production  
LINQ – Language INtegrated Query  
object - obiekt  
QRES - Query Result Stack  
RTTI - run-time type information  
SBA - Stack-Based Approach  
SBQL - Stack-Based Query Language  
SBQL4J - Stack-Based Query Language for Java  
signature - sygnatura  
SQL – Structured Query Language  
struct - struktura  
typechecker - kontroler zgodności typów  
wrapper - osłona  
XML – eXtensible Markup Language

## Dodatek B: Słowa kluczowe języka SBQL4J

Słowo kluczowe	Znaczenie
\$index	Zmienna pomocnicza dla operatorów iterujących po elementach kolekcji
all	Kwantyfikator uniwersalny
and	Operator logiczny
any	Kwantyfikator egzystencjalny
as	Operator nazwy pomocniczej
asc	Klauzula operatora sortowania
avg	Operator wyliczania średniej wartości
bag	Operator tworzenia bagu
close by	Operator tranzytywnego domknięcia
count	Operator zliczania
desc	Klauzula operatora sortowania
exists	Funkcja sprawdzająca, czy liczność argumentu jest większa od 0
foreach	Operator iteracji
group as	Operator pomocniczej nazwy
in	Operator zawierania
instanceof	Operator sprawdzenia typu
intersect	Operator przecięcia zbiorów
join	Operator złączenia
max	Operator wartości maksymalnej
min	Operator wartości minimalnej
minus	Operator różnicy zbiorów
new	Operator tworzenia obiektu
not	Funkcja logiczna
or	Funkcja logiczna
order by	Operator sortowania
sequence	Operator tworzenia sekwencji
struct	Operator tworzenia struktury
sum	Operator sumy
union	Operator sumy zbiorów
unique	Operator usuwania duplikatów
using	Klauzula operatora sortowania
where	Operator selekcji warunkowej

## Dodatek C: Gramatyka języka SBQL4J

Jest to plik definicji gramatyki dla parsera CUP:

```
package pl.wcislo.sqql4j.lang.parser;

import java.io.StringReader;
import java.util.ArrayList;
import java.util.List;

import java_cup.runtime.Symbol;
import pl.wcislo.sqql4j.exception.ParserException;
import pl.wcislo.sqql4j.lang.parser.expression.AsExpression;
import pl.wcislo.sqql4j.lang.parser.expression.BinaryAExpression;
import pl.wcislo.sqql4j.lang.parser.expression.CloseByExpression;
import pl.wcislo.sqql4j.lang.parser.expression.ComaExpression;
import pl.wcislo.sqql4j.lang.parser.expression.ConditionalExpression;
import pl.wcislo.sqql4j.lang.parser.expression.ConstructorExpression;
import pl.wcislo.sqql4j.lang.parser.expression.DerefExpression;
import pl.wcislo.sqql4j.lang.parser.expression.DotExpression;
import pl.wcislo.sqql4j.lang.parser.expression.Expression;
import pl.wcislo.sqql4j.lang.parser.expression.ForEachExpression;
import pl.wcislo.sqql4j.lang.parser.expression.ForallExpression;
import pl.wcislo.sqql4j.lang.parser.expression.ForanyExpression;
import pl.wcislo.sqql4j.lang.parser.expression.GroupAsExpression;
import pl.wcislo.sqql4j.lang.parser.expression.IdentifierExpression;
import pl.wcislo.sqql4j.lang.parser.expression.JavaParamExpression;
import pl.wcislo.sqql4j.lang.parser.expression.JoinExpression;
import pl.wcislo.sqql4j.lang.parser.expression.LiteralExpression;
import pl.wcislo.sqql4j.lang.parser.expression.MethodExpression;
import pl.wcislo.sqql4j.lang.parser.expression.OrderByExpression;
import pl.wcislo.sqql4j.lang.parser.expression.OrderByParamExpression;
import pl.wcislo.sqql4j.lang.parser.expression.RangeExpression;
import pl.wcislo.sqql4j.lang.parser.expression.UnaryExpression;
import pl.wcislo.sqql4j.lang.parser.expression.WhereExpression;
import pl.wcislo.sqql4j.lang.parser.expression.OrderByParamExpression.SortType;
import pl.wcislo.sqql4j.lang.parser.terminals.Identifier;
import pl.wcislo.sqql4j.lang.parser.terminals.operators.OperatorFactory;

parser code {
    public Lexer lexer;
    private String expr;
    public Expression RESULT;
    public List<JavaParamExpression> javaParams = new
ArrayList<JavaParamExpression>();

    public ParserCup(String expr) {
        this.symbolFactory = new SBQLSymbolFactory();
        this.expr = expr;
    }

    void setResult(Expression exp) {
        this.RESULT = exp;
    }

    public void report_fatal_error(String message, Object info) throws ParserException
{
        Token token = (Token) info;
        throw new ParserException(message, token.left + 1, token.right + 1,
token.pos +
            1, token.value.toString());
    }

    public int getCurrentPos() {
        Object s = stack.peek();
        if(s instanceof pl.wcislo.sqql4j.lang.parser.Token) {
            pl.wcislo.sqql4j.lang.parser.Token t =
(pl.wcislo.sqql4j.lang.parser.Token)s;
            return t.pos;
        } else {
            return -1;
        }
    }
}
```

```

    public int getPosition(int tokenStackDistance) {
        int index = stack.size() - tokenStackDistance - 1;
        Object s = stack.get(index);
        if(s instanceof pl.wcislo.sbql4j.lang.parser.Token) {
            pl.wcislo.sbql4j.lang.parser.Token t =
(pl.wcislo.sbql4j.lang.parser.Token)s;
            return t.pos;
        } else {
            return -1;
        }
    }

    public void report_error(String message, Object info) {
        report_fatal_error(message, info);
    }

    public void unrecovered_syntax_error(Symbol symbol) {
        Token token = (Token)symbol;
        String tokenString = token.toString();
        if(token.value != null)
            tokenString += " (" + token.value + ")";
        report_fatal_error("Unexpected token " + tokenString, token);
    }

    public void syntax_error(Symbol symbol) {
        Token token = (Token)symbol;
        String tokenString = token.toString();
        if(token.value != null)
            tokenString += " (" + token.value + ")";
        report_fatal_error("Unexpected token " + tokenString, token);
    }

```

```
};
```

```

init with {:          lexer = new Lexer(new StringReader(expr)); :};
scan with {:          return lexer.next_token(); :};

```

```

terminal Integer INTEGER_LITERAL;
terminal Double DOUBLE_LITERAL;
terminal Boolean BOOLEAN_LITERAL;
terminal String IDENTIFIER;
terminal String GROUPBY_ASC;
terminal String GROUPBY_DESC;

```

```

terminal String STRING_LITERAL;
terminal Character CHAR_LITERAL;

```

```

terminal String PLUS;
terminal String MINUS;
terminal String MULTIPLY;
terminal String DIVIDE;
terminal String MODULO;
terminal String MORE;
terminal String EQUALS;
terminal String NOT_EQUALS;
terminal String UNIQUE;
terminal String UNION;
terminal String SUM;
terminal String AVG;
terminal String COUNT;
terminal String OR;
terminal String AND;
terminal String LESS;
terminal String MORE_OR_EQUAL;
terminal String LESS_OR_EQUAL;
terminal String MIN;
terminal String MAX;
terminal String COMA;
terminal String IN;
terminal String DOT;
terminal String WHERE;
terminal String Deref;
terminal String BAG;
terminal String SEQUENCE;
terminal String STRUCT;

```

```

terminal String ORDER_BY;
terminal String CLOSE_BY;

terminal String FORALL;
terminal String FORANY;

terminal String JOIN;
terminal String AS;
terminal String GROUP_AS;
terminal String NOT;
terminal String EXISTS;
terminal String MINUS_FUNCTION;
terminal String INTERSECT;
terminal String LEFT_ROUND_BRACKET;
terminal String RIGHT_ROUND_BRACKET;
terminal String LEFT_CURLY_BRACKET;
terminal String RIGHT_CURLY_BRACKET;
terminal String LEFT_BOX_BRACKET;
terminal String RIGHT_BOX_BRACKET;

terminal String SEMICOLON;

terminal String FOREACH;
terminal String NEW;
terminal String RANGE;
terminal String QUESTION_MARK;
terminal String COLON;
terminal String INSTANCEOF;
terminal String USING;

non terminal Expression goal;
non terminal Expression expr;
non terminal Expression expr_without_coma;

non terminal UnaryExpression bag_expr;
non terminal UnaryExpression sequence_expr;
non terminal Identifier identifier_literal;
non terminal MethodExpression method_expr;

non terminal LiteralExpression<?> literal_expression;
non terminal OrderByExpression order_by_expr;
non terminal List<OrderByParamExpression> order_by_param_list;
non terminal OrderByParamExpression order_by_param;
non terminal SortType sort_order;
non terminal List<Expression> exprs;
non terminal ConstructorExpression constr_expr;
non terminal DotExpression dot_expr;
non terminal DotExpression dot_ident_expr;
non terminal Expression range_expr;

precedence left Deref;
precedence left NEW;
precedence left WHERE, JOIN;
precedence left FORALL, FORANY;
precedence left NOT;
precedence left UNION, IN;
precedence left ORDER_BY, CLOSE_BY;
precedence left QUESTION_MARK, COLON;
precedence left COMA;
precedence left IDENTIFIER;
precedence left OR, AND;
precedence left EQUALS, NOT_EQUALS, MORE, LESS, MORE_OR_EQUAL, LESS_OR_EQUAL;
precedence left PLUS, MINUS;
precedence left MULTIPLY, DIVIDE, MODULO;
precedence left AS, GROUP_AS;
precedence left SUM, COUNT, AVG, MIN, MAX, UNIQUE, EXISTS, MINUS_FUNCTION, INTERSECT,
INSTANCEOF;
precedence left LEFT_ROUND_BRACKET, RIGHT_ROUND_BRACKET;
precedence left LEFT_BOX_BRACKET, RIGHT_BOX_BRACKET;
precedence left DOT;
precedence left BAG, SEQUENCE, STRUCT;
precedence left RANGE;
precedence left GROUPBY_ASC, GROUPBY_DESC;
precedence left SEMICOLON;
precedence left USING;

```

```

start with goal;
goal ::= expr:e          {: RESULT = e; parser.setResult(e); :} ;
expr ::=
    expr:e1 PLUS:o expr:e2{: RESULT = new
BinaryAExpression(parser.getPosition(1), e1, e2, OperatorFactory.getOperator(o)); :}
    | expr:e1 MINUS:o expr:e2      {: RESULT = new
BinaryAExpression(parser.getPosition(1), e1, e2, OperatorFactory.getOperator(o)); :}
    | expr:e1 MULTIPLY:o expr:e2   {: RESULT = new
BinaryAExpression(parser.getPosition(1), e1, e2, OperatorFactory.getOperator(o)); :}
    | expr:e1 DIVIDE:o expr:e2     {: RESULT = new
BinaryAExpression(parser.getPosition(1), e1, e2, OperatorFactory.getOperator(o)); :}
    | expr:e1 MODULO:o expr:e2     {: RESULT = new
BinaryAExpression(parser.getPosition(1), e1, e2, OperatorFactory.getOperator(o)); :}
    | expr:e1 EQUALS:o expr:e2     {: RESULT = new
BinaryAExpression(parser.getPosition(1), e1, e2, OperatorFactory.getOperator(o)); :}
    | expr:e1 NOT_EQUALS:o expr:e2 {: RESULT = new
BinaryAExpression(parser.getPosition(1), e1, e2, OperatorFactory.getOperator(o)); :}
    | expr:e1 MORE:o expr:e2       {: RESULT = new
BinaryAExpression(parser.getPosition(1), e1, e2, OperatorFactory.getOperator(o)); :}
    | expr:e1 OR:o expr:e2         {: RESULT = new
BinaryAExpression(parser.getPosition(1), e1, e2, OperatorFactory.getOperator(o)); :}
    | expr:e1 AND:o expr:e2        {: RESULT = new
BinaryAExpression(parser.getPosition(1), e1, e2, OperatorFactory.getOperator(o)); :}
    | expr:e1 UNION:o expr:e2      {: RESULT = new
BinaryAExpression(parser.getPosition(1), e1, e2, OperatorFactory.getOperator(o)); :}
    | dot_expr:e   {: RESULT = e; :}
    | expr:e1 JOIN:o expr:e2       {: RESULT = new
JoinExpression(parser.getPosition(1), e1, e2); :}

    | FORALL:o expr:e1 expr:e2     {: RESULT = new
ForallExpression(parser.getPosition(2), e1, e2); :}
    | FORANY:o expr:e1 expr:e2     {: RESULT = new
ForanyExpression(parser.getPosition(2), e1, e2); :}

    | SUM:o expr:e1   {: RESULT = new
UnaryExpression(parser.getPosition(1), e1, OperatorFactory.getOperator(o)); :}
    | UNIQUE:o expr:e1   {: RESULT = new
UnaryExpression(parser.getPosition(1), e1, OperatorFactory.getOperator(o)); :}
    | COUNT:o expr:e1   {: RESULT = new
UnaryExpression(parser.getPosition(1), e1, OperatorFactory.getOperator(o)); :}
    | AVG:o expr:e1   {: RESULT = new
UnaryExpression(parser.getPosition(1), e1, OperatorFactory.getOperator(o)); :}
    | MIN:o expr:e1   {: RESULT = new
UnaryExpression(parser.getPosition(1), e1, OperatorFactory.getOperator(o)); :}
    | MAX:o expr:e1   {: RESULT = new
UnaryExpression(parser.getPosition(1), e1, OperatorFactory.getOperator(o)); :}
    | bag_expr:e   {: RESULT = e; :}
//    | BAG:o expr:e1   {: RESULT = new
UnaryExpression(parser.getPosition(1), e1, OperatorFactory.getOperator(o)); :}
//    | BAG:o expr:e2 expr:e1 {: RESULT = new
UnaryExpression(parser.getPosition(1), e1, OperatorFactory.getOperator(o), e2); :}
    | sequence_expr:e   {: RESULT = e; :}
    | STRUCT:o expr:e1   {: RESULT = new
UnaryExpression(parser.getPosition(1), e1, OperatorFactory.getOperator(o)); :}
    | expr:e1 MINUS_FUNCTION:o expr:e2   {: RESULT = new
BinaryAExpression(parser.getPosition(1), e1, e2, OperatorFactory.getOperator(o)); :}
    | NOT:o expr:e1   {: RESULT = new
UnaryExpression(parser.getPosition(1), e1, OperatorFactory.getOperator(o)); :}
    | EXISTS:o expr:e1   {: RESULT = new
UnaryExpression(parser.getPosition(1), e1, OperatorFactory.getOperator(o)); :}
    | Deref:o expr:e1   {: RESULT = new
DerefExpression(parser.getPosition(1), e1); :}
    | expr:e1 AS:o identifier_literal:l {: RESULT = new
AsExpression(parser.getPosition(1), e1, l); :}
    | expr:e1 GROUP_AS:o identifier_literal:l {: RESULT = new
GroupAsExpression(parser.getPosition(1), e1, l); :}
    | expr:e1 WHERE:o expr:e2   {: RESULT = new
WhereExpression(parser.getPosition(1), e1, e2); :}
    | expr:e1 LESS:o expr:e2   {: RESULT = new
BinaryAExpression(parser.getPosition(1), e1, e2, OperatorFactory.getOperator(o)); :}
    | expr:e1 MORE_OR_EQUAL:o expr:e2   {: RESULT = new
BinaryAExpression(parser.getPosition(1), e1, e2, OperatorFactory.getOperator(o)); :}
    | expr:e1 LESS_OR_EQUAL:o expr:e2   {: RESULT = new
BinaryAExpression(parser.getPosition(1), e1, e2, OperatorFactory.getOperator(o)); :}
    | expr:e1 INSTANCEOF:o expr:e2   {: RESULT = new
BinaryAExpression(parser.getPosition(1), e1, e2, OperatorFactory.getOperator(o)); :}

```

```

    |      expr:e1 COMA:o      expr:e2      {: RESULT = new
ComaExpression(parser.getPosition(1), e1, e2); :)
    |      expr:e1 IN:o expr:e2  {: RESULT = new
BinaryAExpression(parser.getPosition(1), e1, e2, OperatorFactory.getOperator(o)); :)
    |      expr:e1 INTERSECT:o expr:e2  {: RESULT = new
BinaryAExpression(parser.getPosition(1), e1,e2,  OperatorFactory.getOperator(o)); :)
    |      literal_expression:l  {: RESULT = l; :)
    |      identifier_literal:l  {: RESULT = new IdentifierExpression(l.pos, l); :)
|      LEFT_ROUND_BRACKET:o expr:e1 RIGHT_ROUND_BRACKET {: RESULT = e1; :)
    |      expr:e1 CLOSE_BY:o expr:e2 {: RESULT = new
CloseByExpression(parser.getPosition(1), e1, e2); :)
    |      FOREACH:o expr:e1 LEFT_CURLY_BRACKET exprs:es RIGHT_CURLY_BRACKET {: RESULT
= new ForEachExpression(e1.position, e1, es); :)
    |      constr_expr:e  {: RESULT = e; :)
    |      method_expr:e  {: RESULT = e; :)
    |      expr:e1 LEFT_BOX_BRACKET:o range_expr:e2 RIGHT_BOX_BRACKET {: RESULT = new
BinaryAExpression(parser.getPosition(2), e1, e2,  OperatorFactory.getOperator(o)); :)
    |      expr:condExpr QUESTION_MARK expr:trueExpr COLON expr:falseExpr {: RESULT =
new ConditionalExpression(parser.getPosition(3), condExpr, trueExpr, falseExpr); :)
    |      order_by_expr:e  {: RESULT = e; :)
;

range_expr ::=
    expr:e1                                     {: RESULT = e1; :)
    | expr:e1 RANGE expr:e2                       {: RESULT = new
RangeExpression(parser.getPosition(1), e1, e2); :)
    | expr:e1 RANGE MULTIPLY                       {: RESULT = new
RangeExpression(parser.getPosition(1), e1); :)
;

bag_expr ::= BAG:o LESS IDENTIFIER:i MORE expr:e1 {: RESULT = new
UnaryExpression(parser.getCurrentPos(), e1, OperatorFactory.getOperator(o), new
IdentifierExpression(parser.getCurrentPos(), new Identifier(i, parser.getCurrentPos())));
:}
    |
    BAG:o expr:e1 {: RESULT = new
UnaryExpression(parser.getCurrentPos(), e1, OperatorFactory.getOperator(o)); :)
;

sequence_expr ::= SEQUENCE:o LESS IDENTIFIER:i MORE expr:e1 {: RESULT = new
UnaryExpression(parser.getCurrentPos(), e1, OperatorFactory.getOperator(o), new
IdentifierExpression(parser.getCurrentPos(), new Identifier(i, parser.getCurrentPos())));
:}
    |
    SEQUENCE:o expr:e1 {: RESULT = new
UnaryExpression(parser.getCurrentPos(), e1, OperatorFactory.getOperator(o)); :)
;

dot_expr ::= expr:e1 DOT:o expr:e2  {: RESULT = new DotExpression(parser.getCurrentPos(),
e1, e2); :};
dot_ident_expr ::= identifier_literal:i1 DOT:o identifier_literal:i2 {: RESULT = new
DotExpression(parser.getCurrentPos(), new IdentifierExpression(parser.getCurrentPos(),
i1), new IdentifierExpression(parser.getCurrentPos(), i2)); :}
    | identifier_literal:i1 DOT:o dot_ident_expr:i2 {: RESULT =
new DotExpression(parser.getCurrentPos(), new IdentifierExpression(parser.getCurrentPos(),
i1), i2); :}
;

exprs ::= {: RESULT = new ArrayList<Expression>(); :}
    |      exprs:es expr:e1 SEMICOLON {: es.add(e1); RESULT=es; :}
;

identifier_literal ::=
    IDENTIFIER:l  {: RESULT = new Identifier(l, parser.getCurrentPos()); :}
;

literal_expression ::=
    INTEGER_LITERAL:l  {: RESULT = new
LiteralExpression<Integer>(parser.getCurrentPos(), (Integer)l); :}
    |      DOUBLE_LITERAL:l  {: RESULT = new
LiteralExpression<Double>(parser.getCurrentPos(), (Double)l); :}
    |      BOOLEAN_LITERAL:l  {: RESULT = new
LiteralExpression<Boolean>(parser.getCurrentPos(), (Boolean)l); :}
    |      STRING_LITERAL:l  {: RESULT = new
LiteralExpression<String>(parser.getCurrentPos(), (String)l); :}
    |      CHAR_LITERAL:l  {: RESULT = new
LiteralExpression<Character>(parser.getCurrentPos(), (Character)l); :}
;

```

```

method_expr ::= identifier_literal:l LEFT_ROUND_BRACKET RIGHT_ROUND_BRACKET  {:
    String mName = l.val;
    int pos = parser.getCurrentPos();
    RESULT = new MethodExpression(pos, mName, null);
    :}
| identifier_literal:l LEFT_ROUND_BRACKET expr:e1 RIGHT_ROUND_BRACKET
  {:
    String mName = l.val.substring(0, l.val.length()).trim();
    int pos = parser.getCurrentPos();
    RESULT = new MethodExpression(pos, mName, e1);
    :}
;

constr_expr ::=
    NEW dot_ident_expr:classNameExpr LEFT_ROUND_BRACKET RIGHT_ROUND_BRACKET  {:
        RESULT = new ConstructorExpression(parser.getPosition(3),
classNameExpr, null);
    :}
| NEW dot_ident_expr:classNameExpr LEFT_ROUND_BRACKET expr:paramsExpr
RIGHT_ROUND_BRACKET  {:
    RESULT = new ConstructorExpression(parser.getPosition(4),
classNameExpr, paramsExpr);
    :}
| NEW IDENTIFIER:className LEFT_ROUND_BRACKET RIGHT_ROUND_BRACKET  {:
    RESULT = new ConstructorExpression(parser.getPosition(3), className,
null);
    :}
| NEW IDENTIFIER:className LEFT_ROUND_BRACKET expr:paramsExpr
RIGHT_ROUND_BRACKET  {:
    RESULT = new ConstructorExpression(parser.getPosition(4), className,
paramsExpr);
    :}
;

order_by_expr ::=
    expr:e1 ORDER_BY:o order_by_param_list:e2 {: RESULT = new
OrderByExpression(parser.getCurrentPos(), e1, e2); :}

order_by_param_list ::=
    order_by_param:p1 {:
        List<OrderByParamExpression> list = new
ArrayList<OrderByParamExpression>();
        list.add(p1);
        RESULT = list;
    :}
| order_by_param_list:p2 SEMICOLON:o order_by_param:p1 {:
    p2.add(p1);
    RESULT = p2;
    :}
;

order_by_param ::=
    expr:e1 sort_order:s{: RESULT = new OrderByParamExpression(e1.position, e1, s,
null); :}
| expr:e1 sort_order:s USING expr:comparatorExpr {: RESULT = new
OrderByParamExpression(e1.position, e1, s, comparatorExpr); :}
;

sort_order ::=
    {: RESULT = SortType.ASC; :}
| GROUPBY_ASC {: RESULT = SortType.ASC; :}
| GROUPBY_DESC {: RESULT = SortType.DESC; :}
;

```

## Dodatek D: Porównanie zapytań SBQL4J z zapytaniami w języku LINQ

Poniższy dodatek zawiera kod zapytań będący porównaniem możliwości SBQL4J i LINQ. W komentarzach zawarto tekst oryginalnego zapytania LINQ dla języka C#.

```

/**
 * This sample uses where to find all elements of an array less than 5.
 *
 * Original LINQ query:
 * from n in numbers
 * where n < 5
 * select n;
 */
public void linq1() {
    int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
    List<Integer> lowNums = #{
        numbers as n
        where n < 5
    };
    System.out.println("Numbers < 5:");
    System.out.println(lowNums);
}

/**
 * This sample uses where to find all products that are out of stock.
 *
 * Original LINQ query:
 * from p in products
 *   where p.UnitsInStock == 0
 *   select p;
 */
public void linq2() {
    List<Product> products = getProductList();
    List<Product> soldOutProducts = #{
        products
        where unitsInStock == 0
    };
    System.out.println("Sold out products:");
    for(Product product : soldOutProducts) {
        System.out.printf("%s is sold out!\n", product.productName);
    }
}

/**
 * This sample uses where to find all products that are in stock and cost more than
 * 3.00 per unit.
 *
 * Original LINQ query:
 * from p in products
 *   where p.UnitsInStock > 0 && p.UnitPrice > 3.00M
 *   select p;
 */
public void linq3() {
    List<Product> products = getProductList();
    List<Product> expensiveInStockProducts = #{
        products
        where unitsInStock > 0 and unitPrice > 3.00
    };
    System.out.println("In-stock products that cost more than 3.00:");
    for(Product product : expensiveInStockProducts) {
        System.out.printf("%s is in stock and costs more than 3.00.\n",
product.productName);
    }
}

/**
 * This sample uses where to find all customers in Washington and then
 * uses the resulting sequence to drill down into their orders.
 *
 * Original LINQ query:
 * from c in customers
 *   where c.Region == "WA"
 *   select c;

```

```

*/
public void linq4() {
    List<Customer> customers = getCustomerList();
    List<Customer> waCustomers = #{
        customers where region == "WA"
    };
    System.out.println("Customers from Washington and their orders:");
    for(Customer customer : waCustomers) {
        System.out.printf("Customer %s: %s\n", customer.customerID,
customer.companyName);
        for(Order order : customer.orders) {
            System.out.printf("  Order %s: %s\n", order.orderID,
order.orderDate);
        }
    }
}

/**
 * This sample demonstrates an indexed Where clause that
 * returns digits whose name is shorter than their value.
 *
 * Original LINQ query:
 * digits.Where((digit, index) => digit.Length < index);
 */
public void linq5() {
    String[] digits = { "zero", "one", "two", "three", "four", "five", "six", "seven",
"eight", "nine" };
    List<String> shortDigits = #{
        digits where length() < $index
    };
    System.out.println("Short digits:");
    for(String d : shortDigits) {
        System.out.printf("The word %s is shorter than its value.\n", d);
    }
}

/**
 * This sample uses select to produce a sequence of
 * ints one higher than those in an existing array of ints.
 *
 * Original LINQ query:
 * from n in numbers
 *   select n + 1;
 */
public void linq6() {
    int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
    List<Integer> numsPlusOne = #{
        (numbers as n).
        (n + 1)
    };
    System.out.println("Numbers + 1:");
    for(Integer i : numsPlusOne) {
        System.out.println(i);
    }
}

/**
 * This sample uses select to return a sequence of just the names of a list of
 products.
 *
 * Original LINQ query:
 * from p in products
 *   select p.ProductName;
 */
public void linq7() {
    List<Product> products = getProductList();
    List<String> productNames = #{
        products.productName
    };
    System.out.println("Product Names:");
    for(String productName : productNames) {
        System.out.println(productName);
    }
}

/**
 * This sample uses select to produce a sequence of strings representing

```

```

* the text version of a sequence of ints.
*
* Original LINQ query:
* from n in numbers
*   select strings[n];
*/
public void linq8() {
    int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
    String[] strings = { "zero", "one", "two", "three", "four", "five", "six",
"seven", "eight", "nine" };
    List<String> textNums = #{
        (numbers as n).
        (strings[n])
    };
    System.out.println("Number strings:");
    for(String s : textNums) {
        System.out.println(s);
    }
}

/**
* This sample uses select to produce a sequence of the uppercase
* and lowercase versions of each word in the original array.
*
* Original LINQ query:
* from w in words
*   select new {Upper = w.ToUpper(), Lower = w.ToLower()};
*/
public void linq9() {
    String[] words = { "aPPLE", "BlUeBeRrY", "cHeRry" };
    List<Struct> upperLowerWords = #{
        (words).
        (toLowerCase() as upper, toUpperCase() as lower)
    };
    for(Struct ul : upperLowerWords) {
        System.out.printf("Uppercase: %s, Lowercase: %s\n", ul.get("upper"),
ul.get("lower"));
    }
}

/**
* This sample uses select to produce a sequence containing text
* representations of digits and whether their length is even or odd.
*
* Original LINQ query:
* from n in numbers
*   select new {Digit = strings[n], Even = (n % 2 == 0)};
*/
public void linq10() {
    int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
    String[] strings = { "zero", "one", "two", "three", "four", "five", "six",
"seven", "eight", "nine" };
    List<Struct> digitOddEvens = #{
        (numbers as n).
        (strings[n] as digit, (n % 2 == 0) as even)
    };
    for(Struct d : digitOddEvens) {
        System.out.printf("The digit %s is %s.\n", d.get("digit"),
((Boolean)d.get("even")) ? "even" : "odd");
    }
}

/**
* This sample uses select to produce a sequence containing some properties
* of Products, including UnitPrice which is renamed to Price
* in the resulting type.
*
* Original LINQ query:
* from p in products
*   select new {p.ProductName, p.Category, Price = p.UnitPrice};
*/
public void linq11() {
    List<Product> products = getProductList();
    List<Struct> productInfos = #{
        (products).
        (productName as productName, category as category, unitPrice as price)
    };
}

```

```

    };
    System.out.println("Product Info:");
    for(Struct productInfo : productInfos) {
        System.out.printf("%s is in the category %s and costs %s per unit.\n",
productInfo.getValue(0), productInfo.getValue(1), productInfo.getValue(2));
    }
}

/**
 * This sample uses an loop index to determine if the value of ints
 * in an array match their position in the array.
 *
 * Original LINQ query:
 * numbers.Select((num, index) => new {Num = num, InPlace = (num == index)});
 */
public void linq12() {
    int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
    List<Struct> numsInPlace = #{
        (numbers as num).
        (num as num, (num == $index) as inPlace)
    };
    System.out.println("Number: In-place?");
    for(Struct n : numsInPlace) {
        System.out.printf("%s: %s\n", n.get("num"), n.get("inPlace"));
    }
}

/**
 * This sample combines select and where to make a simple query that returns
 * the text form of each digit less than 5.
 *
 * Original LINQ query:
 * from n in numbers
 * where n < 5
 * select digits[n];
 */
public void linq13() {
    int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
    String[] digits = { "zero", "one", "two", "three", "four", "five", "six", "seven",
"eight", "nine" };
    List<String> lowNums = #{
        (numbers as n) where n < 5).
        (digits[n])
    };
    System.out.println("Numbers < 5:");
    for(String num : lowNums) {
        System.out.println(num);
    }
}

/**
 * This sample uses a compound from clause to make a query that returns all
pairs
 * of numbers from both arrays such that the number from numbersA is less than the
number
 * from numbersB.
 *
 * Original LINQ query:
 * from a in numbersA
 * from b in numbersB
 * where a < b
 * select new { a, b };
 */
public void linq14() {
    int[] numbersA = { 0, 2, 4, 5, 6, 8, 9 };
    int[] numbersB = { 1, 3, 5, 7, 8 };
    List<Struct> pairs = #{
        numbersA as a, numbersB as b
        where a < b
    };
    System.out.println("Pairs where a < b:");
    for(Struct pair : pairs) {
        System.out.printf("%s is less than %s\n", pair.get("a"), pair.get("b"));
    }
}
/**

```

```

    * This sample uses a compound from clause to select all orders where the
    * order total is less than 500.00.
    *
    * Original LINQ query:
    * from c in customers
    *     from o in c.Orders
    *     where o.Total < 500.00M
    *     select new {c.CustomerID, o.OrderID, o.Total};
    */
public void linq15() {
    List<Customer> customers = getCustomerList();
    List<Struct> orders = #{
        (customers join (orders where total < 500.00)).
        (customerID as customerID, orderID as orderID, total as total)
    };
    for(Struct order : orders) {
        System.out.println(order);
    }
}

/**
 * This sample uses a compound from clause to select all orders
 * where the order was made in 1998 or later.
 *
 * Original LINQ query:
 *
 * from c in customers
 *     * from o in c.Orders
 *     * where o.OrderDate >= new DateTime(1998, 1, 1)
 *     * select new { c.CustomerID, o.OrderID, o.OrderDate };
 */
public void linq16() {
    List<Customer> customers = getCustomerList();
    Calendar c = Calendar.getInstance();
    c.set(1998, Calendar.JANUARY, 1);
    Date d = c.getTime();
    List<Struct> orders = #{
        (customers join orders where orderDate > d).
        (customerID as customerID, orderID as orderID, orderDate as orderDate)
    };
    for(Struct order : orders) {
        System.out.println(order);
    }
}

/**
 * This sample uses a compound from clause to select all orders where the order total
 * is greater than 2000.00 and uses from assignment to avoid requesting the total
 * twice.
 *
 * Original LINQ query:
 * from c in customers
 *     from o in c.Orders
 *     where o.Total >= 2000.0M
 *     select new { c.CustomerID, o.OrderID, o.Total };
 */
public void linq17() {
    List<Customer> customers = getCustomerList();
    List<Struct> orders = #{
        (customers as c join c.orders as o where o.total > 2000).
        (c.customerID as customerID, o.orderID as orderID, o.total as total)
    };
    for(Struct order : orders) {
        System.out.println(order);
    }
}

/**
 * This sample uses multiple from clauses so that filtering on customers can be done
 * before selecting their orders. This makes the query more efficient by not selecting
 * and then discarding orders for customers outside of Washington.
 *
 * Original LINQ query:
 * from c in customers
 * where c.Region == "WA"
 * from o in c.Orders
 * where o.OrderDate >= cutoffDate

```

```

    * select new { c.CustomerID, o.OrderID };
    */
    public void linq18() {
        List<Customer> customers = getCustomerList();
        Calendar c = Calendar.getInstance();
        c.set(1997, 0, 1);
        Date cutoffDate = c.getTime();
        List<Struct> orders = #{
            (customers as c where c.region == "WA" join c.orders as o where o.orderDate
=> cutoffDate).
                (c.customerID as customerID, o.orderID as orderID)
        };
        for(Struct order : orders) {
            System.out.println(order);
        }
    }

    /**
     * This sample selects all orders,
     * while referring to customers by the order in which they are returned from the
query.
     *
     * Original LINQ query:
     * customers.SelectMany(
     *     (cust, custIndex) =>
     *     cust.Orders.Select(o => "Customer #" + (custIndex + 1) +
     *     " has an order with OrderID " + o.OrderID));
     */
    public void linq19() {
        List<Customer> customers = getCustomerList();
        List<String> customerOrders = #{
            (customers as c).
            ($index as custIndex, c.orders as o).
            ("Customer #" + (custIndex + 1) + " has an order with OrderID " + o.orderID)
        };
        for(String s : customerOrders) {
            System.out.println(s);
        }
    }

    /**
     * This sample uses an range operator to get only the first 3 elements of the array.
     *
     * Original LINQ query:
     * numbers.Take(3);
     */
    public void linq20() {
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
        List<Integer> first3Numbers = #{numbers[0..2]};
        for(Integer n : first3Numbers) {
            System.out.println(n);
        }
    }

    /**
     * This sample uses an range operator to get the first 3 orders from customers in
Washington.
     *
     * Original LINQ query:
     * from c in customers
     * from o in c.Orders
     * where c.Region == "WA"
     * select new { c.CustomerID, o.OrderID, o.OrderDate })
     * .Take(3);
     */
    public void linq21() {
        List<Customer> customers = getCustomerList();
        List<Struct> first3WAOrders = #{
            ((customers where region == "WA" join orders)[0..2]).
            (customerID as customerID, orderID as orderID, orderDate as orderDate)
        };
        System.out.println("First 3 orders in WA:");
        for(Struct order : first3WAOrders) {
            System.out.println(order);
        }
    }
}

```

```

/**
 * This sample uses an range operator to get all but the first 4 elements of the
array.
 *
 * Original LINQ query:
 * numbers.Skip(4);
 */
public void linq22() {
    int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
    List<Integer> allButFirst4Numbers = #{numbers[4..*]};
    System.out.println("All but first 4 numbers:");
    for(Integer n : allButFirst4Numbers) {
        System.out.println(n);
    }
}

/**
 * This sample uses an range operator to get all but the first 2 orders from customers
in Washington.
 *
 * Original LINQ query:
 * from c in customers
 * from o in c.Orders
 * where c.Region == "WA"
 * select new { c.CustomerID, o.OrderID, o.OrderDate };
 */
public void linq23() {
    List<Customer> customers = getCustomerList();
    List<Struct> waOrders = #{
        (customers as c where c.region == "WA" join c.orders as o)[2..*]).
        (c.customerID as customerID, o.orderID as orderID, o.orderDate as
orderDate)
    };
    System.out.println("All but first 2 orders in WA:");
    for(Struct order : waOrders) {
        System.out.println(order);
    }
}

// -----
// ----- Ordering Operators -----
// -----

/**
 * This sample uses orderby to sort a list of words alphabetically.
 *
 * Original LINQ query:
 * from w in words
 * orderby w
 * select w;
 */
public void linq28() {
    String[] words = { "cherry", "apple", "blueberry" };
    List<String> sortedWords = #{
        words as w order by w
    };
    System.out.println("The sorted list of words:");
    for(String w : sortedWords) {
        System.out.println(w);
    }
}

/**
 * This sample uses orderby to sort a list of words by length.
 *
 * Original LINQ query:
 * from w in words
 * orderby w.Length
 * select w;
 */
public void linq29() {
    String[] words = { "cherry", "apple", "blueberry" };
    List<String> sortedWords = #{ words as w order by w.length() };
    System.out.println("The sorted list of words (by length):");
    for(String w : sortedWords) {

```

```

        System.out.println(w);
    }
}

/**
 * This sample uses an order by operator to sort a list of products by name.
 *
 * Original LINQ query:
 * from p in products
 * orderby p.ProductName
 * select p;
 */
public void linq30() {
    List<Product> products = getProductList();
    List<Product> sortedProducts = #{
        products as p
        order by p.productName
    };
    for(Product p : sortedProducts) {
        System.out.println(p);
    }
}

/**
 * This sample uses an order by operator clause with a custom comparer to do
 * a case-insensitive sort of the words in an array.
 *
 * Original LINQ query:
 * words.OrderBy(a => a, new CaseInsensitiveComparer());
 */
public void linq31() {
    String[] words = { "aPPLE", "AbAcUs", "bRaNcH", "BlUeBeRrY", "ClOvEr", "cHeRry" };
    Comparator<String> comp = new Comparator<String>() {
        @Override
        public int compare(String o1, String o2) {
            return o1.toLowerCase().compareTo(o2.toLowerCase());
        }
    };
    List<String> sortedWords = #{
        words as w
        order by w using comp
    };
    for(String w : sortedWords) {
        System.out.println(w);
    }
}

/**
 * This sample uses an order by operator and descending to
 * sort a list of doubles from highest to lowest.
 *
 * Original LINQ query:
 * from d in doubles
 * orderby d descending
 * select d;
 */
public void linq32() {
    double[] doubles = { 1.7, 2.3, 1.9, 4.1, 2.9 };
    List<Double> sortedDoubles = #{
        doubles as d
        order by d desc
    };
    System.out.println("The doubles from highest to lowest:");
    for(Double d : sortedDoubles) {
        System.out.println(d);
    }
}

/**
 * This sample uses an order by operator to sort a list
 * of products by units in stock from highest to lowest.
 *
 * Original LINQ query:
 * from p in products
 *     orderby p.UnitsInStock descending
 *     select p;
 */

```

```

public void linq33() {
    List<Product> products = getProductList();
    List<Product> sortedProducts = #{
        products
        order by unitsInStock desc
    };
    for(Product p : sortedProducts) {
        System.out.println(p);
    }
}

/**
 * This sample uses an order by operator with a custom comparer to do a case-
insensitive
 * descending sort of the words in an array.
 *
 * Original LINQ query:
 * words.OrderByDescending(a => a, new CaseInsensitiveComparer());
 */
public void linq34() {
    String[] words = { "aPPLE", "AbAcUs", "bRaNch", "BlUeBeRrY", "ClOvEr", "cHeRry" };

    List<String> sortedWords = #{
        words
        order by toLowerCase() desc
    };
    for(String w : sortedWords) {
        System.out.println(w);
    }
}

/**
 * This sample uses an order by operator to sort a list of digits,
 * first by length of their name, and then alphabetically by the name itself.
 *
 * Original LINQ query:
 * from d in digits
 * orderby d.Length, d
 * select d;
 */
public void linq35() {
    String[] digits = { "zero", "one", "two", "three", "four", "five", "six", "seven",
"eight", "nine" };
    List<String> sortedDigits = #{
        digits as d
        order by d.length(); d
    };
    System.out.println("Sorted digits:");
    for(String d : sortedDigits) {
        System.out.println(d);
    }
}

/**
 * This sample uses an order by operator with a custom comparer to sort first
 * by word length and then by a case-insensitive sort of the words in an array.
 *
 * Original LINQ query:
 * words.OrderBy(a => a.Length)
 * .ThenBy(a => a, new CaseInsensitiveComparer());
 */
public void linq36() {
    String[] words = { "aPPLE", "AbAcUs", "bRaNch", "BlUeBeRrY", "ClOvEr", "cHeRry" };
    Comparator<String> comp = new Comparator<String>() {
        @Override
        public int compare(String o1, String o2) {
            return o1.toLowerCase().compareTo(o2.toLowerCase());
        }
    };
    List<String> sortedWords = #{
        words as w
        order by w.length(); w using comp
    };
    for(String w : sortedWords) {
        System.out.println(w);
    }
}

```

```

/**
 * This sample uses an order by operator to sort a list of products,
 * first by category, and then by unit price, from highest to lowest.
 *
 * Original LINQ query:
 * from p in products
 * orderby p.Category, p.UnitPrice descending
 * select p;
 */
public void linq37() {
    List<Product> products = getProductList();
    List<Product> sortedProducts = #{
        products
        order by category; unitPrice desc
    };
    for(Product p : sortedProducts) {
        System.out.println(p);
    }
}

/**
 * This sample uses an order by operator with a custom comparer to sort
 * first by word length and then by a case-insensitive descending sort of the words in
 * an array.
 *
 * Original LINQ query:
 * words.OrderBy(a => a.Length)
 * .ThenByDescending(a => a, new CaseInsensitiveComparer());
 */
public void linq38() {
    Comparator<String> comp = new Comparator<String>() {
        @Override
        public int compare(String o1, String o2) {
            return o1.toLowerCase().compareTo(o2.toLowerCase());
        }
    };
    String[] words = { "aPPLE", "AbAcUs", "bRaNch", "BlUeBeRrY", "ClOvEr", "cHeRry" };
    List<String> sortedWords = #{
        words as w
        order by w.length(); w using comp
    };
    for(String w : sortedWords) {
        System.out.println(w);
    }
}

// -----
// -----      Grouping Operators      -----
// -----

/**
 * This sample partition a list of numbers by their remainder when divided by 5.
 *
 * Original LINQ query:
 * from n in numbers
 * group n by n % 5 into g
 * select new { Remainder = g.Key, Numbers = g };
 */
public void linq40() {
    int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
    List<Struct> numberGroups = #{
        (unique((numbers as n).(n % 5)) as r).
        (r as remainder, (numbers as n where n % 5 == r) group as numbers)
    };
    System.out.println(numberGroups);

    for(Struct g : numberGroups) {
        System.out.printf("Numbers with a remainder of %s when divided by 5:\n",
g.getValue("remainder"));
        List<Integer> numbersInGroup = (List<Integer>) g.getValue("numbers");
        for(Integer n : numbersInGroup) {
            System.out.println(n);
        }
    }
}

```

```

}

/**
 * This sample partition a list of words by their first letter.
 *
 * Original LINQ query:
 * from w in words
 * group w by w[0] into g
 * select new { FirstLetter = g.Key, Words = g };
 */
public void linq41() {
    String[] words = { "blueberry", "chimpanzee", "abacus", "banana", "apple",
"cheese" };
    List<Struct> wordGroups = #{
        (unique(words.charAt(0)) as f).
        (f as firstLetter, (words as w where w.charAt(0) == f) group as words)
    };
    for(Struct g : wordGroups) {
        System.out.printf("Words that start with the letter '%s':\n",
g.getValue("firstLetter"));
        List<String> wordsInGroup = (List<String>) g.getValue("words");
        for(String w : wordsInGroup) {
            System.out.println(w);
        }
    }
}

/**
 * This sample partition a list of products by category.
 *
 * Original LINQ query:
 * from p in products
 * group p by p.Category into g
 * select new { Category = g.Key, Products = g };
 */
public void linq42() {
    List<Product> products = getProductList();
    List<Struct> orderGroups = #{
        (unique(products.category) as c).
        (c as category, (products where category == c) group as products)
    };
    for(Struct g : orderGroups) {
        System.out.printf("Products in category '%s':\n",
g.getValue("category"));
        List<Product> productsInGroup = (List<Product>)
g.getValue("products");
        for(Product p : productsInGroup) {
            System.out.println(p);
        }
    }
}

/**
 * This sample partition a list of each
 * customer's orders, first by year, and then by month.
 *
 * Original LINQ query:
 * from c in customers
 * select
 *     new
 *     {
 *         c.CompanyName,
 *         YearGroups =
 *             from o in c.Orders
 *             group o by o.OrderDate.Year into yg
 *             select
 *                 new
 *                 {
 *                     Year = yg.Key,
 *                     MonthGroups =
 *                         from o in yg
 *                         group o by o.OrderDate.Month into mg
 *                         select new { Month = mg.Key, Orders = mg }
 *                 }
 *     }
 */

```

```

*           }
*       };
*/
public void linq43() {
    List<Customer> customers = getCustomerList();

    List<Struct> customerOrderGroups = #{
        (customers as c).
        (c.companyName as companyName join
        (
            ( unique(c.orders.orderDate.year ) as year join
              (c.orders where orderDate.year == year) group as yearGroups
            ).
            (year as year,
              (
                  (unique(yearGroups.orderDate.month) as month) join
                  (yearGroups where orderDate.month == month) group as
orders
                      ) group as monthGroups
                ) group as yearGroups
            )
        );

    for(Struct companyGroup : customerOrderGroups) {
        String companyName = (String) companyGroup.getValue("companyName");
        System.out.println("Groups for company: "+companyName);
        List<Struct> yearGroups = (List<Struct>)
companyGroup.getValue("yearGroups");
        if(yearGroups == null) continue;
        for(Struct yearGroup : yearGroups) {
            Integer year = (Integer)yearGroup.getValue("year");
            System.out.println("    Groups for year: "+year);
            List<Struct> monthGroups = (List<Struct>)
yearGroup.getValue("monthGroups");
            if(monthGroups == null) continue;
            for(Struct monthGroup : monthGroups) {
                Integer month = (Integer)monthGroup.getValue("month");
                System.out.println("        Orders for month: "+month);
                List<Order> orders = (List<Order>)
monthGroup.getValue("orders");
                for(Order order : orders) {
                    System.out.println("            "+order);
                }
            }
        }
    }

    // -----
    // ----- Set Operators -----
    // -----

    /**
     * This sample uses unique operator to remove duplicate
     * elements in a sequence of factors of 300.
     *
     * Original LINQ query:
     * factorsOf300.Distinct();
     */
    public void linq46() {
        int[] factorsOf300 = { 2, 2, 3, 5, 5 };
        Collection<Integer> uniqueFactors = #{
            unique(factorsOf300)
        };
        System.out.println("Prime factors of 300:");
        for(Integer f : uniqueFactors) {
            System.out.println(f);
        }
    }

    /**
     * This sample uses unique operator to find the unique Category names.
     *
     * Original LINQ query:
     * from p in products
     * select p.Category)

```

```

        *         .Distinct();
    */
    public void linq47() {
        List<Product> products = getProductList();
        List<String> categoryNames = #{
            unique(products.category)
        };
        System.out.println("Category names:");
        for(String s : categoryNames) {
            System.out.println(s);
        }
    }

    /**
     * This sample uses union operator to create one sequence
     * that contains the unique values from both arrays.
     *
     * Original LINQ query:
     * numbersA.Union(numbersB);
     */
    public void linq48() {
        int[] numbersA = { 0, 2, 4, 5, 6, 8, 9 };
        int[] numbersB = { 1, 3, 5, 7, 8 };
        Collection<Integer> uniqueNumbers = #{
            unique(numbersA union numbersB)
        };
        System.out.println("Unique numbers from both arrays:");
        for(Integer n : uniqueNumbers) {
            System.out.println(n);
        }
    }

    /**
     * This sample uses union operator to create one sequence that
     * contains the unique first letter from both product and customer names.
     *
     * Original LINQ queries:
     * var productFirstChars =
     *     from p in products
     *     select p.ProductName[0];
     * var customerFirstChars =
     *     from c in customers
     *     select c.CompanyName[0];
     *
     * var uniqueFirstChars = productFirstChars.Union(customerFirstChars);
     */
    public void linq49() {
        List<Product> products = getProductList();
        List<Customer> customers = getCustomerList();
        Collection<Character> uniqueFirstChars = #{
            unique(
                products.productName.charAt(0)
                union
                customers.companyName.charAt(0)
            )
        };
        System.out.println("Unique first letters from Product names and Customer names:");
        for(Character ch : uniqueFirstChars) {
            System.out.println(ch);
        }
    }

    /**
     * This sample uses intersect operator to create one sequence that
     * contains the common values shared by both arrays.
     *
     * Original LINQ query:
     * numbersA.Intersect(numbersB);
     */
    public void linq50() {
        int[] numbersA = { 0, 2, 4, 5, 6, 8, 9 };
        int[] numbersB = { 1, 3, 5, 7, 8 };

        Collection<Integer> commonNumbers = #{
            numbersA intersect numbersB
        };
    }
}

```

```

    };
    System.out.println("Common numbers shared by both arrays:");
    for(Integer n : commonNumbers) {
        System.out.println(n);
    }
}

/**
 * This sample uses intersect operator to create one sequence that
 * contains the common first letter from both product and customer names.
 *
 * Original LINQ query:
 * var productFirstChars =
 *     from p in products
 *     select p.ProductName[0];
 * var customerFirstChars =
 *     from c in customers
 *     select c.CompanyName[0];
 * var commonFirstChars = productFirstChars.Intersect(customerFirstChars);
 */
public void linq51() {
    List<Product> products = getProductList();
    List<Customer> customers = getCustomerList();
    Collection<Character> commonFirstChars = #{
        unique(
            products.productName.charAt(0)
            intersect
            customers.companyName.charAt(0)
        )
    };
    System.out.println("Common first letters from Product names and Customer names:");
    for(Character ch : commonFirstChars) {
        System.out.println(ch);
    }
}

/**
 * This sample uses minus operator to create a sequence that
 * contains the values from numbersA that are not also in numbersB.
 *
 * Original LINQ query:
 * numbersA.Except(numbersB);
 */
public void linq52() {
    int[] numbersA = { 0, 2, 4, 5, 6, 8, 9 };
    int[] numbersB = { 1, 3, 5, 7, 8 };
    Collection<Integer> aOnlyNumbers = #{
        numbersA minus numbersB
    };
    System.out.println("Numbers in first array but not second array:");
    for(Integer n : aOnlyNumbers) {
        System.out.println(n);
    }
}

/**
 * This sample uses minus operator to create one sequence that contains
 * the first letters of product names that are not also
 * first letters of customer names.
 *
 * Original LINQ queries:
 * var productFirstChars =
 *     from p in products
 *     select p.ProductName[0];
 * var customerFirstChars =
 *     from c in customers
 *     select c.CompanyName[0];
 *
 * var productOnlyFirstChars = productFirstChars.Except(customerFirstChars);
 */
public void linq53() {
    List<Product> products = getProductList();
    List<Customer> customers = getCustomerList();
    Collection<Character> productOnlyFirstChars = #{
        unique(products.productName.charAt(0))
        minus

```

```

        unique(customers.companyName.charAt(0))
    };
    System.out.println("First letters from Product names, but not from Customer
names:");
    for(Character ch : productOnlyFirstChars) {
        System.out.println(ch);
    }
}

// -----
// ----- Conversion Operators -----
// -----

/**
 * This sample uses instanceof operator to return only
 * the elements of the array that are of type double.
 *
 * Original LINQ query:
 * numbers.Of<Type<double>();
 */
public void linq57() {
    Object[] numbers = { null, 1.0, "two", 3, "four", 5, "six", 7.0 };
    List<Object> doubles = #{
        numbers as n
        where n instanceof Double
    };
    System.out.println("Numbers stored as doubles:");
    System.out.println(doubles);
}

// -----
// ----- Element Operators -----
// -----

/**
 * This sample uses range operator to return the first matching element as a Product,
 * instead of as a sequence containing a Product.
 *
 * Original LINQ query:
 * (
 *     from p in products
 *     where p.ProductID == 12
 *     select p)
 *     .First();
 */
public void linq58() {
    List<Product> products = getProductList();
    Product product12 = #{
        (products where productID == 12)[0]
    };
    System.out.println(product12);
}

/**
 * This sample uses range operator to find the first element in the array that starts
 * with 'o'.
 *
 * Original LINQ query:
 * strings.First(s => s[0] == 'o');
 */
public void linq59() {
    String[] strings = { "zero", "one", "two", "three", "four", "five", "six",
"seven", "eight", "nine" };
    String startsWithO = #{
        (strings as s where s.charAt(0) == 'o')[0]
    };
    System.out.printf("A string starting with 'o': %s\n", startsWithO);
}

/**
 * This sample uses range operator to retrieve the second number greater than 5 from
 * an array.
 *
 * Original LINQ query:
 * (from n in numbers
 *  where n > 5

```

```

* select n
* .ElementAt(1)
*/
public void linq64() {
    int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
    int fourthLowNum = #{
        (numbers as n where n > 5)[1]
    };
    System.out.printf("Second number > 5: %s\n", fourthLowNum);
}

// -----
// ----- Quantifiers -----
// -----

/**
 * This sample uses any operator to determine if any of the words
 * in the array contain the substring 'ei'.
 *
 * Original LINQ query:
 * words.Any(w => w.Contains("ei"));
 */
public void linq67() {
    String[] words = { "believe", "relief", "receipt", "field" };
    Boolean iAfterE = #{
        any words contains("ei")
    };
    System.out.println("There is a word that contains in the list that contains 'ei':
"+iAfterE);
}

/**
 * This sample uses any operator to return a grouped a list of products only for
 * categories that have at least one product that is out of stock.
 *
 * Original LINQ query:
 * from p in products
 *   group p by p.Category into g
 *   where g.Any(p => p.UnitsInStock == 0)
 *   select new { Category = g.Key, Products = g };
 */
public void linq69() {
    List<Product> products = getProductList();
    List<Struct> productGroups = #{
        (unique(products.category) as cat).
        (cat as category, (products where category == cat) group as products)
        where any products unitsInStock == 0
    };
    for(Struct catGroup : productGroups) {
        String category = (String) catGroup.getValue("category");
        System.out.println("Products in category: "+category+" that have at least 1
product out of stock");
        List<Product> prGroup = (List<Product>) catGroup.getValue("products");
        for(Product p : prGroup) {
            System.out.println("    "+p);
        }
    }
}

/**
 * This sample uses all operator to determine whether an array contains only odd
 numbers.
 *
 * Original LINQ query:
 * numbers.All(n => n % 2 == 1);
 */
public void linq70() {
    int[] numbers = { 1, 11, 3, 19, 41, 65, 19 };
    Boolean onlyOdd = #{
        all (numbers as n) (n % 2 == 1)
    };
    System.out.println("The list contains only odd numbers: "+onlyOdd);
}

/**
 * This sample uses all operator to return a grouped a list of products only for

```

```

* categories that have all of their products in stock.
*
* Original LINQ query:
* from p in products
  group p by p.Category into g
  where g.All(p => p.UnitsInStock > 0)
  select new { Category = g.Key, Products = g };
*/
public void linq72() {
    List<Product> products = getProductList();
    List<Struct> productGroups = #{
        (unique(products.category) as cat).
        (cat as category, (products where category == cat) group as
products)
            where all products unitsInStock > 0
    };
    for(Struct catGroup : productGroups) {
        String category = (String) catGroup.getValue("category");
        System.out.println("Products in category '"+category+"' that have at all of
their product in stock: ");
        List<Product> prGroup = (List<Product>) catGroup.getValue("products");
        for(Product p : prGroup) {
            System.out.println("    "+p);
        }
    }
}

// -----
// ----- Aggregator Operators -----
// -----

/**
 * This sample uses count operator to get the number of unique factors of 300.
 *
 * Original LINQ query:
 * factorsOf300.Distinct().Count();
 */
public void linq73() {
    int[] factorsOf300 = { 2, 2, 3, 5, 5 };
    int uniqueFactors = #{
        count(unique(factorsOf300))
    };
    System.out.println("There are "+uniqueFactors+" unique factors of 300.");
}

/**
 * This sample uses count operator to get the number of odd ints in the array.
 *
 * Original LINQ query:
 * numbers.Count(n => n % 2 == 1);
 */
public void linq74() {
    int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
    int oddNumbers = #{
        count(numbers as n where n % 2 == 1)
    };
    System.out.println("There are "+oddNumbers+" odd numbers in the list.");
}

/**
 * This sample uses count operator to return a list
 * of customers and how many orders each has.
 *
 * Original LINQ query:
 * from c in customers
 * select new { c.CustomerID, OrderCount = c.Orders.Count() };
 */
public void linq76() {
    List<Customer> customers = getCustomerList();
    List<Struct> orderCounts = #{
        (customers as c).
        (c.customerID as customerID, count(c.orders) as orderCount)
    };
    for(Struct o : orderCounts) {
        String customerID = (String) o.getValue("customerID");
        Integer orderCount = (Integer) o.getValue("orderCount");
    }
}

```

```

        System.out.println("There are "+orderCount+" orders for customerID:
"+customerID);
    }
}

/**
 * This sample uses count operator to return a list
 * of categories and how many products each has.
 *
 * Original LINQ query:
 * from p in products
 * group p by p.Category into g
 * select new { Category = g.Key, ProductCount = g.Count() };
 */
public void linq77() {
    List<Product> products = getProductList();
    List<Struct> categoryCounts = #{
        (unique(products.category) as cat).
        (cat as category, count(products where category == cat) as
productCount)
    };
    for(Struct o : categoryCounts) {
        String category = (String) o.getValue("category");
        Integer productCount = (Integer) o.getValue("productCount");
        System.out.println("There are "+productCount+" products in category:
"+category);
    }
}

/**
 * This sample uses sum operator to get the total of the numbers in an array.
 *
 * Original LINQ query:
 * numbers.Sum();
 */
public void linq78() {
    int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
    int numSum = #{
        sum(numbers)
    };
    System.out.println("The sum of the numbers is "+numSum+".");
}

/**
 * This sample uses sum operator to get the total
 * number of characters of all words in the array.
 *
 * Original LINQ query:
 * words.Sum(w => w.Length);
 */
public void linq79() {
    String[] words = { "cherry", "apple", "blueberry" };
    int totalChars = #{
        sum(words.length())
    };
    System.out.println("There are a total of "+totalChars+" characters in these
words.");
}

/**
 * This sample uses sum operator to get the total units in stock for each product
 * category.
 *
 * Original LINQ query:
 * from p in products
 * group p by p.Category into g
 * select new { Category = g.Key, TotalUnitsInStock = g.Sum(p => p.UnitsInStock) };
 */
public void linq80() {
    List<Product> products = getProductList();
    List<Struct> categories = #{
        (unique(products.category) as cat).
        (cat as category, sum( (products where category == cat).unitsInStock ) as
totalUnitsInStock)
    };
    for(Struct o : categories) {
        String category = (String) o.get("category");

```

```

        Integer totalUnitsInStock = (Integer) o.get("totalUnitsInStock");
        System.out.println("There are "+totalUnitsInStock+" products in stock in
category: "+category);
    }
}

/**
 * This sample uses min operator to get the lowest number in an array.
 *
 * Original LINQ query:
 * numbers.Min();
 */
public void linq81() {
    int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
    int minNum = #{
        min(numbers)
    };
    System.out.println("The minimum number is "+minNum+".");
}

/**
 * This sample uses min operator to get the
 * length of the shortest word in an array.
 *
 * Original LINQ query:
 * words.Min(w => w.Length);
 */
public void linq82() {
    String[] words = { "cherry", "apple", "blueberry" };
    int shortestWord = #{
        min(words.length())
    };
    System.out.println("The shortest word is "+shortestWord+" characters long.");
}

/**
 * This sample uses min operator to get the cheapest
 * price among each category's products.
 *
 * Original LINQ query:
 * from p in products
 * group p by p.Category into g
 * select new { Category = g.Key, CheapestPrice = g.Min(p => p.UnitPrice) };
 */
public void linq83() {
    List<Product> products = getProductList();
    List<Struct> categories = #{
        (unique(products.category) as cat).
        (cat as category, min((products where category == cat).unitPrice) as
cheapestPrice)
    };
    for(Struct o : categories) {
        String category = (String) o.get("category");
        Double cheapestPrice = (Double) o.get("cheapestPrice");
        System.out.println("category="+category+"\tcheapestPrice="+cheapestPrice);
    }
}

/**
 * This sample uses min operator to get the products with the cheapest price in each
 * category.
 *
 * Original LINQ query:
 * from p in products
 * group p by p.Category into g
 * let minPrice = g.Min(p => p.UnitPrice)
 * select new { Category = g.Key, CheapestProducts = g.Where(p => p.UnitPrice ==
minPrice) };
 */
public void linq84() {
    List<Product> products = getProductList();
    List<Struct> categories = #{
        (unique(products.category) as cat).
        (cat as cat, (products where category == cat) group as pr).
        (cat as category, (pr where unitPrice == min(pr.unitPrice)) group as
cheapestProducts)
    };
}

```

```

        for (Struct o : categories) {
            String category = (String) o.get("category");
            List<Product> cheapestProducts = (List<Product>) o.get("cheapestProducts");

            System.out.println("category="+category+"\tcheapestProducts="+cheapestProducts);
        }
    }

    /**
     * This sample uses max operator to get the highest number in an array.
     *
     * Original LINQ query:
     * numbers.Max();
     */
    public void linq85() {
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
        int maxNum = #{
            max(numbers)
        };
        System.out.println("The maximum number is "+maxNum+".");
    }

    /**
     * This sample uses max operator to get the
     * length of the longest word in an array.
     *
     * Original LINQ query:
     * words.Max(w => w.Length);
     */
    public void linq86() {
        String[] words = { "cherry", "apple", "blueberry" };
        int longestLength = #{
            max(words.length())
        };
        System.out.println("The longest word is "+longestLength+" characters long.");
    }

    /**
     * This sample uses max operator to get the most expensive
     * price among each category's products.
     *
     * Original LINQ query:
     * from p in products
     * group p by p.Category into g
     * select new { Category = g.Key, MostExpensivePrice = g.Max(p => p.UnitPrice) };
     */
    public void linq87() {
        List<Product> products = getProductList();
        List<Struct> categories = #{
            (unique(products.category) as cat).
            (cat as category, max((products where category == cat).unitPrice) as
mostExpensivePrice)
        };
        for (Struct o : categories) {
            String category = (String) o.getValue("category");
            Double mostExpensivePrice = (Double) o.getValue("mostExpensivePrice");

            System.out.println("category="+category+"\tmostExpensivePrice="+mostExpensivePrice
);
        }
    }

    /**
     * This sample uses max operator to get the products with the most expensive price in
     each category.
     *
     * Original LINQ query:
     * from p in products
     * group p by p.Category into g
     * let maxPrice = g.Max(p => p.UnitPrice)
     * select new { Category = g.Key, MostExpensiveProducts = g.Where(p => p.UnitPrice ==
maxPrice) };
     */
    public void linq88() {
        List<Product> products = getProductList();
        List<Struct> categories = #{
            (unique(products.category) as cat).

```

```

        (cat as cat, (products where category == cat) group as pr).
        (cat as category, (pr where unitPrice == max(pr.unitPrice)) group as
mostExpensiveProducts)
    };
    for(Struct o : categories) {
        String category = (String) o.get("category");
        List<Product> mostExpensiveProducts = (List<Product>)
o.get("mostExpensiveProducts");

        System.out.println("category="+category+"\tmostExpensiveProducts="+mostExpensivePr
oducts);
    }
}

/**
 * This sample uses avg operator to get the average of all numbers in an array.
 *
 * Original LINQ query:
 * numbers.Average();
 */
public void linq89() {
    int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
    double averageNum = #{
        avg(numbers)
    };
    System.out.println("The average number is "+averageNum);
}

/**
 * This sample uses avg operator to get the average length of the words in the array.
 *
 * Original LINQ query:
 * words.Average(w => w.Length);
 */
public void linq90() {
    String[] words = { "cherry", "apple", "blueberry" };
    double averageLength = #{
        avg(words.length())
    };
    System.out.println("The average word length is "+averageLength+" characters.");
}

/**
 * This sample uses avg operator to get the average price of each category's products.
 *
 * Original LINQ query:
 * from p in products
 * group p by p.Category into g
 * select new { Category = g.Key, AveragePrice = g.Average(p => p.UnitPrice) };
 */
public void linq91() {
    List<Product> products = getProductList();
    List<Struct> categories = #{
        (unique(products.category) as cat).
        (cat as category, avg((products where category == cat).unitPrice) as
averagePrice)
    };
    for(Struct o : categories) {
        String category = (String) o.get("category");
        Double averagePrice = (Double) o.get("averagePrice");
        System.out.println("category="+category+"\taveragePrice="+averagePrice);
    }
}

// -----
// ----- Miscellaneous Operators -----
// -----

/**
 * This sample uses union operator to create one sequence that
 * contains each array's values, one after the other.
 *
 * Original LINQ query:
 * numbersA.Concat(numbersB);
 */
public void linq94() {
    int[] numbersA = { 0, 2, 4, 5, 6, 8, 9 };

```

```

    int[] numbersB = { 1, 3, 5, 7, 8 };
    List<Integer> allNumbers = #{
        numbersA union numbersB
    };
    System.out.println("All numbers from both arrays:");
    for(Integer n : allNumbers) {
        System.out.println(n);
    }
}

/**
 * This sample uses union operator to create one sequence that contains the
 * names of all customers and products, including any duplicates.
 *
 * Original LINQ queries:
 * var customerNames =
 *     from c in customers
 *     select c.CompanyName;
 * var productNames =
 *     from p in products
 *     select p.ProductName;
 * var allNames = customerNames.Concat(productNames);
 */
public void linq95() {
    List<Customer> customers = getCustomerList();
    List<Product> products = getProductList();
    List<String> allNames = #{
        customers.companyName
        union
        products.productName
    };
    System.out.println("Customer and product names:");
    for(String n : allNames) {
        System.out.println(n);
    }
}

/**
 * This sample check if two sequences match on all elements in the same order.
 *
 * Original LINQ query:
 * wordsA.SequenceEqual(wordsB);
 */
public void linq96() {
    String[] wordsA = new String[] { "cherry", "apple", "blueberry" };
    String[] wordsB = new String[] { "cherry", "apple", "blueberry" };

    Boolean match = #{
        all wordsA as a
        a == wordsB[$index]
    };
    System.out.println("The sequences match: "+match);
}

private LinqExampleData data = new LinqExampleData();

private List<Product> getProductList() {
    return data.getProductList();
}

private List<Customer> getCustomerList() {
    return data.getCustomerList();
}

```