



UNIwersytet w Białymstoku  
Instytut Informatyki

# FORMALNA SKŁADNIA GQL

Damian Wileński

Promotor pracy  
dr inż. Dominik Tomaszuk

Białystok 2023

# Spis treści

<b>Wstęp</b>	<b>4</b>
<b>1 Omówienie grafowych baz danych</b>	<b>9</b>
1.1 Wstęp do grafowych baz danych . . . . .	10
1.1.1 Podejście grafowe a podejście relacyjne . . . . .	11
1.1.2 Cechy grafowych baz danych . . . . .	13
1.2 Podstawowe pojęcia i terminologia związana z grafami . . . . .	14
1.2.1 Wierzchołki . . . . .	16
1.2.2 Krawędzie . . . . .	16
1.2.3 Atrybuty . . . . .	18
1.3 Struktura i model danych . . . . .	19
1.3.1 Hipergraf . . . . .	19
1.3.2 Graf właściwości . . . . .	21
1.3.3 Trójki RDF . . . . .	23
1.4 Języki zapytań i modyfikacji do grafowych baz danych . . . . .	25
1.4.1 Cypher . . . . .	26
1.4.2 PGQL . . . . .	29
1.4.3 G-CORE . . . . .	32
1.4.4 GQL . . . . .	34
1.4.5 Gremlin . . . . .	34
1.4.6 SPARQL . . . . .	35
1.4.7 GSQL . . . . .	39
1.5 Przegląd grafowych baz danych . . . . .	41
<b>2 Teoria języków formalnych i definiowanie gramatyk</b>	<b>45</b>
2.1 Typy języków formalnych . . . . .	46
2.1.1 Języki regularne . . . . .	46
2.1.2 Języki bezkontekstowe . . . . .	48
2.1.3 Języki kontekstowe . . . . .	50
2.1.4 Języki rekurencyjnie przeliczalne . . . . .	53
2.2 Proces analizy leksykalnej i składniowej . . . . .	55
2.2.1 Lekser . . . . .	56
2.2.2 Parser . . . . .	57
2.3 Przegląd narzędzi i notacji do definiowania gramatyk . . . . .	61
2.3.1 BNF . . . . .	61

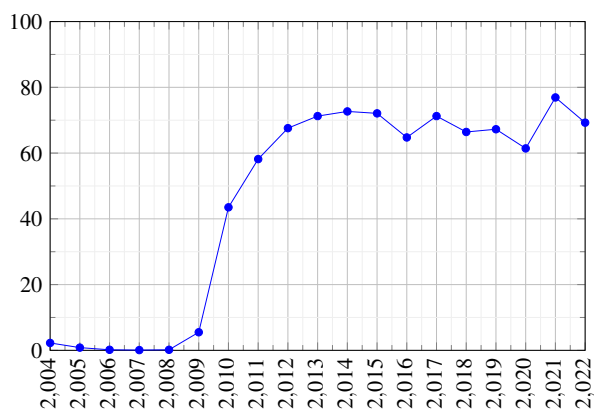
2.3.2	EBNF . . . . .	63
2.3.3	ABNF . . . . .	64
2.3.4	ANTLR . . . . .	64
2.3.5	Porównanie elementów syntaktycznych notacji . . . . .	67
<b>3</b>	<b>Implementacja</b>	<b>68</b>
3.1	Wykorzystane technologie . . . . .	68
3.2	Omówienie zaimplementowanej aplikacji . . . . .	90
3.3	Weryfikacja wydajności i testy . . . . .	99
	<b>Podsumowanie</b>	<b>102</b>
	<b>Bibliografia</b>	<b>103</b>

## Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum. Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

# Wstęp

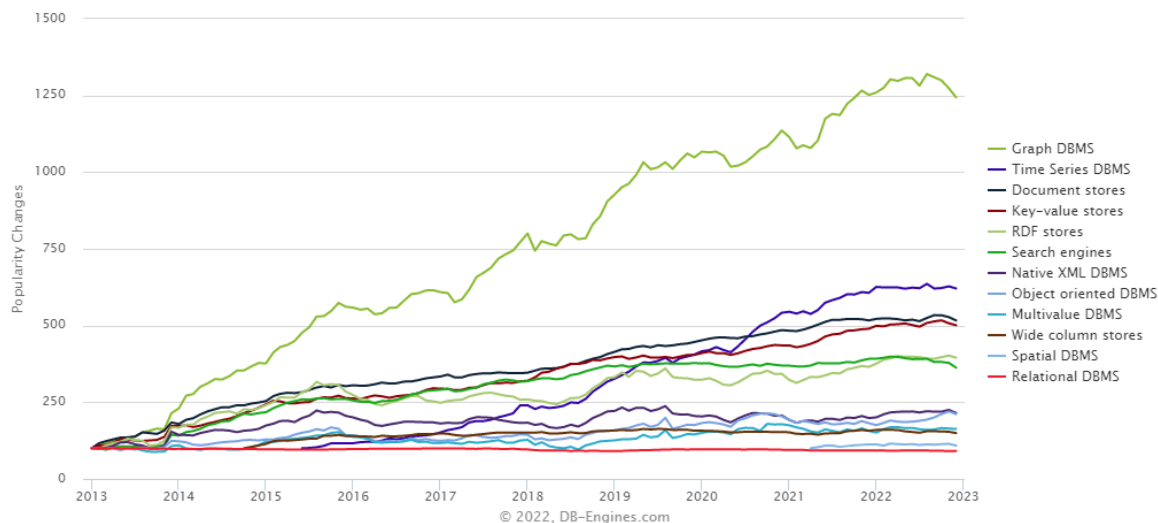
Na przełomie ostatnich lat można zauważyć znaczący wzrost zainteresowania bazami danych określanymi jako “NoSQL”. To zjawisko możemy zauważyć na przykład poprzez analizę wykresu ilości wyszukiwań frazy “nosql” w najpopularniejszej wyszukiwarce treści dostarczanej przez Google. Takie dane przedstawia poniżej załączony wykres [1].



Rysunek 1: Wykres wzrostu zainteresowania frazą "nosql"

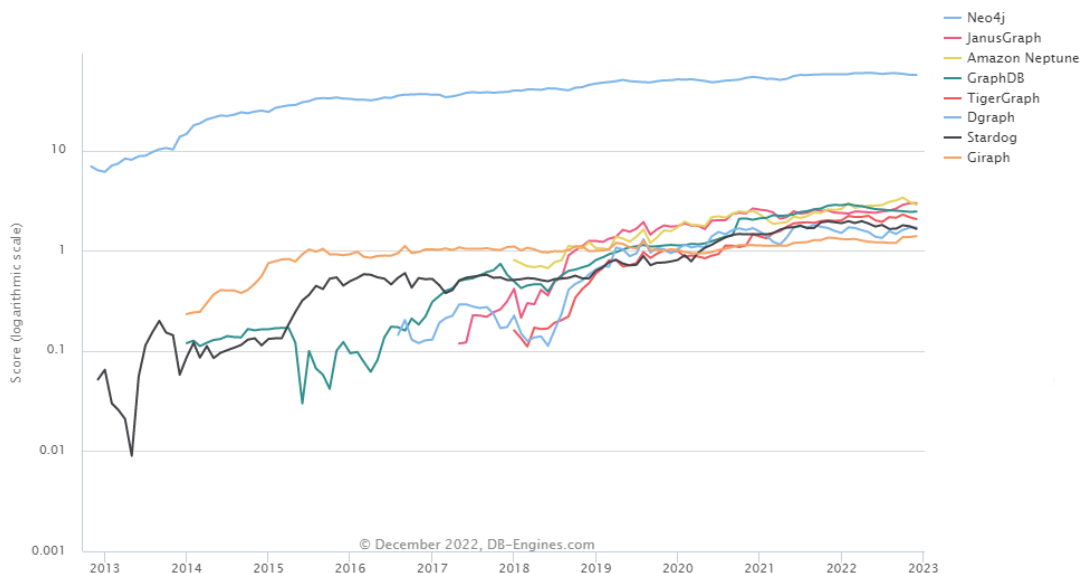
Wykres zawarty na rysunku 1 przedstawia średni procentowy wzrost ilości wyszukiwań danej frazy rok do roku. Możemy zauważyć, że od roku 2009 do roku 2013 mamy do czynienia ze skokowym wzrostem. Nierelacyjne bazy danych są wykorzystywane powszechnie w aplikacjach internetowych, które przetwarzają większy zasób danych niż aplikacje w oparciu tradycyjne relacyjne bazy danych. Możliwość do przetwarzania większej ilości danych wynika ze skalowalności oraz wysokiej dostępności baz tego typu. Dodatkową zaletą nierelacyjnego podejścia do baz danych jest między innymi łatwość adaptowania tego typu rozwiązania w aplikacjach przez programistów.

Sporą popularność nabierają także, tak zwane grafowe bazy danych, które należy kategoryzować jako bazy nierelacyjne. Grafowe bazy danych mają bardzo charakterystyczną strukturę, co sprawia, że ich zastosowanie ma dosyć określone, a zarazem ograniczone spektrum. Takiej bazy danych nie wykorzystamy na przykład do propagowania danych typowo tabelarycznych jak rekordy zawierające się w grach MMO. Będą to bardziej zastosowania określające charakterystyki danych aniżeli szczegółowe wypisanie ich w postaci na przykład tabel.



Rysunek 2: Wykres przedstawiający popularność poszczególnych typów systemów baz danych

Jak widać na załączonym rysunku 2, wzrost zainteresowania grafowymi bazami danych jest znaczący. Posiada on największy wzrost względem innych baz danych. Można więc założyć, że jest to jedna z najdynamiczniej rozwijających się technologii w dziedzinie baz danych. Grafowe bazy danych są doskonałym narzędziem do reprezentacji relacji między danymi i umożliwiają szybkie wyszukiwanie i agregację danych. Są również elastyczne i pozwalają na łatwą integrację z innymi systemami. Mogą być one jednak mniej skalowalne niż inne rodzaje baz danych i mogą wymagać większej ilości pamięci do przechowywania danych. Warto przyjrzeć się poziomem zainteresowania najpopularniejszych grafowych systemów baz danych. Rysunek 3 przybliży stosunek niniejszych systemów, dane są przedstawione w formie oceny na podstawie skali logarytmicznej wykorzystania tych systemów.



Rysunek 3: Wykres przedstawiający najpopularniejsze grafowe systemy baz danych

Analizując dane załączone w postaci wykresu, możemy stwierdzić, że najbardziej popularną bazą danych opartą o grafy jest Neo4j. Kolejno po Neo4j są JanusGraph, Amazon Neptune, GraphDB oraz TigerGraph. W kolejnych rozdziałach przybliżymy najważniejsze cechy wyróżniające podane systemy baz danych. Dla każdego z tych systemów zostaną przybliżone takie informacje jak architektura, model danych, język zapytań. Przybliżymy także informacje o badaniach wydajności oraz skalowalności danych systemów. Zostaną także przedstawione sposoby licencjonowania danych systemów.

Dużą uwagę zwrócimy także na języki zapytań które są wykorzystywane w tego typu systemach. Każda grafowa baza danych posiada własny język zapytań oraz ewentualnie posiada obsługę innych języków przez rozszerzenia. Zostaną przedstawione języki takie jak Cypher (dla Neo4j), PGQL (dla Oracle Graph), GSQL (dla TigerGraph) lub Gremlin (dla Apache TinkerPop). Dodatkowo warto wspomnieć o języku zapytań G-CORE, został on przedstawiony jako koncepcja potencjalnego standard przez Linked Data Benchmark Council. Oprócz języków zapytań, oprócz wymienionych już języków zostanie przedstawiony też SPARQL (dla grafów RDF) i AQL (dla baz danych ArangoDB) oraz GQL. Warto zaznaczyć, że w przypadku grafowych baz danych niektóre z języków zapytań są trudniejsze w nauce niż w pozostałych typach baz danych.

Każdy z języków zapytań, które wymieniono, ma pewne ograniczenia. Mają również różne sposoby definiowania zapytań, które odczytują dane lub je modyfikują, oraz różną strukturę grafową. Wynika to często z różnic między innymi w sposobie reprezentacji danych w grafie oraz wykorzystanych modeli.

Rozwiązaniem tego problemu ma być standaryzacja grafowych systemów zarządzania bazami danych. Takim rozwiązaniem ma być GQL (Graph Query Language). Idea GQL została opisana w dokumencie The GQL Manifesto [green2019gql].

The GQL Manifesto to dokument przedstawiający wizję komponowalnego, deklaratywnego języka zapytań grafowych opartego na dopasowywaniu wzorca i operacjach grafowych. Dokument ten podkreśla potrzebę standaryzacji języków zapytań grafowych, aby umożliwić

ich łatwiejsze użytkowanie i integrację z innymi systemami. GQL Manifesto zwraca również uwagę na fakt, że wiele obecnie dostępnych języków zapytań grafowych jest ograniczone w swoich możliwościach i ma różne sposoby definiowania zapytań czytających i modyfikujących dane, co utrudnia ich użytkowanie. Dokument ten sugeruje, że GQL (Graph Query Language) może być rozwiązaniem tych problemów poprzez zapewnienie kompozycyjności i deklaratywności, a także skupienie się na dopasowywaniu wzorca i operacjach grafowych. GQL Manifesto ma na celu zainspirować rozwój standaryzowanego języka zapytań grafowych, który będzie użyteczny dla programistów i użytkowników grafowych baz danych.

Standard ten ma skupiać w sobie szereg funkcjonalności, które umożliwią efektywne zarządzanie i analizę danych zapisanych w grafowych bazach danych. Wśród tych funkcjonalności warto wymienić:

- Szybkie i łatwe wyszukiwanie danych za pomocą dopasowywania wzorca grafów,
- Wykonywanie różnych operacji na grafach, takich jak tworzenie, modyfikacja i usuwanie wierzchołków i krawędzi oraz wyznaczanie ścieżek i łączy między wierzchołkami,
- Integrację z innymi systemami, takimi jak relacyjne bazy danych lub zewnętrzne narzędzia do analizy danych, co pozwoli na wykorzystanie zalet obu typów technologii.
- Tworzenie złożonych zapytań za pomocą składni deklaratywnej, co pozwoli na łatwe przetwarzanie i analizę dużych ilości danych.
- obsługę transakcji, co pozwoli na zapewnienie integralności i bezpieczeństwa danych podczas ich modyfikacji.
- Wysoką wydajnością podczas wykonywania zapytań.
- Powinien być skalowalny, co pozwoli na sprawne działanie nawet przy dużych ilościach danych i wysokiej liczbie zapytań.
- Powinien charakteryzować się prostą składnią, co umożliwi łatwe użytkowanie przez programistów i użytkowników.
- Powinien być dobrze udokumentowany, co ułatwi jego naukę i użytkowanie.

Kluczowym aspektem rozwoju każdego języka programowania jest jego składnia i sposób walidacji jej poprawności. Pozwala to na dynamiczniejsze wprowadzanie zmian oraz dostarczanie możliwości zapoznania się z technologią dla osób wchodzących w jej arkana.

Standard GQL jest rozwojową technologią, która ma pełnić funkcję duchowego odpowiednika SQL. Sprawia to, że w najbliższej przyszłości może stanowić on bardzo ważny filar w rozwoju grafowych baz danych. Ważne jest, aby technologia o takim potencjale posiadała odpowiednie narzędzie do walidacji składni.

Celem niniejszej pracy jest opracowanie parsera oraz zdefiniowanie formalnej składni języka GQL z wykorzystaniem narzędzia ANTLR (Another Tool for Language Recognition). Parser ten ma na celu umożliwienie skutecznego przetwarzania zapytań w języku GQL przez maszyny i udostępnienie interfejsu do ich obsługi.



Ponadto praca ta zakłada stworzenie edytora online, który umożliwi wykorzystanie parsera GQL do tworzenia i testowania zapytań w języku GQL. Edytor ten ma być łatwy w użytku oraz ma umożliwiać łatwe tworzenie zapytań za pomocą interaktywnego interfejsu.

Aby zrozumieć zagadnienie GQL i przygotować parser oraz edytor online, konieczne jest zapoznanie się z podstawową wiedzą na temat grafowych baz danych, języków zapytań grafowych oraz metod ich przetwarzania. W pracy zostanie przedstawiona ta wiedza w odpowiednim stopniu szczegółowości.

W drugim rozdziale pracy zostanie omówiona teoria związana z grafami wykorzystywanymi w bazach danych. W szczegółach omówimy model danych, jakim jest Property Graph, o którego podstawy opiera się GQL. Zostanie także omówiony model RDF, który jest drugim pod względem popularności modelem po Property Graphie. Zostanie omówiona definicja grafowych baz danych, ich zastosowania, wady oraz zalety. Przedstawimy elementy grafów, które są powszechnie stosowane w grafowych bazach. Sekcja ta będzie także zawierała opis wyróżnionych systemów baz danych oraz systemów zarządzania bazą danych.

Rozdział trzeci będzie zawierał szczegółowe informacje dotyczące analizy składniowej i przetwarzania języków. Są to ważne zagadnienia pod względem implementowania parsera. Przybliżymy proces przebiegu tłumaczenia oraz podstawowe elementy jak lekser, parser, abstrakcyjne drzewo składni oraz inne wyszczególnione elementy. Omówimy także dostępne na rynku rozwiązania dotyczące definiowania rozpoznawania języka. W głównej mierze skupimy się jednak na ANTLR-ze wykorzystanego do implementacji parsera.

Przedostatni rozdział będzie poświęcony implementacji aplikacji oraz gramatyki wraz z najważniejszymi zagadnieniami dotyczącymi wykorzystanej technologii. Przedstawiony zostanie projekt gramatyki zaimplementowany w ANTLR. Zostanie także przedstawiona implementacja parsera, który będzie oparty o wygenerowany kod z ANTLR. Zostanie wskazany także edytor kodu online, w którym będzie wykorzystywany parser w celu weryfikacji skłaniania wprowadzanych zapytań GQL. Rozdział ten będzie także opisywał zaimplementowane testy w wybranych framework'ach.

# Rozdział 1

## Omówienie grafowych baz danych

Rozwój grafowych baz danych jest nierozdzielnie związany z postępem w dziedzinie teorii grafów i technologii informacyjnej. Początki teorii grafów sięgają XVIII wieku, kiedy to szwajcarski matematyk Leonhard Euler sformułował problem mostów królewieckich, uważany za pierwszy problem w teorii grafów. W tym kontekście grafy były początkowo narzędziem matematycznym służącym do modelowania złożonych problemów.

Przez wiele lat, główne zastosowania teorii grafów znajdowały się w matematyce i naukach ścisłych. Dopiero w połowie XX wieku, z rozwojem komputerów, grafy zaczęły być stosowane w informatyce. Na przykład, w latach 60., grafy były używane do modelowania struktur danych i algorytmów, takich jak drzewa przeszukiwań binarnych i algorytmy przeszukiwania grafów.

W latach 70 i 80 poprzedniego wieku, kiedy dominującym modelem bazy danych stały się relacyjne bazy danych, grafy stosowano głównie jako narzędzie analityczne. Chociaż model relacyjny był efektywny dla wielu zastosowań, okazało się, że ma trudności z efektywnym modelowaniem i przeszukiwaniem skomplikowanych struktur danych, takich jak sieci.

Przełom nadszedł na początku XXI wieku z pojawieniem się grafowych baz danych. Systemy takie jak Neo4j (pierwsza wersja z 2007 roku), zostały zaprojektowane specjalnie do obsługi danych w formie grafu. Od tego czasu, grafowe bazy danych szybko zyskały na popularności, dzięki ich zdolności do efektywnego modelowania złożonych relacji i łatwego dostępu do danych. Z biegiem czasu powstała duża liczba nowych systemów a samo pojęcie grafowych baz danych nabrało bardzo mocno na znaczeniu.

Grafowe bazy danych są coraz bardziej popularne w różnorodnych dziedzinach, takich jak sieci społeczne, bioinformatyka, rekomendacje, analiza zależności w danych i wiele innych. Ich wykorzystanie umożliwia efektywne modelowanie oraz odwzorowywanie relacji i powiązań między danymi, które są trudne do uchwycenia w tradycyjnych bazach danych opartych na podejściu relacyjnym.

Pierwsza sekcja tego rozdziału będzie skupiać się na omówieniu podstawowych założeń baz danych opartych o grafy. Skupimy się na przedstawieniu najważniejszych aspektów modelu grafowego. Porównamy także model grafowy z modelem relacyjnym, w celu ukazania w jak dużym stopniu grafowe bazy danych są bardziej zaawansowane w kwestii relacji. Zostaną także przedstawione najważniejsze cechy oraz zalety tego podejścia do modelowania danych.

W drugiej sekcji tego rozdziału przybliżymy pojęcie grafów oraz ich matematyczną re-

prezentację. Grafy są strukturami składającymi się z wierzchołków (węzłów) i krawędzi (relacji) między tymi wierzchołkami. Przedstawienie tej definicji może okazać się bardzo ważne w zrozumieniu grafowych baz danych. Jest to podstawowa wiedza stanowiąca o tym modelu reprezentacji danych. Definicje zostaną przedstawione na podstawie interpretacji matematycznej. Jest ono intuicyjne oraz pokazuje jasno reguły operowania na grafach.

Następna sekcja przybliży nam modele danych stosowane w grafowych bazach danych. Współczesne systemy bazodanowe oferują różne modele. Sekcja ta będzie zawierała najbardziej popularne z nich. Skupimy się także na określeniu różnic między modelami, a także na ich wadach oraz zaletach. Kulminacyjnym elementem tej sekcji będzie porównanie ze sobą wszystkich omówionych modeli.

W kolejnej sekcji przedstawione zostaną systemy zarządzania grafowymi bazami danych (GDBM). Tego typu systemy to specjalistyczne oprogramowanie do zarządzania danymi i strukturą grafowych baz danych. Przedstawione zostaną najpopularniejsze systemy GDBM, takie jak “Neo4j”, “Amazon Neptune”, “JanusGraph” i inne. Opisane zostaną ich funkcje i możliwości oraz przyjrzymy się także ich składni zapytań na podstawie przykładów.

Ostatnia sekcja tego rozdziału skupia się na językach zapytań i modyfikacji danych w grafowych bazach danych. Przedstawione zostaną języki zapytań wykorzystywane przez GDBM omówione w sekcji poprzedniej. Zostaną omówione takie języki zapytań jak Cypher, Grem-lin, SPARQL oraz inne. Zaprezentowane zostaną podstawowe operacje zapytań, takie jak wyszukiwanie węzłów, pobieranie relacji, analiza grafów, a także modyfikowanie i aktualizowanie danych w bazach grafowych.

## 1.1 Wstęp do grafowych baz danych

Dzisiaj, wraz z rosnącym zainteresowaniem technologiami takimi jak Big Data i sztuczna inteligencja, grafowe bazy danych są coraz bardziej uznawane za kluczowe narzędzie do analizy i manipulacji skomplikowanymi danymi, które są niewygodne lub niemożliwe do obsługi w tradycyjnych relacyjnych bazach danych.

Dane w modelu grafowym, opisujemy dwoma elementami abstrakcyjnymi, są nimi wierzchołki oraz krawędzie. Wierzchołki reprezentują encje (takie jak osoby, produkty, konta), a krawędzie reprezentują relacje między tymi encjami (takie jak znajomość, własność, członkostwo). W zależności od rodzaju modelu jesteśmy w stanie zdefiniować również takie elementy jak atrybuty i właściwości czy też możemy interpretować w inny sposób definicję krawędzi. Grafy z założenia są bardzo modalne, to znaczy, że jesteśmy w stanie stworzyć wiele podejść, które umożliwiają lepsze dostosowanie modelu danych.

Grafowe bazy danych są typem bazy danych, które wykorzystują struktury grafu do przedstawiania i przechowywania danych. Jak większość systemów baz danych posiadają one wbudowane metody Create, Read, Update, Delete (CRUD). Implementacja ich jednak różni się znacząco od powszechnie znanych baz relacyjnych. Analizując tego rodzaju bazy danych, musimy zwrócić szczególną uwagę na dwa elementy, są nimi bazowe magazynowanie danych oraz silnik przetwarzający. Elementy te stanowią jądro całego systemu bazy danych.

Rozważając pierwszy element, jakim jest bazowe magazynowanie danych, należy rozumieć ten element jako sposób zapisu danych. Kwestia zapisu danych w formie grafu zdawa-

łaby się intuicyjna, jako iż mówimy o systemach ściśle opartych o natywny model, w którym wykorzystujemy wierzchołki i krawędzie. Jednak nie zawsze dane te są przechowywane w takiej formie. Część systemów owszem opiera się o natywne podejście do grafów, są jednak także rozwiązania oparte o inne modele danych. Otóż dane nie necessarily muszą być przechowywane w takiej formie, jaką ją nazywamy. Niektóre z systemów stworzyły specjalny mechanizm serializacji, który umożliwia wykorzystanie sprawdzonych rozwiązań występujących już na rynku. Innymi słowy, istnieją rozwiązania, które zamiast natywnego podejścia stosują serializację na przykład model relacyjny. Umożliwia to wykorzystanie takich mechanizmów jak indeksowanie i inne tego typu narzędzia, które w bazach typu relacyjnego osiągnęły bardzo wysoki stopień optymalizacji. W obu podejściach, zarówno natywnym, jak i opartym o serializację systemy te starają się tworzyć optymalne rozwiązania w kwestii modyfikacji, wybierania i przetwarzania danych.

Drugim elementem jest silnik przetwarzający, który odpowiada, w jaki sposób dane w omawianym typie baz danych są indeksowane. W przypadku baz danych grafowych istnieje pewna reguła, która wymaga, aby baza danych korzystała z indeksu bezpośredniego sąsiedztwa. Oznacza to, że połączone węzły fizycznie wskazują na siebie nawzajem. Jednakże istnieje inna reguła opisująca grafową bazę danych, zgodnie z którą każda baza danych, która udostępnia użytkownikowi model danych grafowych poprzez operacje CRUD, może być uznana za bazę danych grafową. Ważnym aspektem rozważanego podejścia jest jednak uwagę na znaczną poprawę wydajności, jaką zapewnia indeks bezpośredniego sąsiedztwa, i dlatego termin “przetwarzanie grafów natywnych” jest używany do opisanie baz danych grafowych, które wykorzystują ten rodzaj indeksu.

Możemy słusznie wywnioskować, że podejście do baz w natywny sposób jest bliższe definicji grafowej bazy danych, mija się to jednak z prawdą. Oba podejścia stanowią tylko formę, w jaki sposób dane są przechowywane i przetwarzane. Nie niesie to za sobą żadnych większych konsekwencji. Bazy tego typu w obu podejściach są dalej grafowymi bazami danych. Natywne przechowywanie grafów ma swoje zalety, ponieważ jest specjalnie zaprojektowane pod kątem wydajności i skalowalności. Z kolei przechowywanie grafów w sposób nienatywny korzysta z dojrzałych rozwiązań rynkowych, których właściwości i możliwości są doskonale znane. Natywne przetwarzanie grafów przyspiesza przeglądanie grafu, ale może utrudniać niektóre zapytania, które nie polegają na przeglądaniu grafu lub wymagają dużej ilości pamięci.

Systemy baz danych oparte o grafy są ściśle przystosowane do pracy z narzędziami wspierającymi transakcje. Można śmiało założyć, że tego rodzaju bazy danych są wprost przystosowane do obsługi przez systemy OLTP (Online Transaction Processing). Systemy OLTP możemy zdefiniować jako systemy, które jak sama nazwa wskazuje, przetwarzają dużą ilość danych na podstawie szeregu transakcji wykonujących się jedna po drugiej. Transakcje mają to do siebie, że jeżeli pojawi się jakakolwiek nieprawidłowość, zatrzymują operacje wykonywane w danej transakcji lub grupie transakcji. Tego typu mechanizmy zapewniają integralność danych oraz zabezpieczenie przed nieupoważnionym odczytem danych zmienianych.

### **1.1.1 Podejście grafowe a podejście relacyjne**

Zgodnie z definicją (definicja 1.1), relacyjne bazy danych, organizują dane w strukturach tabelarycznych. Każda tabela, znana również jako relacja, składa się z kolumn (atrybutów)

i wierszy (rekordów). Relacje między tabelami są ustalane poprzez klucze obce, czyli wartości, które odwołują się do kluczy głównych rekordów z innych tabel. Przede wszystkim, jeżeli mamy do czynienia z prostą strukturą danych, która nie koncentruje się na relacjach między danymi, podejście relacyjne jest optymalne. Ważnym aspektem jest także zmienność struktury danych, relacyjne bazy danych są mniej skalowalne. Wynika to z faktu, że dokonanie zmiany na istniejącej tabeli może być czasochłonne, a przede wszystkim może zakłócić jednorodność danych w niej zawartych.

W przypadku relacyjnych baz danych mamy także do czynienia z takimi regułami jak reguły normalizacji. Stanowią one podstawę modelowania tabel, ponieważ stanowią konsensus związany z dobrymi praktykami oraz zapewnieniem integralności i nieutrącalności danych. Powszechnie wiadomo, że model relacyjny zdominował znacząco rynek baz danych ze względu na efektywność implementacji oraz optymalność.

### **Definicja 1.1. Relacyjna baza danych**

*Relacyjna baza danych to zbiór powiązanych ze sobą tabel, które przechowują dane w formie relacji. Każda tabela w relacyjnej bazie danych składa się z wierszy i kolumn. Wiersze odpowiadają rekordom lub obiektom danych, podczas gdy kolumny reprezentują atrybuty lub cechy tych rekordów. Formalnie, relacyjna baza danych jest definiowana jako:*

$$R = \{R_1, R_2, \dots, R_n\},$$

gdzie:

- $R$  jest zbiorem tabel w bazie danych,
- $R_i$  reprezentuje pojedynczą tabelę o określonej strukturze i nazwie.

Z drugiej strony, grafowe bazy danych opierają się na teorii grafów i reprezentują dane jako węzły (encje) i krawędzie (relacje). W przeciwieństwie do modelu relacyjnego model grafowy pozwala na bezpośrednie połączenie dowolnych dwóch węzłów za pomocą krawędzi, co oznacza, że relacje są obywatelami pierwszej klasy w modelu grafowym. Warto także zaznaczyć, że istnieje kilka rodzajów modeli grafowych. Każdy z tych modeli znajduje zastosowanie w różnych systemach. Najczęściej spotykanym modelem grafowym jest graf właściwości. Jest to znacząca różnica w między tymi dwoma podejściami, jako iż model relacyjny ma jeden określony model.

W przypadku złożonych, wieloaspektowych relacji, model grafowy jest często bardziej intuicyjny i elastyczny. Grafowe bazy danych pozwalają na proste i efektywne przeszukiwanie złożonych ścieżek i wzorców relacji, co jest trudne do osiągnięcia w modelu relacyjnym bez skomplikowanych zapytań i kosztownych operacji, takich jak złączenia. Słowo kluczowe "relacja" jest bardzo ważna, przeważnie modelując dane w podejściu grafowym, skupiamy się na połączeniach między tymi danymi. Tego rodzaju podejście przede wszystkim jest bardziej intuicyjne. Łatwiej jest powiązać dane, które są reprezentowane węzłem niż złożone struktury tabelaryczne.

Ważnym aspektem w przypadku grafowych baz danych jest możliwość rekurencyjnego operowania po grafie. Jest to element, który jest trudny do zaimplementowania w relacyjnym podejściu. Przechodząc przez graf na podstawie przykładu grafu skierowanego czy drzewa,

zjawisko rekurencji jest osiągalne w bardzo łatwy sposób. Algorytm porusza się po wierzchołkach do momentu osiągnięcia warunku kończącego. Przechodzenie po wierzchołkach odbywa się za pośrednictwem odwoływania się do kolejnych wierzchołków połączonych relacjami. Przypadek modelu relacyjnego wymagałby wykorzystania dodatkowych tabel, które znacznie zwiększyłyby złożoność zapytań, a także samej struktury danych.

Jednak model relacyjny ma swoje zalety w przypadku prostych, dobrze zdefiniowanych struktur danych z ustalonymi schematami i niskim poziomem połączeń. Relacyjne bazy danych są również dobrze wspierane przez wiele narzędzi i technologii, a także posiadają sprawdzone mechanizmy zarządzania transakcjami i zgodnością z ACID (atomicity, consistency, isolation, durability). W praktyce wybór między grafowymi a relacyjnymi bazami danych zależy od specyfiki problemu, który ma zostać rozwiązany, oraz od charakterystyki danych, które mają być przechowywane i przetwarzane.

### **1.1.2 Cechy grafowych baz danych**

Jedną z najważniejszych cech grafowych baz danych jest ich zdolność do naturalnego i intuicyjnego modelowania złożonych struktur i relacji. Grafowe bazy danych traktują relacje między danymi jako równie ważne co same dane, co pozwala na łatwe i efektywne modelowanie skomplikowanych struktur danych. Ważną zaletą tego typu podejścia do strukturyzowania danych jest łatwość ich ilustrowania. Poprzez przedstawienie grafu możemy wprost wywnioskować relacje między określonymi wierzchołkami.

Bardzo ważną zaletą grafowych baz danych oprócz łatwości modelowania jest wydajność. Wydajność zapytań jest bardzo wysoka nawet przy bardzo skomplikowanych zapytaniach. Dzięki temu, że relacje są przechowywane bezpośrednio w strukturze danych, operacje na wielostopniowych relacjach, takie jak znalezienie najkrótszej ścieżki między dwoma węzłami czy przeszukiwanie sieci, mogą być wykonane znacznie szybciej niż w tradycyjnych bazach relacyjnych. Aspektem, na który również należy zwrócić uwagę w kwestii wydajności, jest łatwość znajdowania wzorców w danych. Można osiągnąć to w bardzo łatwy sposób, analizując określone relacje między węzłami.

Grafowe bazy danych oferują również wysoką elastyczność modelu danych. Oznacza to, że model tego typu bazy danych nie zakłada odgórnych schematów. Możemy dowolnie dodawać nowe dane bez konieczności wpisywania się w określone standardy. Jedyną rzeczą, na jakiej musimy się skupiać, jest stworzenie odpowiednich relacji między istniejącymi danymi a danymi, które już znajdują się w bazie.

Interesującym aspektem jest też rekurencyjna natura reprezentowanych danych. Jest to temat bardzo interesujący w przypadku analizy danych. W grafowych bazach danych w łatwy sposób możemy reprezentować takie struktury jak drzewa, hierarchie oraz możemy wykorzystywać na przykład model grafów skierowanych. Daje to bardzo dużo możliwości jak ustalenie, że dana informacja odnosi się do ustalonego źródła.

Należy także pamiętać, że na rynku znajduje się wiele marek rozwijających tę formę systemów bazodanowych. Wynika z tego fakt, że istnieje wiele rodzajów grafowych baz danych, które mogą dostarczać różnorodne narzędzia lub rozwiązania wpasowujące się w nasz problem.

## 1.2 Podstawowe pojęcia i terminologia związana z grafami

Graf jest abstrakcyjną strukturą matematyczną składającą się z dwóch zbiorów: zbioru wierzchołków (vertex set) i zbioru krawędzi (edge set). Na podstawie tych dwóch zbiorów jesteśmy w stanie stworzyć model, który opisuje dane i tworzy między nimi relacje. Dodatkowo struktura graf może również zawierać dodatkowe informacje, takie jak wagi krawędzi lub etykiety wierzchołków, jednak podstawowym elementem definicji jest para zbiorów wierzchołków i krawędzi.

### Definicja 1.2. Graf

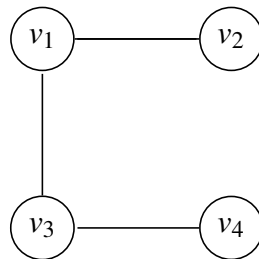
Niech  $G = (V, E)$  będzie grafem skierowanym lub nieskierowanym, gdzie  $V$  oznacza zbiór wierzchołków, a  $E$  oznacza zbiór krawędzi. Krawędź  $e$  jest reprezentowana jako para wierzchołków  $(u, v)$ , gdzie  $u, v \in V$ . Graf może być skierowany, gdy krawędzie posiadają kierunek, lub nieskierowany, gdy krawędzie są bez kierunku. Formalnie, graf można zdefiniować jako parę zbiorów:

$$G = (V, E),$$

gdzie:

- $V$  jest zbiorem wierzchołków,  $V = \{v_1, v_2, \dots, v_n\}$ ,
- $E$  jest zbiorem krawędzi,  $E \subseteq V \times V$ .

W zależności od charakterystyki zbioru krawędzi grafy mogą być klasyfikowane na różne sposoby. Przykładowo, graf nieskierowany to taki, którego krawędzie nie mają określonego kierunku. Na przedstawionym rysunku (Rysunek 1.1) widnieje sposób reprezentacji grafów nieskierowanych. Zgodnie z definicją (Definicja 1.3), graf nieskierowany to taki graf, którego relacje nie zakładają nadrzędności wierzchołków. Oznacza to tyle, że wierzchołki w tym grafie mają taką samą rangę. Relacje nie określają, w którym kierunku graf powinien być czytany lub możemy powiedzieć, że relacje w takim grafie są wielokierunkowe.



Rysunek 1.1: Przykład grafu nieskierowanego

### Definicja 1.3. Graf nieskierowany

Niech  $G = (V, E)$  będzie grafem nieskierowanym, gdzie  $V$  oznacza zbiór wierzchołków, a  $E$  oznacza zbiór krawędzi. Krawędź  $e$  jest reprezentowana jako para wierzchołków  $(u, v)$ , gdzie  $u, v \in V$ . Graf nieskierowany nie ma określonego kierunku dla krawędzi. Formalnie, graf nieskierowany można zdefiniować jako parę zbiorów:

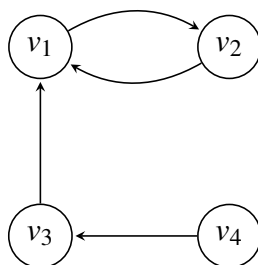
$$G = (V, E),$$

gdzie:

- $V$  jest zbiorem wierzchołków,  $V = \{v_1, v_2, \dots, v_n\}$ ,
- $E$  jest zbiorem krawędzi,  $E \subseteq \{\{u, v\} \mid u, v \in V\}$ .

Para  $\{u, v\}$  reprezentuje krawędź łączącą wierzchołki  $u$  i  $v$ . Graf nieskierowany nie różni krawędzi na podstawie ich kierunku, więc  $\{u, v\}$  jest równoważne  $\{v, u\}$ .

Graf skierowany, z kolei, ma krawędzie zorientowane, co oznacza, że istnieje określony kierunek przepływu. Na przedstawionym rysunku (Rysunek 1.2), został zilustrowany sposób reprezentacji graficznej grafu skierowanego. Zgodnie z definicją (Definicja 1.4), graf skierowany jest to forma grafu, w której jawnie określony jest sposób czytania wierzchołków. Oznacza to, że relacja między wierzchołkami może być interpretowana tylko w jedną stronę, stąd relacje w grafie skierowanym nazywamy relacjami jednokierunkowymi.



Rysunek 1.2: Przykład grafu skierowanego

#### Definicja 1.4. Graf skierowany

Niech  $G = (V, E)$  będzie grafem skierowanym, gdzie  $V$  oznacza zbiór wierzchołków, a  $E$  oznacza zbiór krawędzi skierowanych. Krawędź skierowana  $e$  jest reprezentowana jako para wierzchołków  $(u, v)$ , gdzie  $u, v \in V$ , i oznacza krawędź skierowaną z wierzchołka  $u$  do wierzchołka  $v$ . Formalnie, graf skierowany można zdefiniować, jako parę zbiorów:

$$G = (V, E),$$

gdzie:

- $V$  jest zbiorem wierzchołków,  $V = \{v_1, v_2, \dots, v_n\}$ ,
- $E$  jest zbiorem krawędzi skierowanych,  $E \subseteq V \times V$ .

Oprócz podstawowej teorii związanej z definiowaniem grafu w tej sekcji rozszerzymy definicję sub elementów grafu. Pierwszym elementem, który zostanie omówiony, jest wierzchołek. Jest on kluczowym elementem grafu reprezentującym dane. Przybliżymy najważniejszą wiedzę z nim związaną. Kolejnym elementem, który zostanie omówiony w większym stopniu szczegółowości, są krawędzie. Krawędzie w grafie stanowią sposób poruszania się po danych. Skupimy się na najważniejszych elementach definicji krawędzi oraz na relacjach, które



to krawędzi mają reprezentować. Ostatnim elementem tej sekcji będzie omówieni atrybutów w grafach. Atrybuty stanowią ważny element opisujący konkretne cechy wierzchołków. Przybliżam definicję atrybutu oraz sposób zapisu atrybutów.

### 1.2.1 Wierzchołki

Zgodnie z definicją (Definicja 1.5) w grafowych bazach danych węzły (nodes) pełnią funkcję reprezentacji obiektów lub koncepcji w bazie danych. Węzły są połączone ze sobą przez krawędzie (edges) lub relacje, które opisują różne rodzaje powiązań między obiektami. W ten sposób model grafu wykorzystany w bazach danych, umożliwia modelowanie i przechowywanie złożonych struktur danych, w których obiekty są połączone w sposób dynamiczny i zmienny.

#### **Definicja 1.5. Wierzchołek**

*Wierzchołek grafu, oznaczany jako  $v$ , jest jednym z elementów składowych grafu. Może reprezentować obiekt, punkt, stan lub dowolny inny element, który jest połączony z innymi wierzchołkami za pomocą krawędzi. Formalnie, wierzchołek jest elementem zbioru wierzchołków  $V$  w grafie  $G = (V, E)$ :*

$$v \in V.$$

Wierzchołki przybierają różne znaczenie w przypadku różnego podejścia do grafu. Rozważymy dwa przypadki grafu, graf skierowany i nieskierowany. Gdy rozważamy graf skierowany, wierzchołek może być początkowym lub końcowym punktem krawędzi, w zależności od kierunku krawędzi. W przypadku grafu nieskierowanego wierzchołek może być połączony z innymi wierzchołkami za pomocą krawędzi. Wtedy “komunikacja” między tymi wierzchołkami odbywa się w jednym kierunku. Warto także zaznaczyć, że wierzchołek może mieć przypisaną etykietę lub inną dodatkową informację. Dzięki temu jesteśmy w stanie w bardziej szczegółowy sposób, opisać co dany wierzchołek reprezentuje.

Na przykład, w bazie danych opisującej ludzi i ich relacje rodzinne, węzły mogą być używane do reprezentowania poszczególnych osób, a krawędzie mogą być używane do opisywania relacji rodzicielskich między nimi.

### 1.2.2 Krawędzie

Zgodnie z definicją (Definicja 1.6) krawędzie w grafowych bazach danych odgrywają rolę reprezentowania relacji lub powiązań między węzłami w grafie. Łączą one ze sobą dwa węzły, opisując relację między nimi. Krawędzie mogą być także oznaczone jako skierowane lub nieskierowane. Możemy także spotkać się z krawędziami oznaczonymi atrybutami, takimi jak na przykład data utworzenia lub rodzaj relacji. Pozwala to na bardziej szczegółowe opisanie powiązań między węzłami w grafie. Atrybutem krawędzi może być także waga, jest ona rozpatrywana podczas przechodzenia po grafie.

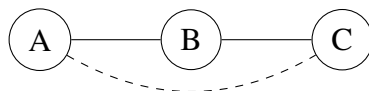
#### **Definicja 1.6. Krawędź**

*Niech  $G = (V, E)$  będzie grafem skierowanym lub nieskierowanym, gdzie  $V$  oznacza zbiór wierzchołków, a  $E$  oznacza zbiór krawędzi. Krawędź  $e$  jest reprezentowana jako para wierzchołków  $(u, v)$ , gdzie  $u, v \in V$ . W przypadku grafu nieskierowanego para  $(u, v)$  reprezentuje*

krawędź łączącą wierzchołki  $u$  i  $v$ , przy czym  $(u, v)$  jest równoważne  $(v, u)$ . Natomiast w przypadku grafu skierowanego, para  $(u, v)$  reprezentuje krawędź skierowaną z wierzchołka  $u$  do wierzchołka  $v$ .

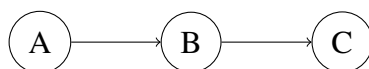
Krawędzie przede wszystkim służą do opisywania relacji, stanowią one podstawową informację, za pomocą której czytamy graf. Możemy wyróżnić następujące rodzaje relacji: przechodnie, nieprzechodnie, jednostronne i dwustronne. W dalszej części przedstawimy ich przykłady oraz ilustrację graficzną.

Relacje przechodnie (transitive relationships) jest to rodzaj relacji, w których obiekt "A" jest powiązany z obiektem "B", a obiekt "B" jest powiązany z obiektem "C". Oznacza to, że obiekt "A" jest również powiązany z obiektem "C" w sposób pośredni. Przykładem takiej relacji może być relacja "rodzeństwo", gdzie, jeśli "A" jest bratem "B", a "B" jest bratem "C", to "A" jest również bratem "C". Poniższy rysunek (Rysunek 1.3) prezentuje sposób ilustrowania tego typu relacji w postaci graficznej.



Rysunek 1.3: Przykład relacji przechodniej

Relacje nieprzechodnie (non-transitive relationships) jest to rodzaj relacji, w których obiekt "A" jest powiązany z obiektem "B", a obiekt "B" jest powiązany z obiektem "C", ale obiekt "A" nie jest powiązany z obiektem "C". Przykładem takiej relacji może być relacja "przyjaźń", gdzie, jeśli "A" jest przyjacielem "B", a "B" jest przyjacielem "C", to "A" nie jest automatycznie przyjacielem "C". Poniższy rysunek (Rysunek 1.4) prezentuje sposób ilustrowania tego typu relacji w postaci graficznej.



Rysunek 1.4: Przykład relacji nieprzechodniej

Relacje jednostronne (one-way relationships) jest to rodzaj relacji, w których jeden obiekt jest powiązany z drugim, ale drugi obiekt nie jest powiązany z pierwszym. Przykładem takiej relacji może być relacja "podległość", gdzie, jeśli "A" jest podwładnym "B", to "B" nie jest podwładnym "A". Poniższy rysunek (Rysunek 1.5) prezentuje sposób ilustrowania tego typu relacji w postaci graficznej.

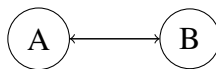
Podwładny Przełożony



Rysunek 1.5: Przykład relacji jednostronnej

Relacje dwustronne (two-way relationships) jest to rodzaj relacji, w których oba obiekty są powiązane ze sobą w obie strony. Przykładem takiej relacji może być relacja "małżeń-

stwo”, gdzie, jeśli “A” jest małżonkiem “B”, to “B” jest również małżonkiem “A”. Poniższy rysunek (Rysunek 1.6) prezentuje sposób ilustrowania tego typu relacji w postaci graficznej.



Rysunek 1.6: Przykład relacji dwustronnej

### 1.2.3 Atrybuty

Atrybuty zgodnie z definicją (Definicja 1.7) w grafowych bazach danych służą do przechowywania informacji o połączonych wierzchołkach i krawędziach. Atrybuty mogą być używane do opisywania zasobów, takich jak ludzie, produkty lub miejsca. Wyróżniamy trzy rodzaje typów atrybutów, to jest liczbowe, tekstowe oraz binarne. Atrybuty są przechowywane w postaci tabeli zawierającej klucz obiektu, nazwę atrybutu i wartość atrybutu.

#### Definicja 1.7. Atrybut

Niech  $G = (V, E)$  będzie grafem, gdzie  $V$  oznacza zbiór wierzchołków, a  $E$  oznacza zbiór krawędzi. Atrybut w grafie to dodatkowa informacja przypisana do wierzchołków lub krawędzi, która opisuje pewną cechę lub wartość pomagającą lepiej zrozumieć dane, lub relacje. Formalnie, atrybut w grafie można zdefiniować jako funkcję:

$$A : V \cup E \rightarrow X,$$

gdzie:

- $A$  jest funkcją atrybutu,
- $V \cup E$  oznacza zbiór wierzchołków i krawędzi,
- $X$  oznacza zbiór możliwych wartości atrybutu.

Klucz obiektu	Nazwa atrybutu	Wartość atrybutu
1	Cena	10.99
1	Nazwa	Książka o programowaniu
1	Dostępność	Tak
2	Cena	29.99
2	Nazwa	Telefon komórkowy
2	Dostępność	Nie
3	Cena	5.99
3	Nazwa	Długopis
3	Dostępność	Tak

Tabela 1.1: Przykładowa tabela atrybutów w grafowej bazie danych.

Zamieszczona wyżej tabela (Tabela 1.1) reprezentuje przykładowy sposób zapisu atrybutów przypisanych do wierzchołków. Tabela tego typu tak jak było nadmienione wcześniej,

zawiera przede wszystkim klucz wierzchołka oraz nazwę atrybutów. Liczba atrybutów może przypisana do jednego wierzchołka może być dowolna. Kluczowym elementem stanowiącym o sensie idei atrybutów jest trzecia kolumna tabeli. Dostarcza ona informacje, jaką wartość dla danego wierzchołka posiada dany atrybut.

## 1.3 Struktura i model danych

Na przestrzeni lat rozwoju baz danych opartych na modelu grafowym powstało wiele różnych podejść do opisywania samego modelu grafowego. Aktualnie jesteśmy w stanie znaleźć bardzo wiele różnych podejść. Struktura i modele stanowią podstawę elementu zwanego bazowy magazynowaniem danych. Definiują one sposób opisywania i oznaczania informacji na grafie. Różne podejścia wymagają odpowiedniego dostosowania formy zapisu grafu, czy to w formie natywnej, czy też poprzez dostosowanie odpowiedniej serializacji.

W niniejszej sekcji skupimy się na omówieniu podstawowych modeli danych wykorzystywanych w grafowych systemach baz danych. Każdy z omówionych modeli posiada swoje charakterystyczne cechy i dostarcza własne mechanizmy umożliwiające efektywne operacje na danych grafowych. Omówimy trzy podstawowe modele, to jest trójki RDF, grafy właściwości oraz hipergrafy. W każdym przypadku przedstawimy podstawowe założenia modelu, formalną definicję każdego z modeli grafów oraz przedstawimy odpowiednią reprezentację graficzną. Zostaną nakreślone zalety oraz wady danych modeli, oraz przyjrzymy się ich cechom szczególnym.

### 1.3.1 Hipergraf

Hipergraf zgodnie z definicją (Definicja 1.8) to struktura danych, która rozszerza pojęcie grafu poprzez umożliwienie krawędziom łączenie więcej niż dwóch wierzchołków. Krawędzie poprzez zmienioną definicję zastępujemy słowem hiperkrawędzi, aby odróżnić ten model od ogólnej definicji grafów. Model ten jest reprezentowany jako zbiór wierzchołków oraz zbiór hiperkrawędzi. Wierzchołki reprezentują elementy lub obiekty, natomiast hiperkrawędzie reprezentują grupy wierzchołków powiązanych ze sobą.

#### **Definicja 1.8. Hiper graf**

*Hiper graf to struktura danych rozszerzająca pojęcie grafu, w której krawędzie mogą łączyć więcej niż dwa wierzchołki. Formalnie, hiper graf jest definiowany jako:*

$$H = (V, E),$$

gdzie:

- $V$  jest zbiorem wierzchołków,  $V = \{v_1, v_2, \dots, v_n\}$ ,
- $E$  jest zbiorem hiperkrawędzi,  $E \subseteq 2^V \setminus \emptyset$ , czyli  $E$  to zbiór niepustych podzbiorów  $V$ .

Hiperkrawędzie niejako wyznaczają zupełnie inne grafy. Grafy te są zazwyczaj połączone ze sobą poprzez połączenie jednego lub więcej wierzchołków z danej hiperkrawędzi do wierzchołków innej hiperkrawędzi. Takie podejście umożliwia przetwarzanie danych wielopoziomowych. Ważnymi koncepcjami w hipergrafach są stopień wierzchołka (Definicja 1.9) i ranga hiperkrawędzi (Definicja 1.10).

**Definicja 1.9. Stopień wierzchołka**

Niech  $H = (V, E)$  będzie hipergrafem, gdzie  $V$  oznacza zbiór wierzchołków, a  $E$  oznacza zbiór hiperkrawędzi. Stopień wierzchołka w hipergrafie odnosi się do liczby hiperkrawędzi, połączonych z danym wierzchołkiem. Im większy stopień wierzchołka, tym więcej hiperkrawędzi jest z nim związanych. Formalnie, stopień wierzchołka  $v \in V$  w hipergrafie  $H$  jest definiowany jako:

$$\deg(v) = |\{e \in E : v \in e\}|,$$

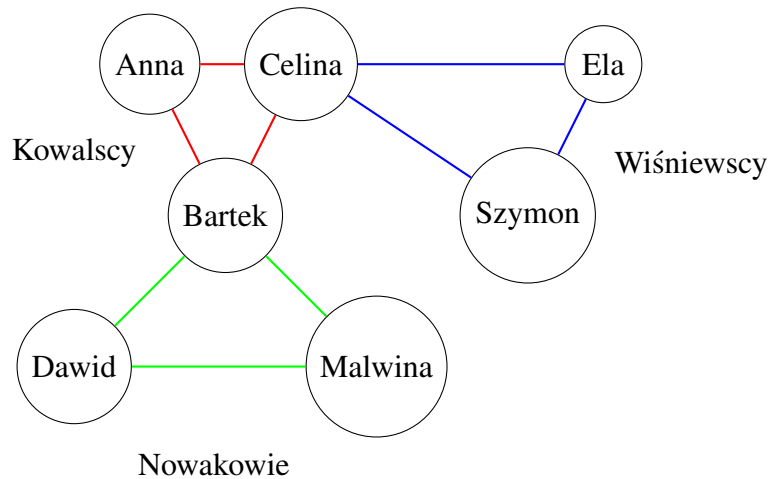
gdzie  $|\{e \in E : v \in e\}|$  oznacza liczbę hiperkrawędzi w  $E$ , zawierających wierzchołek  $v$ .

**Definicja 1.10. Ranga hiperkrawędzi**

Ranga hiperkrawędzi w hipergrafie to liczba wierzchołków, które są związane z daną hiperkrawędzią. Im większa ranga, tym więcej wierzchołków jest łączonych przez tę hiperkrawędź. Wysoka ranga hiperkrawędzi może wskazywać na silne powiązanie między większą liczbą wierzchołków, podczas gdy niska ranga może oznaczać mniej istotne związki. Formalnie, ranga hiperkrawędzi  $e$  w hipergrafie  $H = (V, E)$  jest definiowana jako:

$$\text{ranga}(e) = |e|,$$

gdzie  $|e|$  oznacza liczbę wierzchołków należących do hiperkrawędzi  $e$ .



Rysunek 1.7: Przykład hiper grafu na relacjach rodzinnych

Omawiając przykład (Rysunek 1.7) widzimy trzy grupy wierzchołków oznaczone przez trzy hiperkrawędzie. Każda z hiperkrawędzi oznacza relacje rodzinne między wierzchołkami. Kolejno czerwona hiperkrawędź wyznacza rodzinę Kowalskich, zielona hiperkrawędź wyznacza rodzinę Nowaków natomiast trzecia hiperkrawędź, która oznaczona jest kolorem niebieskim, wyznacza rodzinę Wiśniewskich. Jednak należy zauważyć, że wierzchołki Bartek oraz Celina należą do dwóch rodzin. Oznacza to, że odpowiednio dla Bartka, Malwina i Dawid są rodzicami, oraz w przypadku Celiny rodzicami są Ela i Szymon. Rodzicami Anny są natomiast Celina i Bartek.

Przedstawienie tych danych w hipergrafie zarysowuje nam jasno hierarchię genealogiczną wymienionych rodzin. Widząc ten przykład, łatwo wyobrazić sobie wielopoziomowość, jaką umożliwia nam wykorzystanie hipergrafów. Kluczowe jest także wykorzystanie informacji o stopniu wierzchołka oraz rangi hiperkrawędzi. W celu określenia stopnia wierzchołka weźmy za przykład Bartka. Posiada on połączenie do dwóch hiperkrawędzi, do rodziny Kowalskich oraz Nowaków. Wynika z tego fakt, że stopień wierzchołka oznaczonego imieniem Bartek ma wartość 2. Aby określić rangę hiperkrawędzi, należy wybrać rodzinę, która nas interesuje. Dla przykładu weźmy rodzinę Wiśniewskich, w tej rodzinie widzimy trzy połączone wierzchołki. Oznacza to, że ranga hiperkrawędzi oznaczonej jako Wiśniewscy ma wartość 3.

Zilustrowany graf jest oczywiście zbyt prosty, aby zauważyć znaczące różnice między tymi dwiema zmiennymi. W realnym przypadku liczby te byłyby prawdopodobnie znacznie większe i za ich pomocą moglibyśmy określić na przykład główne miejsce wejściowe do grafu w przypadku znaczącego stopnia wierzchołka. Byłoby to dobrym wyborem, ponieważ oznacza to nie mniej nie więcej, że ten wierzchołek jest połączony z największą ilością grup, co skracałoby proces indeksowania wierzchołków.

Hiper grafy umożliwiają także znaczące zmniejszenie liczby krawędzi. Dobrze zoptymalizowany graf będzie posiadał mniejszą ilość wierzchołków, a co za tym idzie mniejszą liczbę krawędzi. Jest to bardzo duży atut tego modelu, ponieważ złożone wartości mogą zostać przedstawione w znacznie optymalniejszy sposób. Sprawi to, że będzie potrzebne mniej pamięci do przechowywania danych w tej formie oraz do znaczącego wzrostu wydajności metod przetwarzających.

Model ten posiada jednak także wady, jedną z tych wad jest na przykład trudność w analizie grafu przez człowieka. Oczywiście bazy danych rozważamy jako system informatyczny, w którym jedynym wkładem człowieka jest pobieranie, wprowadzanie i wyciąganie danych. Jednak powstaje duży problem w kwestii ilustrowania tego typu grafów. Zazwyczaj złożonego hiper grafu nie jesteśmy w stanie przedstawić za pomocą grafiki dwuwymiarowej. Jest to duży problem, jednak tak jak zostało zaznaczone, bazy danych są systemem. Najważniejszym elementem tego systemu jest interfejs komunikacyjny z innymi systemami i człowiekiem.

### 1.3.2 Graf właściwości

Grafy właściwości są zdecydowanie jednym z najważniejszych modeli struktur danych wykorzystywanych w dziedzinie baz danych. Charakteryzują się dużą elastycznością, pozwalając na reprezentację skomplikowanych struktur danych w sposób intuicyjny i łatwy do interpretacji. Ze względu na swoją strukturę, grafy własnościowe są idealnym narzędziem do modelowania i analizowania skomplikowanych relacji, takich jak sieci społeczne, modele zależności w systemach informatycznych czy struktury związane z analizą big data.

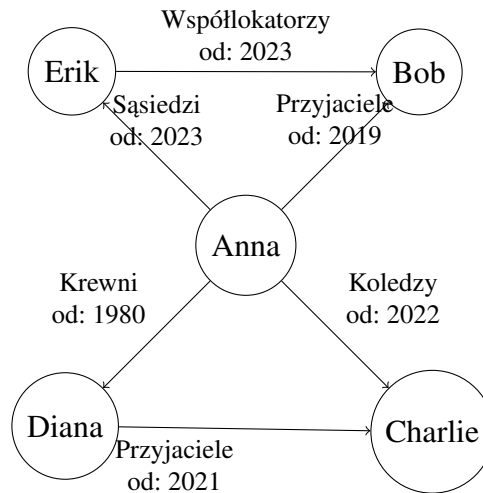
Zatem zgodnie z definicją (Definicja 1.11) graf własnościowy składa się z wierzchołków i krawędzi, z których każdy posiada unikalny identyfikator i zestaw par klucz-wartość, nazywany „właściwościami”. Każdy wierzchołek może mieć przypisane różne właściwości, które są przechowywane jako atrybuty wierzchołka. Na przykład, dla wierzchołka reprezentującego osobę, można przypisać atrybuty takie jak imię, wiek, adres itp. Właściwości mogą być również przedstawione jako wierzchołki połączone z odpowiednimi obiektami, co umożliwia bardziej złożone struktury grafu.

**Definicja 1.11. Graf właściwości**

Formalnie, model grafu właściwości  $G$  jest definiowany jako uporządkowana czwórka  $G = (V, E, \lambda_V, \lambda_E)$ , gdzie:

- $V$  jest zbiorem wierzchołków,
- $E$  jest zbiorem krawędzi, które są uporządkowanymi parami wierzchołków, oznaczającymi relację między nimi,
- $\lambda_V : V \rightarrow P(V)$  jest funkcją przypisującą wierzchołkom zbiór par klucz-wartość, reprezentujący ich własności,
- $\lambda_E : E \rightarrow P(E)$  jest analogiczną funkcją dla krawędzi.

W celu lepszego zrozumienia modelu grafów właściwości posłużymy się przykładem w formie graficznej (rysunek 1.8). Przedstawiony przykład ma na celu pokazanie, zastosowanie modelu grafów właściwości do reprezentowania i analizowania prostej sieci relacji w sposób jasny i zrozumiały. Ukazany został scenariusz społeczny, w którym pięć osób Anna, Bob, Charlie, Diana i Erik są powiązane ze sobą różnymi relacjami. Relacje te obejmują przyjaźń, współpracę, pokrewieństwo, sąsiedztwo i współlokatorstwo.



Rysunek 1.8: Przykładowy graf właściwości przedstawiający relacje społeczne

Przedstawiony graf zawiera pięć wierzchołków oraz sześć krawędzi, będących relacjami między przedstawionymi osobami. Każda z krawędzi zawiera właściwość opisującą początek tej relacji. Możemy zauważyć na pierwszy rzut oka, że najdłuższa relacja występuje między Anną a Dianą. Są one krewnymi od 1980 roku.

W środku grafu znajduje się Anna, która posiada jakąkolwiek relację z każdym innym wierzchołkiem. Należy zwrócić uwagę, że przedstawiony graf jest grafem skierowanym. Możemy wyczytać następujące relacje dla Anny, jest sąsiadką Erika od 2023, jest przyjaciółką Boba od 2019, jest krewną Diany, odkąd się urodziła, oraz jest koleżanką Charliego. Można

także zauważyć ważny element, jako iż Anna jest również sąsiadką Boba przez relację między Bobem i Erikiem, ponieważ są współlokatorami od 2023 roku. Ostatnim elementem analizy jest relacja Diany i Charliego, są oni przyjaciółmi od 2021.

Zilustrowany przykład jest dosyć uproszczonym modelem, modele wytworzone podczas pracy realnego systemu mogą być znacznie bardziej złożone. Przykład ten jednak przedstawia najważniejsze założenia grafów właściwości. Zaznaczmy tutaj, że oprócz opisywania relacji atrybutami możemy zrobić podobną operację na wierzchołkach. Do aktualnego przykładu możemy na przykład dopisać etykietę informującą o zawodach danych osób.

Model grafów właściwości łączy w sobie wszystkie zalety modeli opartych o grafy. Dodatkowo znacząca wspomaga łatwość odczytu danych z grafu. Wynika to z faktu, że zarówno atrybuty wierzchołków, jak i atrybuty relacji mogą zostać wykorzystane do odczytu danych. Możemy także konstruować zapytania wykorzystujące wzorce, występujące w grafie.

Analizując ten model, musimy także zwrócić uwagę na fakt, iż wydajność przetwarzania tego modelu nie jest liniowa. W momencie, gdy objętość grafu wzrośnie znacząco, mogą pojawić się problemy z wydajnością. Operacje wyszukiwania i analizy danych mogą być czasochłonne, zwłaszcza jeśli zapytania obejmują złożone wzorce połączeń i duże zbiory właściwości.

Mówiąc o wadach tego modelu, nie możemy ominąć bardzo ważnego faktu. Obecnie model ten nie posiada standardu, który byłby powszechnie akceptowany. Aktualnie rozwijane systemy baz danych w oparciu o ten model wykorzystują swoje własne indywidualne podejście. Jest to jednak wada, która w najbliższym czasie może stać się nieaktualna. Omawiany w pracy standard GQL ma być odpowiedzią na ten problem. Ma on stanowić podstawowy standard, który będzie duchowym odpowiednikiem SQL w przypadku modelu relacyjnego.

### 1.3.3 Trójki RDF

Model trójek RDF (Resource Description Framework) zgodnie z definicją (Definicja 1.12) oparty jest o trójki składające się z zasobu, właściwości oraz wartości. W oparciu o te trójki budowany jest graf skierowany, w którym węzłami są zasoby a krawędziami właściwości. Każdy z zasobów wprowadzonych do tego grafu posiada unikalny identyfikator. Tym identyfikatorem jest URI (Uniform Resource Identifier), który jest powszechnym standardem wykorzystywanym w dzisiejszym internecie i nie tylko. Cecha ta pozwala zachować globalną jednoznaczność odnoszącą się do identyfikacji zasobów. Dzięki temu również możemy łączyć dane, które pochodzą z różnych źródeł.

#### **Definicja 1.12. Trójki RDF**

*Niech  $\mathcal{G}$  będzie grafem RDF, składającym się z węzłów i krawędzi. Węzły reprezentują zasoby, a krawędzie reprezentują relacje między zasobami Formalnie, graf RDF można zdefiniować jako trójkę  $\mathcal{G} = \{V, E, L\}$ ,*

*gdzie:*

- $V$  to zbiór węzłów, gdzie każdy węzeł reprezentuje unikalny identyfikator zasobu.
- $E \subseteq V \times V$  to zbiór krawędzi, gdzie każda krawędź reprezentuje relację między dwoma zasobami.



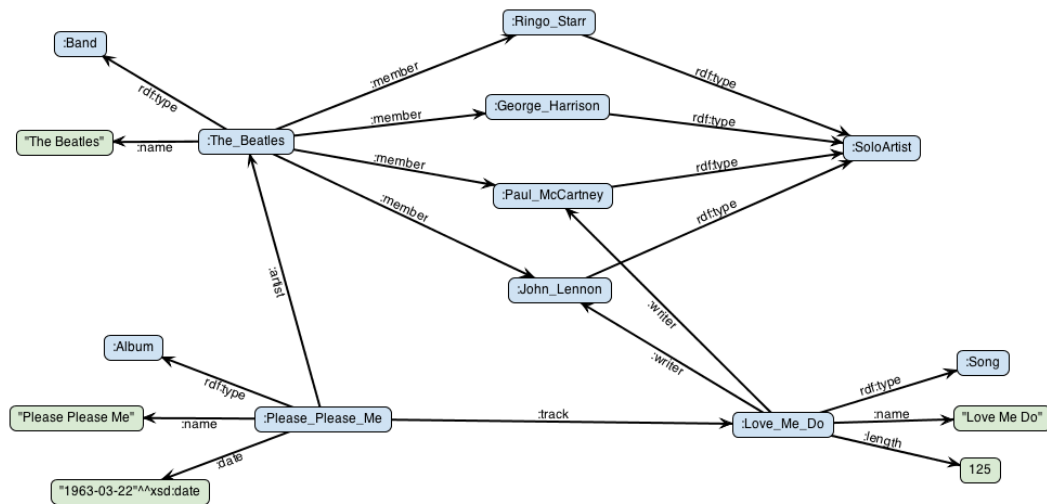
- $L : E \rightarrow L_v \times L_p \times L_o$  to funkcja etykietująca, która przypisuje każdej krawędzi etykiety w formie trójki (etykieta węzła źródłowego, etykieta relacji, etykieta węzła docelowego).

Zasoby w modelu trójek RDF mogą zostać opisane wieloma właściwościami. Umożliwia to przedstawienie zasobu pod kątem różnych aspektów. Oprócz tego model ten umożliwia traktowanie informacji w sposób semantyczny. Oznacza to, że danym można przypisać znaczenie poprzez zastosowanie ontologii (Definicja 1.13). Ułatwia to znacząco rozumienie danych przedstawionych na grafie. Kolejnym ważnym elementem modelu trójek RDF jest możliwość wymiany danych semantycznych między różnymi systemami. Zarówno dane, jakie wprowadzamy do modelu, jak i właściwości nie posiadają określonego schematu, co za tym idzie, do istniejących danych możemy dodać dowolnie inne właściwości, nawet jeżeli takie właściwości jeszcze nie istnieją.

### Definicja 1.13. Ontologia

*Ontologia to formalny model reprezentacji wiedzy, który opisuje pojęcia, relacje i aksjomaty w określonej dziedzinie. Jest to narzędzie stosowane w dziedzinach takich jak sztuczna inteligencja i sieci danych semantycznych, które pomagają w organizacji, uporządkowaniu i zrozumieniu wiedzy w danym obszarze.*

Rozpatrzmy ilustrację (Rysunek 1.9) na której został przedstawiony graf oparty o trójki RDF. Reprezentuje on związki między różnymi informacjami dotyczącymi zespołu muzycznego The Beatles. Poszczególne węzły reprezentują różne aspekty, takie jak członkowie zespołu, albumy, utwory, a także ich powiązania z innymi artystami i zespołami muzycznymi. Krawędzie między węzłami reprezentują relacje, jakie zachodzą pomiędzy tymi informacjami.



Rysunek 1.9: Przykład grafu RDF

W przykładzie, węzeł oznaczony jako "The Beatles" reprezentuje sam zespół, a krawędzie wychodzące z tego węzła wskazują na członków zespołu, takich jak "John Lennon", "Paul McCartney", "George Harrison" i "Ringo Starr". Węzły "John Lennon" i "Paul

McCartney” są powiązane z albumami, takimi jak “Rubber Soul” i “Revolver”, poprzez krawędzie z etykietami reprezentującymi relację “napisał”. Podobnie, można zauważyć krawędzie między węzłami “The Beatles” i “Bob Dylan”, co sugeruje jakąś formę powiązania między tymi artystą a zespołem. Te krawędzie mogą reprezentować współpracę, inspirację lub inne relacje. W ten sposób, graf RDF pozwala na reprezentację i wizualizację złożonych relacji między różnymi danymi, dotyczącymi zespołu muzycznego The Beatles.

Wadami tego modelu zdecydowanie jest problem skomplikowanych zapytań. Mogą one być bardziej złożone niż w innych modelach. Wymaga to wykorzystania specjalistycznych narzędzi i technik, aby zapewnić efektywność i wydajność. Dodatkowo podobnie jak w przypadku grafów własności, również ten model nie posiada globalnego standardu. Jednak problem tutaj jest bardziej złożony, wynika on z braku określonych schematów reprezentacji danych, zwłaszcza jeśli wymagają przeszukiwania wielu połączeń między trójkami.

## 1.4 Języki zapytań i modyfikacji do grafowych baz danych

Języki zapytań w bazach danych, służą do tworzenia i modyfikowania ich struktury. Pełnią one także rolę interfejsu umożliwiającego odczyt informacji w celu uzyskania określonych rezultatów. Najbardziej popularnym językiem zapytań jest SQL (Structured Query Language). Odnosi się on jednak do baz danych opartych o model relacyjny. W przypadku grafowych baz danych również mamy do czynienia z podobnymi rozwiązaniami.

Język tego typu dysponuje dwoma rodzajami zapytań, zapytaniem strukturalnym (structural queries) oraz zapytaniem o dane (data queries).

Zapytania strukturalne umożliwiają wyszukiwanie danych oraz manipulację nimi. Wyszukiwanie może odbywać się z poziomu wierzchołków, jak i krawędzi. Umożliwiają one również przechodzenie po grafie, odbywa się to na zasadzie przechodzenia z jednego wierzchołka do drugiego w oparciu o relacje. Przechodzenie grafu możemy przeprowadzić przeszukiwaniem grafu w głąb (Definicja 1.14) lub wszerz (Definicja 1.15) w celu odnalezienia określonych połączeń pomiędzy węzłami. Do elementów zapytań strukturalnych możemy również zaliczyć filtrację danych, tworzenie nowych węzłów i krawędzi, aktualizację i usuwanie danych a także wykonanie analiz i obliczeń na danych.

### **Definicja 1.14. Przeszukiwanie w głąb (Depth-first search)**

*Przeszukiwanie w głąb w grafie to algorytm przeszukiwania grafu, który polega na eksplorowaniu jak najgłębiej danego gałęzi przed powrotem i kontynuowaniem przeszukiwania innych gałęzi*

### **Definicja 1.15. Przeszukiwanie wszerz (Breadth-first search)**

*Przeszukiwanie wszerz w grafie polega na odwiedzaniu wierzchołków grafu w kolejności zgodnej z ich odległością od wybranego wierzchołka początkowego. Algorytm rozpoczyna od wierzchołka początkowego, a następnie odwiedza wszystkich sąsiadów tego wierzchołka, a potem odwiedza sąsiadów sąsiadów, i tak dalej, aż wszystkie wierzchołki zostaną odwiedzone lub osiągnięte, zostanie określone kryterium zakończenia.*

Zapytania o dane odnoszą się do operacji na właściwościach węzłów i krawędzi oraz na danych przechowywanych w grafie. Tego typu zapytania umożliwiają także sortowanie

pobieranych danych, agregację tych danych, a także grupowanie i filtrowanie. Dzięki zapytaniom tego typu jesteśmy w stanie odkrywać związki zachodzące między danymi, a także jesteśmy w stanie wyszukiwać wzorce danych.

Operacje na grafowych bazach danych są zgoła inne w porównaniu do operacji wykonywanych na tradycyjnych relacyjnych bazach danych, ze względu na wykorzystaną strukturę grafu. Zrozumienie operacji możliwych do wykonania na grafowych bazach danych, takich jak tworzenie, modyfikowanie, zapytania i usuwanie węzłów i krawędzi, jest niezbędne dla pełnej eksploatacji potencjału tych systemów. Przedstawmy więc podstawowe funkcje CRUD w kontekście grafowych baz danych.

Tworzenie, w podejściu grafowym polega na generowaniu nowych węzłów lub krawędzi w grafie. W zależności od użytego systemu zarządzania bazą danych może to wymagać definiowania typu lub właściwości nowo tworzonego elementu. Na przykład, przy tworzeniu węzła można określić jego typ, nazwę i inne właściwości, a przy tworzeniu krawędzi można określić typ relacji, kierunek i inne informacje.

Odczytywanie, w podejściu grafowym polega na pobieraniu informacji o węzłach lub krawędziach grafu. Można to zrobić poprzez zapytanie, odwołujące się do różnych aspektów grafu, takich jak typ węzła, właściwości krawędzi, kierunek krawędzi, i tak dalej. Wynikiem odczytywania może być jeden lub wiele węzłów, lub krawędzi, a także całe pod-grafy.

Aktualizacja, w podejściu grafowym polega na modyfikacji istniejących węzłów lub krawędzi. Operacja ta może na przykład obejmować zmianę typu węzła, dodanie, usunięcie lub modyfikację właściwości węzła, lub krawędzi, zmianę kierunku krawędzi.

Usuwanie, w podejściu grafowym to operacja, która usuwa węzły lub krawędzie z grafu. Usunięcie węzła często pociąga za sobą usunięcie wszystkich powiązanych z nim krawędzi, zależy to jednak od specyfiki używanego systemu zarządzania bazą danych.

Zapytania do grafowych baz danych są kluczowym elementem stanowiącym interfejs komunikacyjny integrujący się z różnymi systemami. Języki zapytań, takie jak Cypher dla Neo4j czy SPARQL dla RDF, umożliwiają interakcję z danymi na bardzo wysokim poziomie abstrakcji, umożliwiając skomplikowane zapytania i analizy. Przez zrozumienie tych języków, będziemy w stanie pełniej zrozumieć, jak mogą być wykorzystywane grafowe bazy danych.

W tej sekcji oprócz wiedzy, która została już przedstawiona, przybliżymy popularne języki zapytań wykorzystywane w grafowych bazach danych. Zostaną przedstawione przykłady składni danych języków wraz z opisem operacji, które wchodzą w skład podstawowych operacji języków zapytań.

### 1.4.1 Cypher

Cypher jest deklaratywnym językiem zapytań stworzonym specjalnie dla bazy danych Neo4J. Został stworzony w celu umożliwienia wygodnego i efektywnego zarządzania danymi w modelu grafowym. Charakteryzuje się on składnią podobną do języka naturalnego, co ułatwia jego czytelność i zrozumienie. Zapytania w tym języku opisują wzorce grafowe, które mają być znalezione lub modyfikowane. W takim zapytaniu można określić węzły, relacje i ich własności, a także warunki, na podstawie których będą wyszukiwane lub modyfikowane dane.

Język ten oferuje takie klauzule jak MATCH, WHERE, RETURN, CREAT, CREATE UNIQUE, MERGE, DELETE, SET, FOREACH, UNION, WITH.

Klauzula MATCH odpowiada za wyszukiwanie wzorców w grafie. Podajemy w niej schemat węzłów i relacji, który chcemy wyodrębnić z bazy danych. Można w niej precyzyjnie określić jakie informacje nas interesują oraz czy założone warunki są spełnione. Wzorec wykorzystywany w klauzuli musi być złożony z przynajmniej jednego węzła oraz przy dwóch węzłach musimy podać przynajmniej jedną relację.

```
1 MATCH (person:Person) -[:FRIEND]->(friend:Person)
2 WHERE person.name = "John"
3 RETURN person, friend
```

#### Przykład 1.1: Przykład klauzuli MATCH w Cypher

Przeanalizujmy przykładowe (Przykład 1.1) zapytanie wyszukujące w grafie węzły typu “Person”, które są połączone relacją “FRIEND” z innymi węzłami typu “Person”. Następnie jest sprawdzany warunek, że właściwość “name” węzła “person” jest równa wartości “John”. Ostatnim elementem przytoczonego przykładu jest zwrócenie ewentualnych znalezionych węzłów “person” i “friend”. Możemy tutaj także dodać takie klauzule jak na przykład ORDER BY czy GROUP BY w przypadku gdy chcemy wykonać dodatkowe operacje na zwracanych węzłach.

Kolejną klauzulą przedstawioną na przykładzie jest klauzula WHERE. Umożliwia on sprawdzenie właściwości danego węzła, oznacza to, że oprócz narzuconego wzorca w klauzuli MATCH sprawdzamy kolejne węzły na wypadek wystąpienia określonej wartości. W tym przypadku klauzula jest ograniczona jedynie do węzłów. Relacje powinny być weryfikowane w klauzuli MATCH.

Ostatnią Klauzulą obecną w przykładzie jest klauzula RETURN. Jest on elementem kończącym zapytanie, w nim określamy jakie węzły mają być zwrócone w wyniku zapytania. Oprócz węzłów możemy również zwracać określone wartości właściwości węzła.

Wykorzystując klauzulę CREATE mamy możliwość tworzenia nowych węzłów oraz relacji między nimi. Poniższy przykład (Przykład 1.2) przedstawia przykładową składnię tego zapytania.

```
1 CREATE (person1:Person {name: 'John'})
2       -[:FRIEND]->
3       (person2:Person {name: 'Jane'})
```

#### Przykład 1.2: Przykład klauzuli CREATE w Cypher

W tym przykładzie tworzone są dwa węzły typu “Person” oraz relacja o typie “FRIEND” między nimi. Relacja między węzłami jest tworzona z wykorzystaniem etykiety odpowiedniej etykiety, którą możemy później wykorzystać w zapytaniu MATCH. Możemy dodatkowo wykorzystać klauzulę UNIQUE. Dzięki niej mamy pewność, że wprowadzony węzeł lub relacja nie powtórzy się ponownie w grafie. Działa to oczywiście na zasadzie sprawdzenia, jeżeli dana struktura już istnieje, to operacja nie wykona żadnych zmian. W przypadku, gdy podana struktura nie istnieje w grafie, to operacja stworzy ją.

Klauzula MERGE jest wykorzystywana do scalania lub tworzenia węzłów i relacji. Jej głównym zadaniem jest upewnienie się, że podany wzorec istnieje w bazie, jeżeli tak zostaje on nadpisany, jeżeli nie zostaje on stworzony. Ważną różnicą między CREATE UNIQUE a MERGE jest taka, że w przypadku znalezienia danych o podanym wzorcu, jest ono

rozszerzane o dodatkowe elementy. To rozwiązanie pomaga zapobiegać zjawisku duplikowania danych oraz kontrolować ich unikalność. Składnia tego zapytania została przedstawiona na poniższym przykładzie (Przykład 1.3). Analizując przykład, możemy zaobserwować dwa przypadki. W momencie, kiedy węzeł nie istnieje, to jest on tworzony i dodatkowo jego wartość właściwości “createdDate” jest ustawiana na aktualną datę. Rozpatrując przypadek, że węzeł już istnieje, wtedy wartość właściwości “lastAccessed” zostaje ustawiona na aktualną datę.

```
1  MERGE (person:Person {name: 'John'})
2  ON CREATE SET person.createdDate = timestamp()
3  ON MATCH SET person.lastAccessed = timestamp()
```

Przykład 1.3: Przykład klauzuli MERGE w Cypher

Klauzula DELETE umożliwia usuwanie węzłów, relacji i właściwości. Jest to polecenie przydatne do organizacji i zachowania spójności danych. Mamy możliwość usuwania danych, które są przedawnione lub zawierają błędy. Klauzulę tę można wykorzystywać wraz z innymi klauzulami takimi jak MATCH i WHERE w celu określenia dokładnych danych do usunięcia. Składania zapytania została przedstawiona na poniższym przykładzie (Przykład 1.4). Przykład ten przedstawia zapytanie, które usuwa węzeł o etykiecie “Person” oraz właściwości “name”, która jest ustalona na wartość “John”.

```
1  MATCH (person:Person {name: 'John'})
2  DELETE person
```

Przykład 1.4: Przykład klauzuli DELETE w Cypher

Wykorzystanie klauzuli SET umożliwia modyfikowanie wartości właściwości węzłów i relacji. Pozwala także na tworzenie nowych właściwości. Klauzula SET jest wykorzystywana w połączeniu z klauzulą MATCH. Składania zapytania została przedstawiona na poniższym przykładzie (Przykład 1.5). Przykład ten przedstawia wykorzystanie klauzuli MATCH w celu określenia celu zmiany wartości. Za pomocą klauzuli SET ustala się wartości “age” oraz “city”.

```
1  MATCH (person:Person {name: 'John'})
2  DELETE person
```

Przykład 1.5: Przykład klauzuli SET w Cypher

Klauzula FOREACH służy do przetwarzania elementów kolekcji w sposób iteracyjny i wykonania na nich określonych operacji. Umożliwia nam to dynamiczne przetwarzanie i modyfikację danych z wykorzystaniem większych zbiorów danych. Dla każdego elementu przetwarzanego przez klauzulę FOREACH możemy wykonywać takie operacje jak tworzenie nowych wierzchołków, aktualizacja właściwości czy usuwanie oraz inne. Z wykorzystaniem tej klauzuli możemy w szybki i wygodny sposób przetwarzać dane, które na przykład otrzymaliśmy w wyniku zapytania MATCH. Składania zapytania została przedstawiona na poniższym przykładzie (Przykład 1.6). W tym przykładzie iterujemy po kolekcji “persons”. Dla każdego elementu w tej kolekcji tworzony jest nowy węzeł z etykietą “Person” i właściwością “name” ustawioną na wartość wyciągniętą z tegoż elementu.

```
1  FOREACH (person IN persons
2  | CREATE (p:Person {name: person.name}))
```

Przykład 1.6: Przykład klauzuli FOREACH w Cypher

Klauzula UNION umożliwia łączenie wyników z więcej niż jednego zapytania MATCH. Jest to bardzo przydatne narzędzie, które umożliwia wybieranie danych z dwiema różnymi cechami, na których dla przykładu chcemy wykonać jedną operację. Składania zapytania została przedstawiona na poniższym przykładzie (Przykład 1.7). Przykład przedstawia połączenie wyników dwóch różnych zapytań MATCH. Należy zwrócić jednak uwagę na fakt, że elementy zwracane z obu zapytań mają przypisywany taki sam alias. Dane te wtedy mają taką samą sygnaturę mimo że pochodzą z dwóch zupełnie innych typów węzłów.

```
1 MATCH (person:Person)
2 WHERE person.age > 30
3 RETURN person.name AS name
4 UNION
5 MATCH (actor:Actor)
6 WHERE actor.movies > 5
7 RETURN actor.name AS name
```

Przykład 1.7: Przykład klauzuli UNION w Cypher

Klauzula WITH pozwala określić, w jakiej formie dane są przekazywane w kolejnych etapach wykonywania zapytań. Może ona zawierać takie operacje jak agregacja, sortowanie, grupowanie, filtrowanie oraz modyfikowanie. Składania zapytania została przedstawiona na poniższym przykładzie (Przykład 1.7). Przykład ten pokazuje wykorzystanie zapytania MATCH, które wyszukuje dane po określonym założeniu i wzorcu. Zwracana jest wartość właściwość “name”, jednak za pomocą klauzuli WITH dokonujemy modyfikacji tej wartości i zmieniamy wszystkie litery na duże.

```
1 MATCH (person:Person)
2 WHERE person.age > 30
3 WITH toUpper(person.name) AS name
4 RETURN name
```

Przykład 1.8: Przykład klauzuli WITH w Cypher

## 1.4.2 PGQL

PGQL (Property Graph Query Language) jest to język zapytań skupiony głównie na aspektach wyszukiwania, filtrowania i analizie danych. Posiada on zaawansowane funkcje do wyrażania złożonych wzorców grafowych oraz do manipulowania danymi w sposób intuicyjny. Podstawę syntaktyki tego języka stanowią takie klauzule jak: SELECT, FROM, MATCH i WHERE. Język ten jest zbudowany na bazie SQL, oferując możliwość wykorzystania wzorców grafowych w dobrze znanej składni użytkownikom. Podejście to jest bardzo pomocne osobom, które są zaznajomione z relacyjnym modelem i wchodzi w przestrzeń modelu grafowego.

Oprócz podstawowych operacji, które udostępnia język SQL, PGQL daje możliwość odzukiwania wzorców o stałej, jak i zmiennej długości. Wzorce o stałej długości to typ wzorca, który zawiera stałą określoną liczbę węzłów i krawędzi na rozwiązanie. Typy wierzchołków i krawędzi możemy dowolnie definiować za pomocą wyrażeń będących jedną z dwóch etykiet, *friend\_of* oraz *sibling\_of*. Oznacza to, że wzór krawędzi jest wysokopoziomową wersją operacji join, która łączy wiele typów encji w jednym momencie.

Wzorce o zmiennej długości wzorców opierają się o mechanizmy rekurencyjne. Do porównywania węzłów i krawędzi wykorzystują one takie symbole jak \*, +, ? lub {*n,m*}. Pozwala to osiągać informacje czy graf jest domknięty oraz pozwala określić najtańszą i najkrótszą drogę. Tabela 1.2 opisuje wszystkie możliwe symbole do wykorzystania we wzorcach o zmiennej długości.

Tak jak zostało już nadmienione, składania tego język nie odbiega znacząco od tego co znamy z SQL. Dotyczy to wszystkich elementów zawierających się w zbiorze funkcji CRUD. Główną zmianą jest wprowadzenie klauzuli MATCH. Poniżej zamieszczony przykład (Przykład 1.9) przedstawia zapytanie SELECT które jako dane z tabeli przyjmuje wynik podzapytania MATCH.

```

1  SELECT fof.name, COUNT(friend) AS num_common_friends
2  FROM MATCH (p:Person)
3           -[:has_friend]->
4           (friend:Person)
5           -[:has_friend]->
6           (fof:Person)
7  WHERE NOT EXISTS (SELECT * FROM MATCH (p)-[:has_friend]->(fof))

```

Przykład 1.9: Przykład klauzuli MATCH w PGQL

Symbol	Znaczenie	Dopasowania
*	zero lub więcej	Droga łącząca dwa węzły z wykorzystaniem zerowej, lub niezerowej liczby dopasowań dla danego wzorca.
+	jeden lub więcej	Droga łącząca dwa węzły z wykorzystaniem jednego, lub więcej dopasowań dla danego wzorca.
?	opcjonalne	Droga łącząca dwa węzły z wykorzystaniem jednego lub zera dopasowań dla danego wzorca.
{ <i>n</i> }	dokładnie <i>n</i>	Droga łącząca dwa węzły z wykorzystaniem <i>n</i> dopasowań dla danego wzorca.
{ <i>n</i> ,}	<i>n</i> lub więcej	Droga łącząca dwa węzły z wykorzystaniem <i>n</i> , lub więcej dopasowań dla danego wzorca.
{ <i>n,m</i> }	między <i>n</i> a <i>m</i> (inkluzywne)	Droga łącząca dwa węzły z wykorzystaniem dopasowań w przedziale od <i>n</i> do <i>m</i> dla danego wzorca.
{, <i>m</i> }	między 0 a <i>m</i> (inkluzywne)	Droga łącząca dwa węzły z wykorzystaniem dopasowań w przedziale od 0 do <i>m</i> dla danego wzorca.

Tabela 1.2: Symbole dostępne we wzorcach ze zmienną długością (PGQL)

Oprócz odczytu danych PGQL umożliwia wprowadzanie, modyfikowanie, a także usuwanie. Możemy wprowadzać takie rzeczy jak węzły, krawędzie, etykiety, a także właściwości. Obsługiwane jest także zmienianie wartości oraz usuwanie czy to węzłów, czy krawędzi.



Zamieszczony poniżej przykład (Przykład 1.10) ilustruje wykorzystanie klauzuli insert. Zapytanie to wprowadza dwie zmiany, co jest możliwe dzięki umożliwieniu wprowadzania kilku klauzul VERTEX i EDGE. Dzięki temu zapytaniu prowadzimy nowy wierzchołek oraz nową krawędź.

```
1  INSERT VERTEX x LABELS (Person)
2      PROPERTIES (x.name = 'John'),
3      EDGE e BETWEEN x AND y LABELS (knows)
4      PROPERTIES (e.since = DATE '2017-09-21')
5  FROM MATCH (y)
6  WHERE y.name = 'Jane'
```

Przykład 1.10: Przykład klauzuli INSERT w PGQL

Warto wspomnieć o etykietach, ponieważ aktualnie możliwe jest przypisanie tylko jednej etykiety zarówno do relacji, jak i do wierzchołka. Ta kwestia może ulec zmianie w przyszłości, jednak działa tutaj mechanizm, z którym warto się zapoznać. Otóż w przypadku braku podania etykiety dla jednego z elementów automatycznie etykieta zostaje ustawiona na wartość aliasu tabeli. Jednak może też zaistnieć sytuacja, w której tabela również nie posiada etykiety, wtedy etykieta staje się nazwa tabeli. Przykład poniżej (Przykład 1.11) przedstawia zapytanie umożliwiające stworzenie nowego węzła z etykieta.

```
1  INSERT VERTEX TABLES (Person AS Person LABEL Person)
2  FROM MATCH (y)
3  WHERE y.name = 'Jane'
```

Przykład 1.11: Przykład klauzuli LABEL w PGQL

W przypadku aktualizacji danych składnia zapytania jest bardzo podobna do składni wykorzystanej w SQL. Główną różnicą jest to, że zamiast tabeli relacyjnej wykorzystujemy określony wzór, w celu wybrania potrzebnych zmiennych do modyfikacji. Poniższy przykład (Przykład 1.12) ukazuje, w jaki sposób można stworzyć zapytanie UPDATE.

```
1  UPDATE v SET (v.carOwner = true),
2      u SET (u.weight = 3500),
3      e SET (e.since = DATE '2010-01-03')
4  FROM MATCH (v:Person) <-[e:belongs_to]- (u:Car)
5  WHERE v.name = 'John'
```

Przykład 1.12: Przykład klauzuli UPDATE w PGQL

Usuwanie węzłów i krawędzi również jest bardzo podobne do usuwania w SQL. Polega ono głównie na określeniu wzorca, który nas interesuje i wykorzystanie go w klauzuli MATCH. Po czym z tego wzorca wybieramy elementy, które chcemy usunąć. Przykład poniżej (Przykład 1.13) przedstawia przykładowe zapytanie z wykorzystaniem klauzuli DELETE.

```
1  DELETE e
2  FROM MATCH () -[e]-> ()
3
4  DELETE x, y
5  FROM MATCH (x) -> (y)
```

Przykład 1.13: Przykład klauzuli DELETE w PGQL

Powyższy przykład przedstawia dwa zapytania z klauzulą DELETE. W pierwszym zapytaniu wybieramy wzorzec, w którym chcemy wyodrębnić wyłącznie określoną relację. Po



określeniu tej relacji jesteśmy w stanie usunąć ją poprzez podanie jej sygnatury za klauzulą DELETE. Działa to podobnie z węzłami, pokazuje to przykład drugi. Wybieramy wzorzec, który wyszukuje dwa węzły  $x$  i  $y$ , przy czym istnieje relacja skierowana z  $x$  na  $y$ . Podobnie jak w przypadku relacji, w celu usunięcia tych węzłów wystarczy podać ich sygnatury za słowem kluczowym DELETE.

### 1.4.3 G-CORE

G-CORE (Graph-Core) jest językiem zapytań pracującym na modelu grafowym. Umożliwia on wyrażanie różnorodnych wzorców i zależności między wierzchołkami i krawędziami. Charakterystycznym elementem tego języku jest semantyka typów. Pozwala ona na definiowanie typów dla wierzchołków i krawędzi, dzięki temu możemy tworzyć hierarchię typowania. Hierarchia typowania znacząco ułatwia precyzyjne określenie wzorców i ograniczeń wykorzystywanego modelu danych. Ważnym elementem G-CORE jest także skupienie się na wydajności. Język ten nie wspiera funkcji aktualizacji oraz usuwania definiowanych w CRUD.

Wszelkie zapytania w G-CORE zaczynają się od klauzuli CONSTRUCT. Wynika to z faktu, iż każde zapytanie musi zwracać finalnie graf. Graf ten może składać się przynajmniej z jednego węzła, co za tym idzie, nie musi posiadać relacji. W tej klauzuli podajemy wzór grafu, który ma zostać utworzony, dane te zostaną wypełnione po wykonaniu się zapytania.

W G-CORE tak jak w innych grafowych językach zapytań mamy także do czynienia z klauzulą MATCH. Odpowiada ona za określanie wzorców, które są szukane w grafie. Można podać kilka wzorców, oddzielając je znakiem przecinka (.). Poniższy przykład (Przykład 1.14) przedstawia wykorzystanie klauzuli MATCH z wykorzystaniem założonego z góry grafu *test\_graph*.

```
1  CONSTRUCT (p)
2    MATCH (p: Person)
3      ON test_graph
4      WHERE p.name = "John"
```

Przykład 1.14: Przykład klauzuli MATCH w G-CORE

Przykład ten tworzy w wyniku swojego działania nowy graf, który zawiera jeden węzeł. Węzłem tym jest wierzchołek z *test\_graph* którego właściwość “name” jest równa “John”. Cechą składni klauzuli MATCH w tym języku odróżniającą go od na przykład Cypher lub PGQL jest klauzula ON. Umożliwia ona wybranie grafu identyfikowanego nazwą. Występuje ona zaraz po określeniu wzorca w KLAUZULI MATCH. Klauzula WHERE działa bardzo podobnie jak w języku SQL. Wykorzystujemy w niej elementy znajdujące się w zdefiniowanym wzorcu klauzuli MATCH.

Klauzula GROUP umożliwia grupowanie wyników zapytania na podstawie kryteriów. Można ją wykorzystać we wzorcu podanym po klauzuli CONSTRUCTOR. Dzięki tej klauzuli mamy możliwość tworzenia logicznych grup w wynikach zapytań oraz umożliwia on wykonanie operacji agregacji na poszczególnych grupach. Agregację możemy dokonać na podstawie wybranych atrybutów oraz relacji. Wykorzystując ten mechanizm, możemy otrzymać bardziej skondensowane i przetworzone wyniki. Przykładowe wykorzystanie zostało przedstawione na poniższym przykładzie (Przykład 1.15).

```

1  CONSTRUCT test_graph ,
2      (x GROUP e: Person { birthPlace := e })
3      -[y: LIVES_IN]->
4      (n: City)
5  MATCH (n: City { name = e })

```

Przykład 1.15: Przykład klauzuli GROUP w G-CORE

Powyższy przykład tworzy nowy graf, który wybiera wyłącznie użytkowników mieszkających w mieście, w którym się urodzili.

Język G-CORE umożliwiające modyfikację grafu udostępniając dwie klauzule SET i REMOVE. Ważnym elementem jest fakt, że klauzule te nie powodują zmiany struktury w grafie źródłowym. Powodują one wyłącznie zmiany tymczasowe, które dokonywane są w grafie wynikowym zapytania. Klauzula SET umożliwia wprowadzenie zmian we właściwościach wierzchołków i krawędzi grafu wynikowego. Analogicznie klauzula REMOVE wprowadza możliwość do usuwania właściwości z grafu wynikowego.

Język ten umożliwia także łączenie wyników klauzuli CONSTRUCT z wykorzystaniem klauzuli UNION. Działają to na zasadzie sumowania lub łączenia grafów, stworzonych z wyników grafów.

Charakterystyczną cechą G-CORE jest zaawansowany mechanizm wyszukiwania dróg. Do tego mechanizmu należą takie Klauzule jak COST, SHORTEST, ALL oraz zmienna @p. Zaczynając od słowa kluczowego COST, jest to mechanizm umożliwiający przypisanie kosztu (długości, wagi) ścieżki do zmiennej. Przykład 1.16 przedstawia wykorzystanie słowa kluczowego COST.

```

1  CONSTRUCT (m)
2  MATCH (n: Person)-/p <:KNOWS*> COST c/->(m: Person)
3  WHERE n.name = "John"

```

Przykład 1.16: Przykład klauzuli COST w G-CORE

Klauzule SHORTEST i ALL oznaczają sposób przejścia między węzłami z jednego wierzchołka do drugiego. Pierwsza klauzula zakłada ograniczenie wyników do najszybszej możliwej trasy. Druga klauzula zakłada brak zwrócenie wszystkich możliwych tras w grafie, wraz z najkrótszymi i najdłuższymi. Przykłady 1.17 i 1.18 przedstawiają sposób wykorzystania obu tych klauzul.

```

1  CONSTRUCT (m)
2  MATCH (n: Person)-/3 SHORTEST p <:KNOWS*>/->(m: Person)
3  WHERE n.name = "John"

```

Przykład 1.17: Przykład klauzuli SHORTEST w G-CORE

Powyższy przykład znajduje trzy najkrótsze ścieżki znalezione w grafie z jednego wierzchołka do drugiego.

```

1  CONSTRUCT (m)
2  MATCH (n: Person)-/ALL p <:KNOWS*>/->(m: Person)
3  WHERE n.name = "John"

```

Przykład 1.18: Przykład klauzuli ALL w G-CORE

Powyższy przykład zwraca wszystkie możliwe ścieżki znalezione w grafie z jednego wierzchołka do drugiego.

### 1.4.4 GQL

GQL (Graph Query Language) jest propozycją globalnego standardu języka zapytań pracujących na modelu grafowym właściwości [1.3.2]. Projekt tego standardu został zaakceptowany w 2019 roku głosowaniem członków ISO/IEC Joint Technical Committee 1. Założeniami projektu jest stworzenie strukturalnego języka zapytań będącego odpowiednikiem języka SQL dla baz relacyjnych. Składnia GQL w dużej mierze przypomina składnię język Cypher [1.4.1].

### 1.4.5 Gremlin

Gremlin jest to język zapytań oparty o programowanie funkcyjne do zarządzania bazami danych opartych o model grafowy. Umożliwia on wykonywanie takich operacji jak modyfikowanie struktury grafu czy wykonywanie zaawansowanych zapytań. Język ten opiera się o strukturę “traversal”. Struktura ta umożliwia definiowanie sekwencji kroków, które należy wykonać na określonym grafie. Krokami w tej sekwencji może być przeszukiwanie, filtrowanie, sortowanie, łączenie, modyfikowanie i inne. Zapewnia to wsparcie do wykonywania skomplikowanych zapytań i manipulacji na danych. Wykorzystanie funkcyjnego podejścia w tym języku sprawia, że jest bardzo łatwy do zintegrowania z innymi rozwiązaniami.

Ważnym elementem implementacji tego języka zapytań jest trawersowanie. Trawersowanie możemy traktować jako przechodzenie po grafie. Proces ten polega na rozbiciu grafu na elementy oraz przypisanie mu odpowiednich operacji, jakie mają zostać na nim wykonane. Przypisanie odbywa się na zasadzie łączenia referencji elementu, jaki i operacji w obiekcie. Operację możemy traktować jako krok, który zostanie wykonany do momentu osiągnięcia założonych efektów. Każdy krok w procesie przejścia posiada podstawowe sześć danych. Pierwszą i drugą daną są odpowiednio pozycja w grafie w postaci dwójki uporządkowanej oraz pozycja w procesie trawersowania. Trzecim elementem jest oznaczona ścieżka (labeled path) będącą zbiorem łańcuchów znaków oraz obiektów. Pozwala ona na określenie, który element wskazuje, na który przykładowo  $((a, x), ((b, c), y), (, z))$  przedstawia ścieżkę trawersowania  $x \rightsquigarrow y \rightsquigarrow z$  ( $x$  wskazuje na  $y$ ,  $y$  wskazuje na  $z$ ). Każdy krok ścieżki posiada odpowiednią etykietę. Czwartą daną dla kroku jest jego waga. Piątą daną jest referencja do określonego trawersu. Ostatnią daną, czyli daną szóstą jest licznik pętli.

Język ten zakłada możliwość wykorzystanie kilkudziesięciu rodzajów kroków, które spełniają odpowiednią funkcjonalność. Podstawę stanowi jednak pięć podstawowych kroków, jakimi są:

- **map** — krok ten zakłada transformację z jednego typu na inny typ bez zmiany zbioru trawersowanego,
- **flatMap** — krok ten działa podobnie jak krok map, jednak umożliwia on utworzenie zbiorów wynikowych większy, równych lub mniejszych od początkowego zbioru trawersowanego,
- **filter** — krok ten umożliwia usunięcie niechcianych elementów ze zbioru trawersowanego,

- `sideEffect` — krok, który umożliwia wykonanie określonych operacji na zbiorze trawersowanym bez żadnej modyfikacji tegoż zbioru,
- `branch` — krok ten umożliwia rozdzielenie zbioru trawersowanego na określoną liczbę mniejszych zbiorów.

Z powyższymi elementami często możemy spotkać się w przypadku programowania reaktywnego. Większość bibliotek wprowadzająca elementy reaktywne również definiuje podobne metody.

W celu przybliżenia składni posłużymy się prostym przykładem. Poniższy przykład (Przykład 1.19) przedstawia proste zapytanie w języku Gremlin, którego zadaniem jest określenie wieku najstarszego znajomego Johna.

```

1  g.V()
2    .has("name","John")
3    .out("KNOWS")
4    .values("age")
5    .max()
```

Przykład 1.19: Prosty przykład zapytania w Gremlin

Podstawowym elementem w powyższym przykładzie jest element `g.V()`. Reprezentuje on kolejny wierzchołek, który będzie przetwarzany w naszym zapytaniu. Pierwszym krokiem, jaki wykonujemy na danym wierzchołku, jest sprawdzenie, czy dany wierzchołek ma odpowiednią właściwość oraz, czy ta właściwość ma określoną wartość (wykorzystanie metody `has()`). W przykładzie weryfikujemy czy wierzchołek odpowiada osobie o imieniu “John” poprzez właściwość “name”. Jest to krok będący krokiem filtrującym (filter), tak jak zostało to opisane wyżej. Kolejnym elementem jest przejście do sąsiedniego wierzchołka za pomocą metody `out()`. Należy zwrócić uwagę, że zakładamy fakt, iż relacja “KNOWS” jest relacją kierunkową wychodzącą z aktualnie przetwarzanego węzła. Możemy również wykorzystać inne typy krawędzi wykorzystując metody `in()` oraz `both()`. Ten krok jest krokiem mapującym ze zmianą grafu początkowego. Oznacza to, że w wyniku tej operacji, w grafie wynikowym będą znajdować się wyłącznie elementy posiadające relację przychodzącą z grafu początkowego. Kolejny krok wykonany na grafie stworzonym po wywołaniu metody `out()` jest metoda `values()`. Transformuje ona wszystkie wierzchołki na wartość podanej właściwości, w przykładzie jest to właściwość “age”. Dzięki temu możemy przejść do analizy potrzebnych nam danych. Ostatnim krokiem, jaki wykonamy jest wybranie, największej wartości z danych (wykorzystanie metody `()`), które wyłuskaliśmy z grafu początkowego.

## 1.4.6 SPARQL

SPARQL (SPARQL Protocol And RDF Query Language) jest językiem zapytań stosowanym w Semantic Web. Został stworzony do pracy na modelu trójek RDF [1.3.3], będący standardem W3C i Semantic Web. W roku 2008 sam SPARQL stał się trzecim podstawowym standardem Semantic Web zaraz obok wyżej wymienionego modelu trójek RDF oraz OWL (W3C Web Ontology Language).

Składnią język ten przypomina język zapytań SQL. Podstawowymi klauzulami wykorzystywanymi do czytania danych są `SELECT`, `CONSTRUCT`, `DESCRIBE` oraz `ASK`. Oprócz

tego język ten umożliwia tworzenie, usuwanie i aktualizowanie. Do wykonania tych operacji wykorzystujemy takie klauzule jak INSERT DATA, DELETE DATA, LOAD. Język ten pozwala także na definiowanie zmiennych. Nazwy zaczynające się od znaku “?” oznaczają deklarację zmiennej o nazwie podanej zaraz za tym znakiem. Przykład 1.21 ukazuje sposób wykorzystania zmiennych.

Do przedstawienia przykładów składni zapytań wykorzystamy trójki RDF przedstawione poniżej (Przykład 1.20).

```
1  :id1 person:name "John"
2  :id1 person:based_near :Austin
3  :id2 person:name "Adam"
4  :id2 person:based_near :Dallas
5  :id3 person:name "Anna"
6  :id3 person:based_near :NewYork
```

Przykład 1.20: Przykładowa definicja trójek RDF wykorzystana w przykładach SPARQL

Zacznijmy od przedstawienia przykładu składni SELECT. Przykład poniżej (Przykład 1.21) przedstawia przykładowe zapytanie, w którym wybieramy imiona osób. Zapytanie zwraca zmienne “?person”, “?name” i “?based\_near”. Dzięki klauzuli SELECT jesteśmy w stanie wyszukiwać trójki RDF. W przypadku tego przykładu wyszukujemy wszystkie trójki, w których występują właściwości “person:name” i “person:based\_near”. Zmienna “?person” przechowuje wartość identyfikatora osoby, natomiast “?name” i “?location” przechowują odpowiednio wartości pól “person:name” i “person:based\_near”. Wynikiem tego zapytania będzie tabela zawierająca wszystko podane zmienne. Warto również wspomnieć o tym, że znak “;” oddziela kolejne wartości w wierszu bloku inicjalizacyjnego, natomiast znak “.” kończy aktualnie definiowany wiersz. Przyjmując, że założenie jest to zbiór warunków.

```
1  PREFIX person: <https://example.org/person/>
2
3  SELECT ?person ?name ?location
4  WHERE {
5      ?person person:name ?name ;
6          person:based_near ?location .
7  }
```

Przykład 1.21: Przykład klauzuli SELECT w SPARQL

W kolejnym przykładzie (Przykład 1.22) konstruujemy nowy graf RDF, który zawiera tylko informacje o nazwie (person:name) i lokalizacji (person:based\_near) osób. Wykorzystujemy do tego klauzulę CONSTRUCT. Umożliwia to nam skonstruowanie nowego grafu RDF, który zawiera tylko interesujące nas informacje.

```
1  CONSTRUCT {
2      ?person person:name ?name ;
3          person:based_near ?location .
4  }
5  WHERE {
6      ?person person:name ?name ;
7          person:based_near ?location .
8  }
```

Przykład 1.22: Przykład klauzuli CONSTRUCT w SPARQL

Przykład poniżej (Przykład 1.23) przedstawia wykorzystanie klauzuli DESCRIBE. Dzięki niej jesteśmy w stanie wyświetlić dokładne informacje dotyczące określonej trójki RDF. W tym zapytaniu wykorzystujemy identyfikator danej trójki, w przypadku przykładu jest to “:id1”.

```
1  DESCRIBE :id1
```

#### Przykład 1.23: Przykład klauzuli DESCRIBE w SPARQL

Następny przykład (Przykład 1.24) przedstawia wykorzystanie klauzuli ASK. Jest ona wykorzystywana w celu sprawdzenia, czy istnieje trójka RDF, która spełnia określone warunki. Kiedy istnieje taka trójka, zapytanie zwraca wartość “true” w innym przypadku otrzymujemy “false”. Omawiany przykład przedstawia zapytanie, w którym sprawdzamy, czy istnieje trójka RDF, której wartość pola “name” jest równa “John” oraz której wartość pola “based\_near” jest równa “:Austin”.

```
1  ASK
2  WHERE {
3      ?person person:name "John" ;
4              person:based_near :Austin .
5  }
```

#### Przykład 1.24: Przykład klauzuli ASK w SPARQL

W przykładzie poniżej (Przykład 1.25) wykorzystujemy klauzulę INSERT DATA do wprowadzania nowych wartości do grafu. Przykład wprowadza trzy nowe trójki reprezentujące osoby o identyfikatorach “id1”, “id2” i “id3”. Każda ta reprezentacja posiada dwie właściwości “name” (imię) i “based\_near” (miejsce zamieszkania). Po wprowadzeniu tych danych możemy wykorzystać ich wartość, odnosząc się w następujący sposób do właściwości “person:name”.

```

1  INSERT DATA {
2      :id1 person:name "John" ;
3          person:based_near :Austin .
4
5      :id2 person:name "Adam" ;
6          person:based_near :Dallas .
7
8      :id3 person:name "Anna" ;
9          person:based_near :NewYork .
10 }

```

Przykład 1.25: Przykład klauzuli INSERT DATA w SPARQL

Powyższy przykład omawia wprowadzanie. Przyjrzyjmy się teraz usuwaniu wartości, możemy to osiągnąć z wykorzystaniem klauzuli DELETE DATA. Klauzula ta działa na zasadzie dopasowania wzorca. Jeżeli podamy dokładne predykaty, wtedy zostanie usunięte tylko trójka spełniająca je. W przypadku gdy predykaty nie wskazują na jedną trójkę, wszystkie wskazane trójki zostaną usunięte. Poniższy przykład (Przykład 1.26) przedstawia przykładowe zapytanie. Zapytanie to usuwa trójkę, w której podmiotem jest “:id1” oraz są spełnione predykaty dotyczące wartości właściwości “name” i “based\_near”.

```

1  DELETE DATA
2  {
3      :id1 person:name "John" ;
4          person:based_near :Austin .
5  }

```

Przykład 1.26: Przykład klauzuli DELETE DATA w SPARQL

SPARQL dostarcza także możliwość ładowania danych z plików. Wykorzystuje się do tego klauzulę LOAD. Poniższy przykład (Przykład 1.27) przedstawia podstawowe wykorzystanie tej klauzuli. W tym przykładzie wykorzystujemy plik zapisany na dysku lokalnym. Możemy także wykorzystywać pliki umieszczone zdalnie poprzez adres URI. Dane z tego pliku są automatycznie wczytywane do bazy danych.

```

1  LOAD <file:/sciezka/do/pliku.rdf>

```

Przykład 1.27: Przykład klauzuli LOAD w SPARQL

Ważnym elementem jest aktualizowanie wartości w trójkach. W przypadku SPARQL do aktualizacji danych wykorzystujemy kombinację trzech klauzul, są to kolejno klauzula DELETE, INSERT oraz WHERE. Poniższy przykład (Przykład 1.28) reprezentuje przykładowe zapytanie wykorzystujące te klauzule. Przykład ten przedstawia kroki, które ma wykonać SZGBD (System Zarządzania Grafową Bazą Danych). Pierwszym elementem jest usunięcie wartości “:Dallas” dla właściwości “person:based\_near”. Zmienna “?person” przechowuje informacje o identyfikatorze trójki, której wartość zostanie usunięta. Kolejnym elementem jest wprowadzenie nowej wartości do tej właściwości, to jest “:Houston”. Ostatnim elementem jest wprowadzenie predykatów, które muszą być spełnione, aby dana trójka została zmieniona.

```

1  DELETE {
2      ?person person:based_near :Dallas .
3  }
4  INSERT {

```

```

5      ?person person:based_near :Houston .
6  }
7  WHERE {
8      ?person person:name "Adam" .
9      ?person person:based_near :Dallas .
10 }

```

Przykład 1.28: Przykład aktualizowania wartości w SPARQL

## 1.4.7 GSQL

GSQL (Graph Structured Query Language) jest językiem zapytań, który rozszerza funkcjonalności języka SQL o możliwość wykorzystania funkcjonalności, związanymi z grafami. Został on stworzony do wykorzystania w TigerGraph. Omówienie składni tego język zaczniemy od elementów umożliwiających definiowanie danych. Poniższy przykład (Przykład 1.29) przedstawia zbiór operacji, które definiują nową strukturę grafu.

```

1  CREATE VERTEX person (PRIMARY_ID id STRING, name STRING, age INT)
2
3  CREATE VERTEX city (PRIMARY_ID id STRING, name STRING)
4
5  CREATE DIRECTED_EDGE lives_in (FROM person TO city, since DATE)
6
7  CREATE UNDIRECTED_EDGE knows (FROM person TO person)
8
9  CREATE GRAPH city_graph (
10     person VERTEX,
11     city VERTEX,
12     lives_in DIRECTED_EDGE,
13     knows UNDIRECTED_EDGE
14 )

```

Przykład 1.29: Przykład tworzenia nowej struktury w GSQL

Pierwszym etapem tego przykładu jest stworzenie dwóch węzłów, pierwszym jest “person”, natomiast drugim jest “city”. Węzeł “person” będzie posiadał trzy właściwości. Pierwsza właściwość będzie reprezentowała unikalny identyfikator węzła w postaci ciągu znaków. Kolejna właściwość węzła “person” będzie reprezentowała imię osoby w postaci ciągu znaków. Ostatnią właściwością węzła jest pole “age” oznaczające wiek opisywanej osoby, ta wartość będzie reprezentowana przez wartość numeryczną. W przypadku węzła “city” definiujemy dwie właściwości. Pierwsza właściwość reprezentuje identyfikator jako ciąg znaków. Druga właściwość reprezentuje nazwę miasta w postaci ciągu znaków.

Kolejnym etapem jest tworzenie relacji między węzłami. W przykładzie tworzymy dwie relacje, jedna jest relacją skierowaną, druga natomiast jest relacją nieskierowaną. Na przedstawionym przykładzie widoczne jest, w jak łatwy sposób możemy to osiągnąć. W przypadku relacji nieskierowanej tworzymy relację “lives\_in”. Kolejność podania węzłów w tym przypadku ma znaczenie, kierunek tej relacji będzie z węzła “person” do węzła “city”. W przypadku relacji nieskierowanej kolejność podania węzłów nie ma znaczenie. Warto zaznaczyć, że dla relacji “lives\_in” definiujemy także właściwość “since” w formacie daty. Właściwość ta jak sama nazwa wskazuje, reprezentuje rok wprowadzenia się do miasta. Dalej tworzymy



relację “knows”, w tym przypadku jest to relacja nieskierowana z węzła “person” na ten sam węzeł, czyli “person”. Oznacza to, że ta relacja reprezentuje stosunek danego węzła między tym samym typem węzła.

Ostatnim elementem jest stworzenie grafu. Wykorzystujemy do tego klauzulę “CREATE GRAPH”. W przykładzie tworzymy graf o nazwie “city\_graph” dodając do niego wcześniej zdefiniowane typy węzłów oraz relacji między nimi.

Przykład poniżej (Przykład 1.30) przedstawia dwa przykładowe zapytania SELECT.

```
1  SELECT *
2  FROM city_graph
3  LIMIT 10;
4
5  SELECT person.name, city.name
6  FROM city_graph
7  WHERE lives_in IS NOT NULL;
```

Przykład 1.30: Przykład prostych zapytań SELECT w GSQL

W powyższym przykładzie rozpatrujemy dwa zapytania. Pierwsze zapytanie zwraca próbkę danych o liczebności dziesięć. Drugi przykład zwraca wszystkie imiona i nazwy miast dla węzłów które posiadają relację “lives\_in” między sobą. Zapytanie to jest dosyć proste jednak pokazuje jak bardzo zbliżona jest składnia GSQL do SQL.

Przytoczmy teraz dodatkowy graf, który będzie wykorzystywał już zdefiniowane węzły z poprzedniego przykładu definicji. Poniższy przykład (Przykład 1.31) przedstawia tworzenie grafu “family\_graph”.

```
1  CREATE UNDIRECTED EDGE siblings (FROM person TO person)
2
3  CREATE GRAPH famili_graph (
4    person VERTEX,
5    siblings UNDIRECTED_EDGE
6  )
```

Przykład 1.31: Przykład tworzenia grafu “family\_graph” w GSQL

Graf “family\_graph” jest kluczowy do pokazania łączenia grafów w zapytaniach SELECT. Rozpatrzmy teraz przykład (Przykład 1.32) wybrania imion osób, które nie mieszkają w tym samym mieście co ich rodzeństwo.

```
1  SELECT p.name
2  FROM family_graph AS person: sib1 -(siblings)- person: sib2,
3       city_graph AS person: p1 -(lives_in>)- city: c1
4  WHERE sib1.name = p1.name
5  AND (
6  SELECT COUNT(x1.name)
7  FROM person: x1 -(lives_in>)- city: y1
8  WHERE x1.name = sib2.name
9  AND y1.name != c1.name
10 ) != 0;
```

Przykład 1.32: Przykład zapytania SELECT z wykorzystaniem łączenia grafów w GSQL

Powyższy przykład przedstawia przede wszystkim zastosowanie wzorców grafowych, za pomocą których wyszukujemy węzłów o określonych relacjach. Zawiera on także zastoso-

wanie zapytania zagnieżdżonego. Efektem tego został przedstawiony przykład “relatywnie” zaawansowanego zapytania.

Pozostałe składnie zapytań są analogiczne do języka zapytań SQL.

## 1.5 Przegląd grafowych baz danych

Sekcja ta będzie skupiała się na omówieniu popularnych systemów baz danych, które przetwarzają i zapisują dane w formie grafów. Skupimy się na najważniejszych cechach tego rodzaju systemu oraz na porównaniu podstawowych cech charakteryzujących te bazy danych. Omówione zostaną najważniejsze elementy, na które należy zwrócić uwagę przy wyborze odpowiedniego systemu. Pierwszym aspektem, który będziemy rozważać, jest wykorzystywany model danych. Aspekt ten będzie omawiał możliwe modele, które są obsługiwane przez dany system. Rozważać będziemy następujące modele: model grafu właściwości, hiper grafy, trójki RDF. Drugim aspektem, któremu się przyjrzymy, są obsługiwane języki baz danych służące do manipulowania, czytania i definiowania danych. Rozważać będziemy następujące języki: Gremlin, G-CORE, PGQL, Cypher, SPARQL. Trzeci aspekt będzie skupiał się na obsługiwanych językach programowania. Jest to ważny aspekt pod kątem integracji z innymi narzędziami. Będziemy rozpatrywać wsparcie następujących języków programowania: JavaScript, Java, C#, C++, Python, Ruby, PHP, Go.

Amazon Neptune jest usługą bazodanową udostępnianą przez Amazon Web Services (AWS). Jest to rozwiązanie przystosowane pod cloud computing wraz z innymi usługami AWS. Pierwszy raz została udostępniona w roku 2017. Środowisko tej bazy danych jest w pełni obsługiwane przez Amazon. Oznacza to, że dystrybucja, utrzymanie, monitorowanie infrastruktury, konwersja i skalowanie leży po stronie dostawcy. Ważnym aspektem jest pełna integracja Neptune z innymi usługami AWS, ułatwia to wdrożenie i zarządzanie bazy na podstawie rozwiązań chmurowych. Baza ta opiera się o model grafu właściwości [1.3.2] oraz o trójki RDF [1.3.3]. Obsługuje ona dwa języki zapytania, pierwszym językiem jest Gremlin [1.4.5], drugim jest SPARQL [1.4.6] (Tabela 1.3). Dostarcza ona także integrację z większością rozważanych języków programowania (Tabela 1.4). Baza danych Amazon Neptune jest publikowana na licencji AWS. AWS oferuje różne modele licencjonowania dla swoich usług, w tym elastyczną strukturę opłat na podstawie wykorzystanie i abonamenty.

Neo4J został opracowany, po czym opublikowany w 2007 roku przez szwedzką firmę Neo4J Inc. Od tego czasu widzimy nieprzerwany rozwój tego rozwiązania. Jest to jedna z pierwszych baz danych oparta o model grafowy. Charakteryzuje się ona skalowalnością i wydajnością, jest w stanie obsługiwać zarówno małe, jak i rozległe grafy. W pełni obsługuje ona transakcje ACID, dzięki czemu użytkownicy mogą być pewni integralności danych. Baza ta oferuje także wysoką dostępność poprzez takie mechanizmy jak replikacja danych i zautomatyzowane klajstrowanie danych. Dzięki tym mechanizmom system ten może być uruchomiony na architekturze wielowęzłowej, co zapewnia działanie systemu mimo awarii jednego z serwerów. Baza danych Neo4J wykorzystuje język zapytań Cypher [1.4.1] oraz umożliwia poprzez wykorzystanie odpowiednich rozszerzeń obsługę języka Gremlin [1.4.5] (Tabela 1.3). Posiada ona także obsługę wszystkich języków rozpatrywanych w tym porównaniu (Tabela 1.4). Podstawowa wersja Neo4j (Neo4j Community Edition) jest udostępniona na licencji ogólnej GNU General Public License version 3 (GPLv3). Jest to wersja open-

source, dzięki czemu jesteśmy w stanie wykorzystywać tę bazę danych bez koniecznych opłat. Ogranicza się to jednak do podstawowej wersji narzędzia. Wersja rozszerzona Neo4J (Neo4j Enterprise Edition) jest dostępna na podstawie licencji komercyjnej. Jest to wersja płatna, jednak oferująca dodatkowe narzędzia i wsparcie techniczne. Rozwiązanie to jest płatne, przeznaczone dla komercyjnego wykorzystania w przedsiębiorstwach i większych projektach publicznych.

RedisGraph jest bazą danych grafową, która została stworzona przez firmę Redis Labs. Została opublikowana po raz pierwszy w roku 2017 podobnie jak Amazon Neptune. Baza ta jest oparta o silnik bazodanowy Redis, który to jest systemem opartym o model klucz-wartość. Łączy ona możliwości modelowania danych w postaci grafu oraz wydajność modelu bazodanowego klucz-wartość. System ten obsługuje język zapytań Cypher [1.4.1] (Tabela 1.3). Posiada on także interfejs komunikacyjny ze sporą częścią rozważanych języków zapytań (Tabela 1.4). Baza danych RedisGraph jest publikowana na licencji Affero General Public License (AGPL), co oznacza, że jest to wolne oprogramowanie, które umożliwia użytkownikom używanie, modyfikowanie i rozpowszechnianie jej kodu źródłowego.

JanusGraph jest to system bazodanowy zorientowany na grafy, który został opracowany przez JanusGraph Community. Jest rozwijany jako projekt open-source przez społeczność programistów. System ten został opublikowany w tym samym roku co RedisGraph i Amazon Neptune, to jest w roku 2017. Główną cechą tej bazy jest koncentracja na obsłudze rozproszonych i rozległych grafów. Zapewnia ona więc wysoką skalowalność, wydajność oraz elastyczność. W tej bazie danych możemy wykorzystać wiele silników przechowywania takich jak Apache Cassandra, Apache HBase, czy Oracle BerkeleyDB. Baza ta obsługuje język zapytań Gremlin [1.4.5] oraz język zapytań SPARQL [1.4.6] (Tabela 1.3). Posiada ona interfejs komunikacyjny dla takich języków jak: JavaScript, Java, Python, Ruby, PHP i Go (Tabela 1.4). System bazodanowy JanusGraph jest publikowany w oparciu o licencję Apache License 2.0.

TigerGraph jest to system bazodanowy zorientowany na model grafowy. Został opublikowany w 2016 roku przez firmę TigerGraph Inc. Najważniejszą cechą tego systemu bazodanowego jest koncentracja na analizie danych zamieszczonych w grafie. Można w nim przechowywać modele danych o bardzo złożonej strukturze relacji. Posiada on możliwość przetwarzania zapytań w czasie rzeczywistym. Baza ta wykorzystuje model grafu właściwości do przechowywania danych. System ten wykorzystuje język zapytań GSQL [1.4.7] (Tabela 1.3). Udostępnia on interfejs komunikacyjny z takimi językami jak Java, C++, Python i Go (Tabela 1.4). Kwestia licencjonowania jest złożone w przypadku tej bazy danych. Posiada ona trzy rodzaje licencji: Developer Edition, Enterprise Edition oraz wersję licencji Cloud. Wszystkie z wymienionych licencji są płatne, jednak można przetestować to rozwiązanie za darmo przez określony czas.

ArangoDB jest wielomodelowym systemem baz danych, który łączy w sobie cechy baz danych grafowych, dokumentowych i klucz-wartość. Została opracowana przez niemiecką firmę ArangoDB GmbH oraz opublikowana przez tą firmę w 2012 roku. Od tego czasu ArangoDB zdobyła pewną popularność oraz wiele usprawnień, stała się również częściowo projektem open-source. Główną cechą tego systemu bazodanowego jest wielomodelowość, co zapewnia ogromną elastyczność zastosowania. Modelem grafowym wykorzystywanym w tym systemie jest model grafu właściwości. Model ten obsługuje takie języki zapytań jak SQL, AQL (ArangoDB Query Language) oraz Gremlin [1.4.5] (Tabela 1.3). Warto zwrócić

tutaj uwagę na język AQL, który nie został opisany w pracy. Jest to język zapytań, który umożliwia wykorzystanie funkcjonalności CRUD na wielu modelach. Omawiany system bazodanowy dostarcza interfejsy komunikacyjne dla takich języków programowania jak: JavaScript, Java, C#, C++, Python, Ruby, PHP i Go (Tabela 1.4). ArangoDB jest udostępniany w dwóch wersjach, to jest Community Edition oraz Enterprise Edition. Wersja Community Edition jest udostępniana na zasadach licencji Apache 2.0, dostarcza ona podstawową funkcjonalność bazy danych. Wersja Enterprise Edition jest udostępniana na licencji płatnej, rozszerza ona podstawową wersję bazy danych oraz dostarcza dodatkowe narzędzia.

OrientDB został stworzony przez włoskiego programistę Lucę Garulliego w 2010 roku. Aktualnie system ten jest rozwijany przez firmę OrientDB Ltd. Podobnie jak ArangoDB jest on systemem bazodanowym opartym o wiele modeli. Rodzajem modelu grafowego wykorzystywanego w OrientDB jest model grafu właściwości. System ten udostępnia wiele mechanizmów indeksowania, takich jak indeksowanie B-drzewiaste, hash indeksy i indeksy pełnotekstowe. Umożliwia on obsługę takich języków zapytania jak SQL, Gremlin [1.4.5] oraz Cypher [1.4.1] (Tabela 1.3). Interfejsy komunikacyjne tej bazy są dostępne dla następujących rozważanych języków programowania: JavaScript, Java, C#, Python, Ruby, PHP, i Go (Tabela 1.4). OrientDB jest publikowany na licencji Apache License 2.0.

<b>Baza danych</b>	<b>Gremlin</b>	<b>Cypher</b>	<b>G-CORE</b>	<b>SPARQL</b>	<b>PGQL</b>	<b>GSQL</b>
Neo4J	•	•				
RedisGraph		•				
JanusGraph	•			•		
TigerGraph						•
ArangoDB	•					
OrientDB	•	•				
Amazon Neptune	•			•		

Tabela 1.3: Obsługiwane języki zapytań przez bazy danych

<b>Baza danych</b>	<b>JavaScript</b>	<b>Java</b>	<b>C#</b>	<b>C++</b>	<b>Python</b>	<b>Ruby</b>	<b>PHP</b>	<b>Go</b>
Neo4J	•	•	•	•	•	•	•	•
RedisGraph	•	•	•		•	•		•
JanusGraph	•	•			•	•	•	•
TigerGraph		•		•	•			•
ArangoDB	•	•	•	•	•	•	•	•
OrientDB	•	•	•		•	•	•	•
Amazon Neptune	•	•	•		•			

Tabela 1.4: Obsługiwane języki programowania przez bazy danych

## Rozdział 2

# Teoria języków formalnych i definiowanie gramatyk

Omawiane w tym rozdziale języki formalne [2.1] stanowią podstawę do wielu dziedzin informatyki i matematyki. Stanowią one system symboliczny zdefiniowany poprzez określone reguły składni oraz semantyki. Języki formalne definiowane są w sposób precyzyjnie określony, bez przestrzeni na wieloznaczność, w odróżnieniu od języków naturalnych [2.2].

### **Definicja 2.1. Język formalny**

*Język formalny  $L$  nad alfabetem  $\Sigma$  jest dowolnym podzbiorem  $\Sigma^*$ , gdzie  $\Sigma^*$  jest zbiorem wszystkich możliwych słów utworzonych z symboli  $\Sigma$ .*

### **Definicja 2.2. Język naturalny**

*Język naturalny to system komunikacji rozwijany przez ludzi w sposób naturalny, nieświadomy i nieplanowany, zawierający unikalny zestaw reguł semantycznych, syntaktycznych, fonetycznych i pragmatycznych. Ich struktury są często niejednoznaczne i złożone, co utrudnia tworzenie ich pełnych i precyzyjnych definicji formalnych.*

W skład języków formalnych wchodzi takie elementy jak symbole, słowa i zdania. Symbol w takich językach stanowi podstawową jednostką informacji, słowami natomiast są zbiory symboli stanowiącą określoną informację. Zdanie natomiast nie jest zwyczajnym zbiorem słów, są one tworzone na podstawie określonych reguł. Reguły te są definiowane przez określony język formalny.

Języki formalne stanowią bardzo ważny element między innymi w programowaniu. Każdy język programowania pod spodem ma zadeklarowany język formalny, który posiada określone symbole, słowa i zdania. Języki formalne mogą mieć znacząco inne reguły, możemy to zauważyć, porównując na przykład różne języki programowania. Oprócz zastosowania w kompilatorach języków programowania języki te mogą być wykorzystane w wielu innych aspektach naukowych.

Rozdział ten będzie przybliżał zagadnienia powiązane z teorią języków formalnych. Omówimy różne typy języków formalnych oraz związane z nimi gramatyki. Zostanie przedstawiona także struktura języków formalnych oraz omówienie relacji między automatami a językami formalnymi. Finalnie przyjrzymy się popularnym narzędziom dostępnym na rynku umożliwiającym definiowanie języków formalnych takich jak BNF, EBNF, ABNF oraz ANTLR, oraz przedstawimy porównanie składni tych narzędzi.

## 2.1 Typy języków formalnych

Ważnym aspektem w przypadku języków formalnych jest przedstawienie ich typów. Wyodróżniamy cztery typy języków formalnych, to jest języki regularne, języki bezkontekstowe, języki kontekstowe, języki rekurencyjnie przeliczalne. Każdy z tych typów ma swoje określone cechy.

W tej sekcji postaramy się omówić najważniejsze elementy każdego z tych typów.

### 2.1.1 Języki regularne

Języki regularne [2.3] są najprostszym rodzajem języków formalnych. Definiowane są one za pomocą operatorów unii, konkatenacji oraz domknięcia Kleene’a [2.4]. Ten typ języków jest determinowany i rozpoznawany przez automaty skończone i opisywany za pośrednictwem wyrażeń regularnych [2.7]. Istnieją dwa typy automatów skończonych, to jest automaty deterministyczne [2.5] i niedeterministyczne [2.6].

#### **Definicja 2.3. Język regularny**

*Formalnie, język jest regularny, jeśli:*

1. Pusty zbiór  $\emptyset$  jest językiem regularnym.
2. Dla każdego symbolu  $a$  w alfabecie  $\Sigma$ ,  $\{a\}$  jest językiem regularnym.
3. Jeśli  $A$  i  $B$  są językami regularnymi, to:
  - $A \cup B$  (unia  $A$  i  $B$ ) jest językiem regularnym.
  - $AB$  (konkatenacja  $A$  i  $B$ ) jest językiem regularnym.
  - $A^*$  (domknięcie Kleene’a  $A$ ) jest językiem regularnym.
4. Żaden inny język nie jest językiem regularnym.

#### **Definicja 2.4. Domknięcie Kleene’a**

*Dla dowolnego języka  $L$  nad danym alfabetem, domknięcie Kleene’a  $L^*$  tego języka jest zdefiniowane następująco:*

1. Jeżeli  $L$  jest językiem,  $\epsilon$  (słowo puste) należy do  $L^*$ .
2. Jeżeli  $w$  należy do  $L$  i  $x$  należy do  $L^*$ , to  $wx$  (konkatenacja  $w$  i  $x$ ) należy do  $L^*$ .

W przypadku języków regularnych ważnym elementem na, który trzeba zwrócić uwagę, jest pojęcie automatu stanu. Automat skończony, inaczej znany także jako maszyna stanów, jest to model obliczeniowy, który może istnieć w jednym z określonej liczby stanów.

Definiujemy go jako piątkę  $(Q, \Sigma, \delta, q_0, F)$ , gdzie:

- $Q$  - to skończony zbiór stanów,
- $\Sigma$  - to skończony zbiór symboli wejściowych (alfabet),

- $\delta$  - to funkcja przejścia, która mapuje pary stan-symbol na stany,
- $q_0$  - to stan początkowy, który należy do  $Q$ ,
- $F$  - to zbiór stanów końcowych (akceptujących), który jest podzbiorem  $Q$ .

Oprócz definicji ogólnej automatu wyróżniamy dwa rodzaje automatu, to jest automat deterministyczny [2.5] i automat niedeterministyczny [2.6].

Zacznijmy od omówienia pierwszego z rodzajów, którym jest automat deterministyczny. Automat deterministyczny w literaturze może być określany także jako deterministyczny automat skończony (DFA — Deterministic Finite Automaton). Jest to model obliczeń, który składa się z zestawu stanów i funkcji przejścia. Funkcja przejścia mapuje pary składające się ze stanu i wejściowego symbolu do kolejnego stanu. Słowo “deterministyczny” w nazwie automatu oznacza, że dla danego stanu i danego wejścia, zawsze jest dokładnie jeden stan, do którego można przejść.

### **Definicja 2.5. Automat deterministyczny**

*Deterministyczny automat skończony (DFA) to krotka  $M = (Q, \Sigma, \delta, q_0, F)$ , gdzie:*

- $Q$  jest skończonym zbiorem stanów,
- $\Sigma$  jest skończonym zbiorem symboli, zwanym alfabetem,
- $\delta : Q \times \Sigma \rightarrow Q$  jest funkcją przejścia,
- $q_0 \in Q$  jest stanem początkowym,
- $F \subseteq Q$  jest zbiorem stanów akceptujących.

Drugim typem automatu stanu jest automat niedeterministyczny. W literaturze automat niedeterministyczny można także spotkać pod nazwą niedeterministyczny automat skończony (NFA — Nondeterministic Finite Automaton). Jest to rodzaj automatu skończonego, w którym dla danego stanu i danego symbolu wejściowego może istnieć wiele możliwych następnych stanów.

### **Definicja 2.6. Automat niedeterministyczny**

*Automat niedeterministyczny (NFA) to piątka  $(Q, \Sigma, \delta, q_0, F)$ , gdzie:*

- $Q$  jest skończonym zbiorem stanów,
- $\Sigma$  jest skończonym zbiorem symboli wejściowych (alfabetem),
- $\delta$  jest funkcją przejścia, tzn.  $\Delta : Q \times \Sigma \rightarrow 2^Q$ ,
- $q_0 \in Q$  jest stanem początkowym,
- $F \subseteq Q$  jest zbiorem stanów akceptujących.



Ważnym elementem, o którym należy wspomnieć w kwestii języków regularnych, są wyrażenia regularne [2.7]. Wyrażenia regularne są stosowane jako narzędzie umożliwiające opisywanie języków regularnych. Są one ściśle powiązane z automatami skończonymi i umożliwiają definiowanie i manipulowanie zbiorami słów nad danym alfabetem.

**Definicja 2.7. Wyrażenie regularne**

*Wyrażenie regularne nad alfabetem  $\Sigma$  definiujemy indukcyjnie:*

1.  $\emptyset$ ,  $\varepsilon$  oraz każde  $a \in \Sigma$  są wyrażeniami regularnymi.
2. Jeżeli  $R$  i  $S$  są wyrażeniami regularnymi, to  $(R + S)$ ,  $(RS)$  oraz  $(R^*)$  również są wyrażeniami regularnymi.
3. Żadne inne wyrażenia nie są wyrażeniami regularnymi.

Rozpatrzmy przykład języka regularnego, w którym rozpatrujemy wszystkie słowa nad alfabetem  $\{a, b\}$ , które zawierają, dokładnie trzy litery  $a$ . Język ten możemy opisać za pomocą następującego wyrażenia regularnego:  $(b^*a)\{3\}$ . Rozpatrując ten język, możemy powiedzieć, że następujące słowa nie należą do języka “aa”, “a”, “aba”. Wynika to z faktu, że nie jest spełniony warunek konieczny, czyli nie występują, przynajmniej trzy litery  $a$ . W przypadku takich słów jak “aaa” czy “ababa” możemy powiedzieć, że należą one do języka.

Przyjrzyjmy się dokładnie sprawdzeniu, czy słowo należy do języka poprzez sprawdzenie słowa “aabbba”. Możemy podzielić to słowo na trzy części, które odpowiadają temu wzorowi:

- Pierwsza część to “a”, co odpowiada wzorowi  $b^*a$ , gdzie  $b^*$  odpowiada zero wystąpień “b”.
- Druga część to “ab”, gdzie  $b^*$  odpowiada jednemu wystąpieniu “b”.
- Trzecia część to “bba”, gdzie  $b^*$  odpowiada dwóm wystąpieniom “b”.

Zatem słowo “aabbba” pasuje do wzorca zdefiniowanego w przykładzie powyżej, co oznacza, że należy ono do języka regularnego opisanego tym wyrażeniem.

## 2.1.2 Języki bezkontekstowe

Język bezkontekstowy to język generowany przez bezkontekstową gramatykę formalną [2.8]. Wykorzystywane są w różnych aspektach informatyki jak analiza składniowa, analiza i projektowanie protokołów komunikacyjnych czy bazy danych. Szczególnie interesuje nas analiza składniowa, ponieważ języki tego typu często są wykorzystywane w kompilatorach i interpretatorach. Definiują one języki programowania i z ich wykorzystaniem możliwe jest utworzenie odpowiedniej składni, analiza tej składni oraz budowanie tak zwanego drzewa parsowania. Wszystkie te elementy tworzone są z poziomu parsera i wykorzystywane do interpretacji kodu lub transformacji kodu wysokopoziomowego na kod maszynowy.

Najważniejszym elementem tego języka jest gramatyka bezkontekstowa. W skład tej gramatyki wchodzi cztery elementy takie jak symbol nieterminalny [2.10], symbol terminalny

[2.9], produkcja i symbol startowy. Symbole nieterminalne i terminalne odpowiadają za reprezentację jednostek wejściowych do produkcji. Stanowią one podstawową informację o przejściu interpretacji języka. W przypadku symbolu nieterminalnego mówimy o formie nieustalonej gramatyki, stanowią one reprezentację strukturalnego elementu języka. Symbol terminalny można traktować rodzaju stałą lub jednostkę, która nie może być interpretowana na inny sposób, niż określa ją wartość. Innymi słowy, symbol terminalny to elementy języka, które nie mogą być dalej rozwinięte. Produkcja natomiast jest zbiorem symboli terminalnych i nieterminalnych. Zbiór ten jest skończony i wyraża zależności między różnymi elementami. Lewa strona produkcji jest zawsze pojedynczym symbolem terminalnym. Wynika z tego fakt, iż wybór produkcji do zastosowania nie zależy od kontekstu, w którym dana zmienna nieterminalna występuje. Co za tym idzie, wartość zastąpienia danej zmiennej nieterminalnej nie zależy od jej otoczenia. Ostatnim elementem składającym się na gramatykę bezkontekstową jest symbol wejściowy. Jest to symbol nieterminalny, od którego zaczynamy analizę lub generowanie ciągów języka.

**Definicja 2.8. Bezkontekstowy gramatyka formalna**

*Bezkontekstowa gramatyka formalna  $G$  jest czwórką  $(V, \Sigma, P, S)$ , gdzie:*

- $V$  jest skończonym zbiorem symboli nieterminalnych (zmiennych),
- $\Sigma$  jest skończonym zbiorem symboli terminalnych, przy czym  $V \cap \Sigma = \emptyset$ ,
- $P$  jest skończonym zbiorem produkcji, gdzie każda produkcja jest postaci  $A \rightarrow \alpha$ , gdzie  $A \in V$  i  $\alpha \in (V \cup \Sigma)^*$ ,
- $S \in V$  jest wyznaczonym symbolem startowym.

**Definicja 2.9. Symbol terminalny**

*W kontekście gramatyki formalnej  $G = (V, \Sigma, P, S)$ , symbole terminalne to te elementy  $\Sigma$ , które nie są przekształcane na inny ciąg symboli. Innymi słowy, jeśli  $a \in \Sigma$ , to nie istnieje żadna produkcja  $p \in P$  taka, że  $a$  jest na lewej stronie  $p$ .*

**Definicja 2.10. Symbol nieterminalny**

*Symbolem nieterminalnym nazywamy symbol, który może być zamieniony na inny ciąg symboli zgodnie z regułami produkcji danej gramatyki. Symbole tego typu są zazwyczaj używane do reprezentowania abstrakcyjnych struktur syntaktycznych w języku. W notacji gramatyk formalnych symbole nieterminalne często są zapisywane jako wielkie litery alfabetu.*

Języki bezkontekstowe są rozpoznawane i przetwarzane przez automaty ze stosem [2.11]. Automat ze stosem to rodzaj automatu skończonego, który w procesie przetwarzania wykorzystuje strukturę stosu do przechowywania informacji. W odróżnieniu od prostego automatu skończonego automat ze stosem zazwyczaj ma bardziej złożoną funkcję przejścia. Wynika to z faktu, iż automat tego typu uwzględnia aktualny symbol na górze stosu. Dodatkowo funkcja przejścia może także prowadzić do zmiany tego symbolu.

**Definicja 2.11. Automat ze stosem**

*Automat ze stosem to piątka:*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$$

gdzie:

- $Q$  jest skończonym zbiorem stanów,
- $\Sigma$  jest skończonym zbiorem symboli wejściowych (alfabet wejściowy),
- $\Gamma$  jest skończonym zbiorem symboli stosu,
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow P(Q \times \Gamma^*)$  to funkcja przejścia,
- $q_0 \in Q$  to stan początkowy,
- $Z_0 \in \Gamma$  to początkowy symbol stosu,
- $F \subseteq Q$  to zbiór stanów akceptujących.

Automat ze stosem akceptuje słowo, jeśli po przetworzeniu całego słowa wejściowego znajduje się w stanie akceptującym.

W celu lepszego poznania zagadnienia języków bezkontekstowych posłużymy się przykładem. Przykład, który będziemy rozpatrywać, jest językiem bezkontekstowym  $L$ , który składa się ze wszystkich poprawnie sparowanych nawiasów. Gramatykę  $G$  będącą gramatyką bezkontekstową tego języka, możemy zdefiniować w następujący sposób:

$$G = (\{S\}, \{(\,, \,)\}, P, S)$$

gdzie  $P$  to zbiór produkcji:

$$S \rightarrow SS$$

$$S \rightarrow (S)$$

$$S \rightarrow \varepsilon$$

Zauważmy, że  $\varepsilon$  oznacza słowo puste. Z wykorzystaniem powyżej zdefiniowanej gramatyki możemy generować wszystkie słowa, które składają się z poprawnie sparowanych nawiasów. Przykładem takich słów są na przykład  $()$ ,  $((\ ))$ ,  $(\ )()$ . Wszystkie wymienione słowa spełniają założenia gramatyki, a co za tym idzie, słowa te należą do języka  $L(G)$  generowanego przez tę gramatykę.

### 2.1.3 Języki kontekstowe

W porównaniu do języków regularnych i bezkontekstowych języki kontekstowe mogą reprezentować bardziej złożone struktury i wzorce. Sprawia to także, że języki tego typu są znacznie trudniejsze do interpretacji. Wymagają one na przykład algorytmów, które są bardziej złożone obliczeniowo. Z perspektywy praktycznej, stosuje się jednak z podzbiorów języków kontekstowych. Mniejsze języki są łatwiejsze do analizy i przetwarzania.

Podobnie jak języki bezkontekstowe języki kontekstowe znajdują zastosowanie w kompilatorach i interpretatorach. Działa to na podobnej zasadzie jak w przypadku języków bezkontekstowych, czyli parser generuje odpowiednią strukturę wprowadzonych danych. Dane te są rozbijane na drzewo parsowania, po czym taka struktura, może zostać wykorzystana w kompilatorach i interpretatorach. Oprócz tego stosuje się je do analizy języka naturalnego czy modelowania struktur cząstek takich jak RNA. W przypadku analizy języka naturalnego języki kontekstowe mogą być wykorzystane do reprezentowania uproszczonych modeli tego języka.

Podstawowym elementem języka kontekstowego podobnie jak w przypadku języka bezkontekstowego jest gramatyka. W przypadku języka kontekstowego wykorzystujemy gramatykę kontekstową [2.12]. Każda reguła produkcyjna zawarta w gramatyce kontekstowej ma specyficzną formę. Formę tą ukazano w podanej definicji gramatyki kontekstowej. W skład tej reguły wchodzi ciąg symboli  $\alpha$ . Ciągi te mogą być przekształcone lub przemapowane w ciąg symboli  $\beta$ . Wymienione symbole mogą być dowolne, lecz oparte o konkretne ograniczenia. Ograniczenia te są następujące:

- ciąg  $\alpha$  musi zawierać co najmniej jeden symbol nieterminalny [2.10],
- długość ciągu  $\alpha$  musi być mniejsza bądź równa długości ciągu  $\beta$ .

**Definicja 2.12. Gramatyka kontekstowa**

*Gramatyka kontekstowa to typ gramatyki formalnej, gdzie każda reguła produkcji ma formę:*

$$\alpha \rightarrow \beta$$

*gdzie  $\alpha$  i  $\beta$  są dowolnymi ciągami symboli gramatyki, pod warunkiem że  $\alpha$  zawiera przynajmniej jeden nieterminalny symbol, a długość  $\alpha$  (oznaczana jako  $|\alpha|$ ) jest mniejsza lub równa długości  $\beta$  (oznaczanej jako  $|\beta|$ ).*

Do rozpoznawania i przetwarzania języków kontekstowych wykorzystuje się maszyny Turinga z ograniczeniami. Jest to znacząca różnica pomiędzy omawianym rodzajem języków a poprzednimi. Nie jesteśmy w stanie wykorzystać do tego celu automatów stanowych ze względu na złożoność.

Maszyna Turinga z ograniczeniami, znana także jako maszyna Turinga z linią podziału, jest to specjalny rodzaj maszyny Turinga będący teoretycznym odpowiednikiem komputera. Podstawowa maszyna Turinga posiada nieskończoną taśmę podzieloną na komórki. Każda komórka może przechowywać jeden symbol. Maszyna przechodzi po taśmie w dwóch kierunkach, w lewo i w prawo. Po wybraniu odpowiedniej komórki maszyna może czytać wartość danego symbolu lub go zmienić.

Wersja maszyny z ograniczeniami różni się od wersji podstawowej tym, że taśma jest podzielona na dwie sekcje. Wyróżniamy lewą stronę i prawą, oddzielone są one odpowiednim symbolem zwanym linią podziału. Poruszanie się maszyny po taśmie jest ograniczone do obrębu aktualnej sekcji. Zmiana linii podziału jest możliwa tylko w przypadku gdy maszyna znajduje się na tej linii.

Aby ukazać praktyczny przykład języka kontekstowego, złożmy przykład języka składającego się ze słów, które mają taką samą liczbę symboli. Niech założonymi słowami będą litery “a”, “b” i “c”. Formalnie język ten można zdefiniować w następujący sposób:

$$L = \{a^n b^n c^n \mid n \geq 0\}$$

Należy to analizować w następujący sposób:

- dla  $n = 0$  mamy słowo puste ( $\epsilon$ ),
- dla  $n = 1$  mamy słowo “abc”,
- dla  $n = 2$  mamy słowo “aabbcc” itd.

W celu wykrycia czy język jest kontekstowy, czy nie, możemy wykorzystać lemat o pompowaniu dla języków niekontekstowych [2.1]. Chcemy pokazać, że język  $\{a^n b^n c^n \mid n \geq 0\}$  nie jest bezkontekstowy. Lemat jednak stosujemy na językach bezkontekstowych do udowodnienia, że język bezkontekstowy bezkontekstowym nie jest. Załóżmy więc, że przedstawiony język jest językiem bezkontekstowym.

**Twierdzenie 2.1. Lemat o pompowaniu dla języków bezkontekstowych**

*Dla dowolnego języka bezkontekstowego  $L$ , istnieje pewna liczba naturalna  $p$  (długość pompowania), taka, że dla każdego słowa  $w$  należącego do języka  $L$ , jeżeli  $|w| \geq p$ , to  $w$  można podzielić na pięć części,  $w = uvxyz$ , takie, że:*

1.  $uv^i xy^i z \in L$  dla każdego  $i \geq 0$
2.  $|vy| > 0$
3.  $|vxy| \leq p$

W celu przeprowadzenia dowodu musimy określić wartość pompowania oznaczoną jako wartość  $p$  w lemacie. Niech nasze słowo dla określonego  $p$  będzie następujące w badanym języku:  $a^p b^p c^p$ . Zgodnie z podanym twierdzeniem powinniśmy być w stanie podzielić słowo na pięć części  $uvwxz$ , tak, że  $uv^i wx^i y$  należy do języka dla każdego  $i$ . Na przykład, dla  $v = a^2$ ,  $w = b^5$ ,  $x = c^3$  otrzymujemy słowo  $s$  jak poniżej:

$$s = uv^2 wx^2 y = ua^4 b^5 c^5 y \quad \text{dla } i = 2$$

Z twierdzenia założymy pompowanie dla  $u = a^8$  i  $y = b^5 c^7$  (wartości te muszą spełniać warunki z lematu), wtedy otrzymujemy:

$$s = a^8 a^4 b^5 c^5 b^5 c^7 = a^{12} b^{10} c^{12}$$

$$s = \text{aaaaaaaaaabbcccccccccc}$$

Wynika z tego, że słowo  $s$  jest sprzeczne z założeniem języka. Nie ważne jak podzielimy słowo  $s = uv^2 wx^2 y$ , zawsze będzie zawierało więcej słów “a”, “b”, “c” niż powinno. Słowo to więc nie będzie spełniało warunku  $\{a^n b^n c^n \mid n \geq 0\}$  dla  $i > 1$ .

Z twierdzenia o pompowaniu wiemy więc, że podany język nie może być językiem bezkontekstowym. Poprzez zaprzeczenie tego faktu jesteśmy w stanie określić, że język ten jest językiem kontekstowym.

### 2.1.4 Języki rekurencyjnie przeliczalne

Języki rekurencyjnie przeliczalne [2.13] są najbardziej ogólną klasą języków formalnych. Ich definicja jest oparta o maszyny Turinga, które to są najbardziej ogólnym modelem obliczeń w teorii języków formalnych.

**Definicja 2.13. Język rekurencyjnie przeliczalny**

*Język nazywamy językiem rekurencyjnie przeliczalnym, gdy istnieje maszyna Turinga, która go akceptuje. Wynika z tego, że muszą zostać spełnione następujące warunki:*

- dla każdego słowa akceptowalnego w języku maszyna zatrzyma się w stanie akceptującym,
- dla każdego słowa nieakceptowalnego w języku maszyna zatrzyma się w stanie nieakceptowalnym lub nie zatrzyma się wcale.

Głównym zastosowaniem języków tego rodzaju jest teoria obliczeń, złożoność obliczeniowa i tworzenie i analiza algorytmów. W przypadku teorii obliczeń języki te stanowią narzędzie do zrozumienia granic obliczeń. Przykładem takiego problemu jest problem zatrzymania, który jest nierozstrzygalny. Problem ten może być zrozumiany przez analizę języka rekurencyjnie przeliczalnego, definiującego wszystkie pary maszyn Turinga i wejść, dla których maszyna się zatrzyma.

Kolejnym przykładem zastosowania jest złożoność obliczeniowa. Dla danego problemu jesteśmy w stanie zdefiniować język, który składa się ze wszystkich instancji problemu. Instancje te są zapisywane jako odpowiedź oznaczająca na przykład “tak”. Możemy wtedy analizować czy ten język jest rekurencyjnie przeliczalny.

W kwestii tworzenia i analizy algorytmów samo zrozumienie zasad działania języków rekurencyjnych jest bardzo pomocne. Szczególnie przydatne jest w algorytmach, które oparte są o skomplikowane struktury danych.

Gramatyka w tym języku zwana jest gramatyką bez ograniczeń [2.14]. Jest to typ gramatyki bardzo mocnej obliczeniowo, jednak zarazem bardzo trudnej w definiowaniu i interpretacji. Można powiedzieć, że są to najbardziej ogólne gramatyki i mogą generować wszystkie języki formalne, które są rekurencyjnie przeliczalne.

**Definicja 2.14. Gramatyka bez ograniczeń**

*Gramatyka  $G$  bez ograniczeń jest określona jako czwórka  $(N, \Sigma, P, S)$ , gdzie:*

- $N$  jest skończonym zbiorem nieterminali,
- $\Sigma$  jest skończonym zbiorem terminali, który jest rozłączny z  $N$  ( $N \cap \Sigma = \emptyset$ ),
- $P$  jest skończonym zbiorem produkcji, gdzie każda produkcja jest postaci  $\alpha \rightarrow \beta$ , gdzie  $\alpha, \beta \in (N \cup \Sigma)^+$ , a  $\alpha$  zawiera przynajmniej jeden symbol z  $N$ ,
- $S$  jest początkowym symbolem, oraz należy do  $N$ .

W gramatyce bez ograniczeń, produkcje mogą mieć dowolną formę, tak długo, jak lewa strona zawiera co najmniej jeden nieterminal, a prawą stronę można zostawić pustą. Dzięki temu wiemy, że gramatyka bez ograniczeń może przekształcić dowolny ciąg symboli w dowolny inny ciąg symboli. Należy jednak pamiętać, że warunkiem możliwości przekształcenia jest to, że ciąg wejściowy zawiera przynajmniej jeden nieterminal.

Dzięki zastosowaniu tego typu gramatyk jesteśmy w stanie modelować dowolny problem obliczeniowy. Model ten musi być możliwy do rozwiązania za pomocą maszyny Turinga. Sama złożoność tego typu gramatyki zatem będzie równoważna mocy maszyny Turinga.

Ważnym elementem w tym rodzaju języków jest maszyn Turinga [2.15]. Została ona zaproponowana przez Alana Turinga w 1936 roku. Jest to prosty model obliczeniowy, umożliwiający wykonanie dowolnych obliczeń, które są możliwe do wykonania na każdym komputerze. W skład maszyny Turinga wchodzi następujące elementy:

- nieskończona taśma podzielona na komórki,
- głowica, która ma możliwość przemieszczania się w lewo lub prawo z możliwością odczytania i zmiany wartości komórki,
- zestaw określonych stanów wraz ze stanem początkowym i stanem akceptacyjnym,
- funkcję przejścia stanowiącą o rzeczach, które należy wykonać dla danej komórki.

### **Definicja 2.15. Maszyna Turinga**

Formalna definicja maszyny Turinga to krotka  $(Q, \Sigma, \Gamma, \delta, q_0, q_{Accept}, q_{Reject})$ , gdzie:

- $Q$  jest skończonym zbiorem stanów.
- $\Sigma$  jest skończonym zbiorem symboli wejściowych (alfabetem wejściowym), niezawierającym specjalnego pustego symbolu (zwyczajowo reprezentowanego jako  $B$ ).
- $\Gamma$  jest skończonym zbiorem symboli taśmowych, które może odczytywać i zapisywać głowica;  $\Sigma$  jest podzbiorem  $\Gamma$ , a  $B$  należy do  $\Gamma$ .
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$  jest funkcją przejścia. Dla danego stanu i symbolu taśmowego,  $\delta$  zwraca trójkę składającą się z nowego stanu, nowego symbolu do zapisania na taśmie, oraz kierunku, w którym głowica powinna się przesunąć ( $L$  dla lewo,  $R$  dla prawo).
- $q_0 \in Q$  jest początkowym stanem.
- $q_{Accept} \in Q$  jest stanem akceptującym.
- $q_{Reject} \in Q$  jest stanem odrzucającym, gdzie  $q_{Accept} \neq q_{Reject}$ .

Rozpoczęcie pracy maszyna Turinga zaczyna się od stanu początkowego, z głowicą ustawioną na pierwszym symbolu wejściowym. Każdy krok maszyny wykorzystuje funkcję  $\sigma$ , aby zdecydować jaką operację należy wykonać. Funkcja ta może określić, jaki jest następny stan, jaki symbol zapisać na taśmie i w którym kierunku przesunąć głowicę. W momencie,

gdy maszyna osiągnie stan akceptujący lub odrzucający, obliczenie można uznać za zakończone. Istnieje także sytuacja, w której maszyna nie zatrzyma się, jednak jest to przypadek skrajny i błąd prawdopodobnie wynika z błędnie określonego stanu końca.

Przykładem języka opartego na językach rekurencyjnie przeliczalnych jest język składający się ze wszystkich poprawnych programów napisanych w dowolnym kompletnej Turinga języku programowania, takim jak C, Python, czy JavaScript.

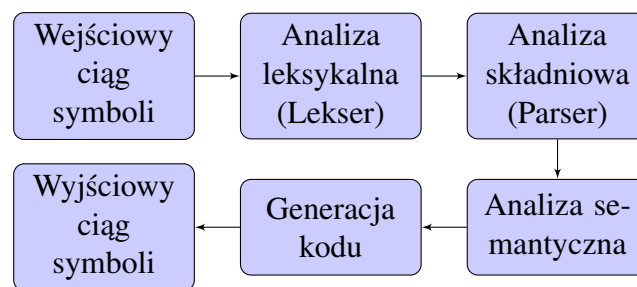
Formalnie, można go zdefiniować jako zbiór wszystkich ciągów znaków, które stanowią poprawny program w danym języku programowania. Przyjmijmy, że  $\Sigma$  jest zbiorem wszystkich możliwych znaków w języku programowania, a  $P$  jest programem w tym języku. Wtedy język  $L$  składa się ze wszystkich  $P$  takich, że  $P$  jest poprawnym programem w języku.

Języki programowania są naturalnym przykładem języków generowanych przez gramatyki bez ograniczeń (czyli języków rekurencyjnie przeliczalnych), ponieważ mogą one wyrażać dowolne obliczenie, które jest możliwe do przeprowadzenia na maszynie Turinga. Każdy program jest skończonym ciągiem symboli, które mogą być generowane za pomocą gramatyki bez ograniczeń.

## 2.2 Proces analizy leksykalnej i składniowej

Kluczowym elementem każdego języka zarówno formalnego, jak i naturalnego jest możliwość tłumaczenia tego języka na inne języki. W przypadku języka naturalnego tłumaczenie odbywa się na poziomie gramatycznym, jak i na poziomie tłumaczenie słów. Języki formalne są jednak znacznie prostsze do tłumaczenia pod względem logicznym. W tłumaczeniu takich języków skupiamy się przede wszystkim na wyszukiwaniu logicznych fragmentów języka.

Proces tłumaczenia języka formalnego polega na konwersji ciągu symboli zgodnie z określoną gramatyką języka formalnego. W kontekście kompilacji, tłumaczenie języka formalnego często obejmuje analizę leksykalną, syntaktyczną, semantyczną oraz generację kodu. Grafika poniżej (Rysunek 2.1) przedstawia sposób, w jaki przebiega proces tłumaczenia języka formalnego.



Rysunek 2.1: Przebieg tłumaczenia języków formalnych

Proces ten opiera się o dwa najważniejsze elementy, nie licząc samej gramatyki języka. Pierwszym kluczowym elementem jest analiza leksykalna, która skupia się na znajdowaniu elementów kluczowych języka. Elementami tymi są słowa kluczowe, operatory, fragmenty opisane określoną formą jak ciągi znaków czy wartości numeryczne. Drugim kluczowym elementem jest analiza składniowa, odpowiadająca za wyszukiwanie logicznych relacji między tokenami, które to są wynikiem analizy leksykalnej.



W tym procesie dane wejściowe w formie tekstu lub innego akceptowalnego formatu są tłumaczone na zbiór instrukcji. Instrukcje te mogą być dalej przetwarzane przez inne języki formalne, języki programowania czy też przez maszyny w postaci kodu maszynowego.

W tej sekcji zostaną omówione dwa najważniejsze elementy tego procesu, które zostały już wymienione wyżej. Skupimy się w szczególności na analizie leksykalnej oraz analizie składniowej. Zostaną opisane najważniejsze elementy tych pojęć, wraz ze szczegółowym omówieniem.

## 2.2.1 Lekser

Analiza leksykalna odpowiada za ekstrakcję z danych wejściowych elementów, które nadają się do analizy składniowej. Za ekstrakcję danych odpowiada aplikacja lub specjalna funkcja zwana lekserem.

Lekser (lexer) [2.16] może być spotkany w literaturze również pod nazwą analizatora leksykalnego. Analiza wykonywana przez lekser jest pierwszym elementem tłumaczenia języka formalnego. Główną rolą tej aplikacji jest przekształcenie sekwencji znaków wejściowych na sekwencję leksemów.

### **Definicja 2.16. Lekser (analizator leksykalny)**

*Niech lekser będzie funkcją  $lex$ , która przekształca sekwencję znaków wejściowych  $input$  na sekwencję tokenów  $output$ . Funkcja  $lex$  może być zdefiniowana jako:*

$$lex : \Sigma^* \rightarrow Token^*$$

*gdzie  $\Sigma$  oznacza zbiór wszystkich możliwych sekwencji znaków wejściowych, a  $Token$  oznacza zbiór wszystkich możliwych sekwencji tokenów wyjściowych.*

Leksemy [2.17] stanowią najmniejszą jednostkę semantyczną w danym języku formalnym. Przyjmują one różne formy, zgodnie z regułami zdefiniowanymi w gramatyce. Mogą one reprezentować takie elementy jak identyfikatory, słowa kluczowe, stałe, operatory oraz symbole specjalne.

### **Definicja 2.17. Token**

*Niech token będzie zbiorem par ( $typ\_leksemu$ ,  $wartość\_leksemu$ ), gdzie  $typ\_leksemu$  to dowolny rodzaj leksemu, a  $wartość\_leksemu$  to konkretna wartość leksemu. Token reprezentuje najmniejszą jednostkę semantyczną w analizie leksykalnej.*

Proces analizy leksykalnej w leksersze składa się z czterech kroków. Pierwszym krokiem jest tokenizacja, następnie lekser filtruje białe znaki. Po odpowiednim przygotowaniu danych następuje rozpoznanie leksemów i weryfikacja błędów leksykalnych.

Pierwszy etap, czyli tokenizacja polega na podziale wejściowej sekwencji znaków na leksemy zwane także potocznie tokenami. Token jest to nic innego jak para składająca się z dwóch wartości, typu leksemu oraz wartości leksemu. Typem leksemu jest na przykład identyfikowana przez niego wartość jak identyfikator, stała itd. Wartość określa konkretny fragment danych opisujący dane leksemy odpowiadającą formatowi ich typów.

Filtracja białych znaków pełni rolę kroku filtracyjnego leksemy, które nie mają znaczenia w kontekście danej gramatyki. Pomija on takie znaki jak tabulatory, spacje, znaki nowych linii oczywiście, jeżeli gramatyka zakłada omijanie tych znaków.

Przed ostatnim elementem jest rozpoznawanie leksemów. Lekser analizuje sekwencję znaków wejściowych, dopasowując je do określonych wzorców. Wzorce te reprezentują różne rodzaje tokenów w języku. Może to obejmować sprawdzanie identyfikatorów zarezerwowanych słów kluczowych, rozpoznawanie złożonych struktur jak komentarze, czy też rozpoznawanie rodzajów wartości reprezentowanych przez token.

Ostatnim elementem w procesie analizy leksykalnej, tak jak zostało już nadmienione wcześniej, jest wykrywanie błędów leksykalnych. Błędy zgłaszane są w momencie, kiedy przeanalizowana struktura tokenów nie zgadza się z żadną dostarczoną definicją. Przykładem takiego błędu może być nieoczekiwany znak na początku zdefiniowanego identyfikatora.

Proces ten sprawia, że wprowadzone dane są łatwiejsze do interpretacji w dalszym ciągu tłumaczenia. Pomaga on także wykryć ewentualne błędy, przed wdrożeniem kolejnego etapu tłumaczenia co sprawia, że wykrywanie błędów staje się bardziej efektywne. Proces analizy leksykalnej jest znacznie mniej złożonym procesem niż proces analizy składniowej. Wynika to z faktu, iż zazwyczaj złożoność obliczeniowa lekserów jest znacznie mniejsza od złożoności parserów ze względu na ilość reguł i ich złożoność.

## 2.2.2 Parser

Parsery są często wykorzystywane w kompilatorach i interpreterach do analizy składni kodu źródłowego w językach programowania. Wynik parsowania zazwyczaj wykorzystywany jest do tłumaczenia określonych operacji na języki maszynowe lub języki niższego poziomu. Wykorzystanie ich można również spotkać w wielu innych kontekstach, takich jak przetwarzanie języka naturalnego. W przypadku przetwarzania języka naturalnego służą do analizy składniowej zdań.

Zagadnienie parsera możemy także spotkać w literaturze jako analizator składniowy. Jest to program lub algorytm, który pełni funkcję przekształcania ciągów symboli wejściowych (zazwyczaj w postaci tokenów) w strukturę danych. Zazwyczaj tą strukturą jest drzewo parsowania [2.18], które reprezentuje składnię tych symboli zgodnie z określoną gramatyką.

### **Definicja 2.18. Drzewo parsowania**

*Drzewo parsowania dla gramatyki  $G$  i ciągu wejściowego  $w$  to drzewo etykietowane, które spełnia następujące warunki:*

- 1. Każdy węzeł jest etykietowany przez jeden z symboli gramatyki (terminali lub nieterminali).*
- 2. Etykieta korzenia drzewa to symbol startowy gramatyki.*
- 3. Jeśli węzeł jest etykietowany przez nieterminal, to dzieci tego węzła odpowiadają symbolom w prawej stronie pewnej produkcji dla tego nieterminala w gramatyce.*
- 4. Jeśli węzeł jest etykietowany przez terminal, to ten węzeł jest liściem drzewa.*
- 5. Ciąg terminali otrzymany przez przeczytanie etykiet liści drzewa z lewej na prawo jest równy ciągowi wejściowemu  $w$ .*

Istnieją różne techniki parsowania, które mogą być stosowane w zależności od rodzaju gramatyki i wymagań aplikacji. Niektóre powszechnie stosowane techniki parsowania obejmują:

- parsowanie rekurencyjne zstępujące,
- parsowanie LL,
- parsowanie adaptatywne LL,
- parsowanie LR,
- parsowanie Earleya.

### **Rekurencyjne parsowanie zstępujące**

Rekurencyjne parsowanie zstępujące jest typem analizy, który zaczyna on najwyższego poziomu drzewa składniowego. W ten sposób z wykorzystaniem produkcji gramatyki próbuje zbudować drzewo od góry do dołu. Podczas procesu parsowania, każda produkcja gramatyki w celu dopasowania danych wyjściowych jest opisana funkcją lub metodą. Jeśli produkcja gramatyki zawiera nieterminale, funkcja dla danej produkcji może rekurencyjnie wywołać inne funkcje reprezentujące te nieterminale. Ten typ analizy jest “rekurencyjny”, ponieważ proces dopasowywania produkcji do danych wejściowych może wymagać rekurencyjnego wywołania funkcji.

### **Parsowanie LL**

Parsowanie LL jest metodą analizy stosowaną w niektórych językach bezkontekstowych. Dane wejściowe w tej metodzie są analizowane od lewej strony do prawej, rozwijając produkcję zgodnie ze strategią od lewej do prawej. Ten typ parsowania często stosowany jest z gramatykami, które nie obsługują lewostronnych rekurencji. Lewostronna rekurencja [2.19] jest istotna, ponieważ może prowadzić do nieskończonych pętli przy zastosowaniu podejścia zstępującego. O lewostronnej rekurencji mówimy, kiedy nieterminał na początku jakiejś produkcji pojawia się także na początku prawej strony tej produkcji.

#### **Definicja 2.19. Lewostronna rekurencja**

*Mówimy, że gramatyka bezkontekstowa zawiera lewostronną rekurencję, jeśli istnieje produkcja postaci  $A \rightarrow A\alpha$ , gdzie  $A$  jest nieterminalem, a  $\alpha$  jest dowolnym ciągiem terminali i nieterminali.*

Notacja LL(k) oznacza parsowanie LL, które korzysta z k-symbolowego podglądu wejścia. Najczęściej używane jest LL(1), które oznacza, że parser korzysta tylko z jednego symbolu podglądu. Oznacza to, że decyzja o tym, którą regułę gramatyki zastosować, jest podejmowana na podstawie bieżącego symbolu wejściowego.

Główna procedura parsowania LL wygląda następująco:

- Na początku, parser tworzy stos z symbolem startowym gramatyki na szczycie.

- Następnie, parser kontynuuje pobieranie symboli z góry stosu i porównuje go z następnym symbolem wejściowym.
- Jeżeli symbol na szczycie stosu i następny symbol wejściowy, są identyczne, to parser usuwa ten symbol ze stosu i przesuwa wskaźnik wejścia, do następnego symbolu.
- Jeżeli symbol na szczycie stosu jest nieterminalem, parser szuka w tabeli parsowania reguły do zastosowania. Reguła jest wybrana na podstawie symbolu na szczycie stosu i następnego symbolu wejściowego. Parser zastępuje symbol na szczycie stosu, prawą stroną reguły.
- Proces jest kontynuowany, dopóki stos nie jest pusty i nie zostanie przetworzone całe wejście.

### **Parsowanie adaptatywne LL**

Adaptatywne parsowanie LL (Adaptive LL, ALL(\*)) jest rozwinięciem tradycyjnej metody parsowania LL. Ta wersja parsowania jest bardziej elastyczna, ponieważ wykorzystuje dynamiczny wybór produkcji podczas parsowania.

Decyzja, którą produkcję zastosować jest podejmowana na podstawie aktualnego stanu analizatora i następnego tokenu. Decyzje te są zapisane w specjalnej tabeli, która jest tworzona na podstawie gramatyki przed rozpoczęciem parsowania. Dzięki takiemu podejściu parsowanie tego typu jest w stanie obsługiwać szerszy zakres gramatyk niż parsowanie LL. Oprócz tego możliwe jest także parsowanie gramatyk, które są poza zasięgiem zarówno parsowania LL, jak i parsowania LR.

Omawiana wersja jest znacznie bardziej elastyczna, jednak dzieje się to kosztem większej ilości zasobów obliczeniowych. Prowadzi to do wolniejszego parsowania w porównaniu do tradycyjnego parsowania LL.

### **Parsowanie LR**

Parsowanie LR (Left-to-right, Rightmost derivation) to technika parsowania wykorzystywana do analizy języków bezkontekstowych. Ten typ parsowania działa na podstawie, rozpatrywania wejście od lewej do prawej (L), przeprowadzając prawostronne redukcje (R), aż do osiągnięcia początkowego symbolu gramatyki.

Podobnie jest w parsowaniu LL, parser wykorzystuje  $k$  (LR( $k$ )) kolejnych symboli wejściowych, aby podejmować decyzję o parsowaniu. Najpowszechniejszym typem jest parsowanie dla  $k=1$  (LR(1)), w którym to uwzględnia się tylko pierwszy następny token.

Procedura parsowania LR jest oparta o algorytm oparty o stos. Stos służy do przechowywania stanów analizatora oraz symboli gramatycznych. Wyróżniam cztery stany analizatora takie jak shift, reduce, accept oraz error.

W przypadku, gdy pierwszym elementem stosu czytając od góry, jest stan  $S$ , i następny symbol wejściowy to "a", to z tabeli wyszukujemy odpowiednią akcję, jaką musimy wykonać:

- gdy tabela parsowania dla przedstawionej sytuacji, zawiera akcję "shift", to symbol wejściowy jest przesuwany na stos,

- gdy tabela parsowania dla przedstawionej sytuacji, zawiera akcję “reduce”, to symbol wejściowy jest usuwany zgodnie z prawą stroną reguły gramatycznej, a następnie na stos jest przekazywany symbol z lewej strony reguły,
- gdy tabela parsowania dla przedstawionej sytuacji, zawiera akcję “accept”, to wprowadzone dane są traktowane jako poprawna składnia gramatyczna,
- gdy tabela parsowania dla przedstawionej sytuacji, zawiera akcję “error”, to parser zwraca błąd składniowy,

Konstrukcja tabeli parsowania LR wymaga analizy gramatyki i jest zazwyczaj wykonana automatycznie przez narzędzie generatora parsera.

### **Parsowanie Earleya**

Parsowanie Earleya jak sama nazwa wskazuje jest oparta o algorytm Earleya. Pozwala on na obsługę dowolnych gramatyk bezkontekstowych, w tym tych, które są lewo- i prawostronnie rekurencyjne. Został stworzony przez Jaya Earleya w 1970 roku. Jest to efektywna technika parsowania, z czasem działania liniowym dla sporej części gramatyk.

Główna idea algorytmu Earleya polega na dynamicznym budowaniu zestawu “punktów” (items) dla każdej pozycji w ciągu wejściowym. Każdy punkt reprezentuje częściowo przetworzoną produkcję gramatyczną. Algorytm wykonuje trzy podstawowe operacje na punktach:

- predykcję — tworzenie nowych punktów na podstawie bieżących produkcji,
- skanowanie — przesuwanie się do przodu w ciągu wejściowym,
- zakończenie — uznawanie, że produkcja została w pełni przetworzona.

Operacje te stanowią podstawę procedury algorytmu Earleya. Możemy teraz przytoczyć opisową wersję algorytmu. Składa się ona z czterech etapów: inicjalizacji, predykcji, skanowania, zakończenia. Procedura jest powtarzana, aż algorytm przejdzie przez cały ciąg wejściowy. Jeśli na końcu istnieje punkt reprezentujący w pełni przetworzoną produkcję startową, ciąg wejściowy jest akceptowany przez gramatykę; w przeciwnym razie, jest odrzucony.

Na początku działania, algorytm Earleya koncentruje się na tworzeniu zestawu punktów. Punkty te są generowane dla pozycji zerowej, czyli na samym początku ciągu wejściowego. Podstawą dla generowania tych punktów są produkcje startowe gramatyki, które reprezentują możliwe początkowe ścieżki przetwarzania.

Algorytm analizuje te początkowe produkcje i na ich podstawie tworzy punkty, które będą reprezentować początkowe etapy przetwarzania ciągu wejściowego. Każdy z tych punktów zawiera informację o tym, jaki fragment produkcji został już przetworzony i jaki fragment jeszcze czeka na przetworzenie.

Tym samym, na początku działania algorytmu, mamy zestaw punktów reprezentujących różne możliwe kierunki przetwarzania ciągu wejściowego. Zestaw ten będzie stopniowo rozbudowywany i przekształcany w kolejnych etapach działania algorytmu.

Drugi krok, jakim jest predykcja, dla każdego punktu wprowadzonego do zestawu, który reprezentuje produkcję z nieterminaliem, algorytm dodaje nowe punkty. Te nowe punkty są reprezentacjami wszystkich możliwych produkcji, które mogą nastąpić dla danego nieterminalu. W ten sposób algorytm Earleya tworzy i rozszerza zestaw punktów, które reprezentują potencjalne ścieżki parsowania przez strukturę gramatyczną ciągu wejściowego. Każdy punkt w zestawie reprezentuje częściowo zinterpretowaną produkcję gramatyczną, a dodawanie nowych punktów na podstawie nieterminalów pozwala na rozpatrzenie wszystkich możliwych ciągów dalszego parsowania.

Kolejnym etapem jest skanowanie, które dla każdego punktu będącego produkcją z nieterminaliem, algorytm tworzy nowy punkt w następnym zestawie. Dzieje się to tylko i wyłącznie w przypadku gdy podany terminal odpowiada aktualnemu stanowi wejściowemu. Nowy punkt reprezentuje produkcję po przetworzeniu dopasowanego terminala. Innymi słowy, symbol terminala, który dopasowaliśmy, jest “przesunięty” do przodu w produkcji reprezentowanej przez nowy punkt. Proces ten jest kontynuowany dla wszystkich punktów w bieżącym zestawie, a następnie powtarzany dla każdego kolejnego zestawu punktów, aż algorytm przejdzie przez cały ciąg wejściowy.

Rozpatrując ostatni krok będący zakończeniem, weźmy pod uwagę punkt z aktualnego zestawu punktów. Załóżmy, że ten konkretny punkt reprezentuje produkcję, którą algorytm Earleya uważa za w pełni przetworzoną. Teraz, w obrębie tego samego zestawu punktów, mamy inne punkty, które przewidują pojawienie się nieterminalu na bieżącej pozycji w produkcji. To znaczy, że te punkty są ustawione tak, aby oczekiwać na pojawienie się tego nieterminalu, którego produkcję właśnie przetworzyliśmy. W następnym kroku algorytm Earleya dodaje do obecnego zestawu punktów dodatkowe punkty dla każdego z tych punktów przewidujących. To oznacza, że algorytm “dokończy” przetwarzanie tych punktów, które czekały na przetworzenie nieterminalu. Ta operacja jest kluczowym elementem procedury algorytmu Earleya, pozwalając mu śledzić, które sekwencje symboli są zgodne z gramatyką.

## **2.3 Przegląd narzędzi i notacji do definiowania gramatyk**

### **2.3.1 BNF**

BNF (Backus-Naur Form) to notacja używany do opisu składni języków formalnych bez-kontekstowych. Został opracowany przez Johna Backusa i Petera Naura w latach 50 i 60 XX wieku.

Głównym celem notacji BNF jest dostarczenie przejrzystego sposobu przedstawienia składni języków formalnych. Jej główne cechy to czytelność i formalność. Pozwala ona na intuicyjne przedstawianie struktury języka formalnego, umożliwiając prostsze badanie jego składni.

Przykład 2.1 ukazuje podstawową składnię gramatyki zdefiniowanej w BNF. W przykładzie możemy zauważyć takie elementy jak symbole nieterminalne, symbole terminalne, oraz różnego rodzaju operatory wchodzące w skład syntaktyki BNF. Wszystkie elementy składniowe zostały przedstawione w tabeli 2.1.

```

1  <expression> ::= <term>
2                  | <expression> + <term>
3                  | <expression> - <term>
4  <term>         ::= <factor>
5                  | <term> * <factor>
6                  | <term> / <factor>
7  <factor>        ::= <number>
8                  | ( <expression> )
9  <number>        ::= <digit>
10                 | <digit> <number>
11 <digit>          ::= 0 | 1 | 2 | 3 | 4 | 5 | 6
12                 | 7 | 8 | 9

```

Przykład 2.1: Przykład gramatyki napisanej w BNF

Elementy składniowe w tym języku są reprezentowane przez symbole terminalne i nieterminalne. Symbolem terminalnym jest, każdy element wprowadzony do gramatyki niebędący elementem opisanym przez notację. Definicje składni są przedstawione za pomocą operatora definicji “:=”, która łączy symbole z definicjami. Nazwę symboli definiujemy wewnątrz ostrych nawiasów “<...>”.

Podstawowa wersja notacji BNF nie posiada specjalnych elementów typu element opcjonalny czy powtórzenia. Musimy posłużyć się wykorzystaniem podstawowych elementów. Dla przykładu element opcjonalny musimy zbudować z dwóch reguł. W przykładzie 2.2 została przedstawiona definicja opcjonalnego symbolu, jakim jest znak w liczbie.

```

1  <number with sign> ::= <optional sign> <number>
2  <number>           ::= <digit> | <digit> <number>
3  <digit>            ::= 0 | 1 | 2 | 3 | 4 | 5 | 6
4                   | 7 | 8 | 9
5  <optional sign>    ::= | <sign>
6  <sign>             ::= + | -

```

Przykład 2.2: Przykład elementu opcjonalnego i powtórzeń w BNF

Ważnym elementem są także powtórzenia, które są bardzo często wykorzystywane w gramatykach. W tym przypadku BNF również nie przychodzi ze wbudowanym rozwiązaniem. Powtórzenia muszą zostać zbudowane manualnie z wykorzystaniem rekurencji czy też poprzez określenie odpowiedniej liczby wystąpień danej reguły. Wykorzystanie rekurencji umożliwi nam jednak stworzenie powtórzeń typu 1 lub więcej (<number> ::= <digit> | <digit> <number>) oraz 0 lub więcej (<number> ::= | <digit> <number>).

Syntaktyka poszczególnych elementów notacji jest podana w tabeli poniżej 2.1.

Element	Opis
<symbol> ::= ...	Definicja reguły
<symbol>	Symbol nieterminalny
...	Symbol terminalny
(spacja)	Operator konkatencji
	Operator alternatywy

Tabela 2.1: Syntaktyczne elementy języka BNF

Każda gramatyka musi składać się przynajmniej z jednego symbolu nieterminalnego. Pierwszy symbol nieterminalny jest punktem startowym, wykorzystywanym przez parser.

### 2.3.2 EBNF

EBNF (Extended Backus-Naur Form) jest rozszerzoną wersją notacji BNF. Tak samo jak BNF stosowana jest do definiowania gramatyk języków formalnych. Notacja ta wprowadza kilka rozszerzeń składniowych. Przykładem takiego rozszerzenia jest możliwość definiowania opcjonalnych elementów z wykorzystaniem kwadratowych nawiasów. Oprócz tego dodane zostały także składnie do definiowania powtórzeń zero lub więcej razy czy jeden, lub więcej razy oraz możliwość grupowania elementów. Umożliwia ona definiowanie gramatyk wykorzystywanych w językach formalnych bezkontekstowych. Jesteśmy także w stanie wykorzystywać wyrażenia regularne do definiowania różnych rodzajów wartości i ich form. Przykład 2.4 przedstawia gramatykę opisującą adres email.

```

1  expression      = term, { ("+" | "-"), term } ;
2  term            = factor, { ("*" | "/"), factor } ;
3  factor          = number | "(", expression, ")" ;
4  number          = digit, { digit } ;
5  digit           = "0" | "1" | "2" | "3" | "4" | "5"
6                  | "6" | "7" | "8" | "9" ;

```

Przykład 2.3: Przykład gramatyki w EBNF

W tym przykładzie gramatyka EBNF składa się z kilku reguł. Pierwszy element “expression” to główna reguła, która opisuje wyrażenie arytmetyczne. Składa się ona z “term” oraz opcjonalnych operatorów dodawania lub odejmowania, oddzielonych przecinkiem. Reguła “term” składa się z “factor” oraz opcjonalnych operatorów mnożenia lub dzielenia. Kolejna reguła “factor” opisuje liczby oraz nawiasy, które mogą otaczać wyrażenie arytmetyczne. Dwie najważniejsze reguły to “number” opisująca pojedynczą cyfrę oraz “digit” definiuje zestaw możliwych cyfr.

Syntaktyka poszczególnych elementów notacji jest podana w tabeli poniżej 2.2.

Element	Opis
symbol = ...	Definicja reguły
symbol	Symbol nieterminalny
'...' lub "..."	Symbol terminalny
,	Operator konkatencji
	Operator alternatywy
[...]	Operator opcjonalności
{...}	Operator powtórzenia (0 lub więcej)
{...}-	Operator powtórzenia (1 lub więcej)
n*...	Operator powtórzenia (n powtórzeń)
(...)	Grupowanie
(* ... *)	Komentarz
;	Znak końca

Tabela 2.2: Syntaktyczne elementy języka EBNF



### 2.3.3 ABNF

ABNF (Augmented Backus-Naur Form) podobnie jak EBNF rozszerza możliwości podstawowej notacji BNF. Oprócz elementów, które dostarcza EBNF ta wersja notacji dostarcza rozszerzone operatory powtórzeń. Jesteśmy w stanie definiować w nim powtórzenia o określonej liczbie reprezentacji czy też określenia zakresu ilości wystąpień danego elementu składni. Głównym założeniem tego podejścia jest wykorzystanie dwukierunkowych protokołów komunikacyjnych w językach formalnych. Przykład 2.4 przedstawia gramatykę opisującą adres email.

```
1 email-address = local-part "@" domain
2 local-part    = 1*atext
3 domain        = sub-domain *("." sub-domain)
4 sub-domain    = 1*alphanum
5 atext         = ALPHA / DIGIT / "!" / "#" / "$" / "%" / "&"
6               / "'" / "*" / "+" / "-" / "/" / "=" / "?"
7               / "^" / "_" / "`" / "{" / "|" / "}" / "~"
8 alphanum      = ALPHA / DIGIT
9 ALPHA         = %x41-5A / %x61-7A ; A-Z / a-z
10 DIGIT        = %x30-39 ; 0-9
```

Przykład 2.4: Przykład gramatyki w ABNF

W tym przykładzie “email-address” składa się z “local-part” (część lokalna) oraz domain (domena). Element “local-part” może zawierać jeden lub więcej znaków “atext”, a “domain” składa się z jednej lub więcej “sub-domain”, oddzielonych kropkami. Znak “atext” definiuje zestaw dozwolonych znaków w adresie email.

Syntaktyka poszczególnych elementów notacji jest podana w tabeli poniżej 2.3.

Element	Opis
symbol = ...	Definicja reguły
symbol lub <symbol>	Symbol nieterminalny
'...' lub "..."	Symbol terminalny
,	Operator konkatencji
/	Operator alternatywy
*1...lub [...]	Operator opcjonalności
*...	Operator powtórzenia (0 lub więcej)
1*...	Operator powtórzenia (1 lub więcej)
n*n...	Operator powtórzenia (n powtórzeń)
n*m...	Operator powtórzenia (od n do m powtórzeń)
(...)	Grupowanie
; ...	Komentarz

Tabela 2.3: Syntaktyczne elementy języka ABNF

### 2.3.4 ANTLR

ANTLR4 (ANother Tool for Language Recognition, version 4) to potężne narzędzie do generowania lekserów, parserów i drzew parsowania. Zostało stworzone przez Terence’a

Parr’a i jest używane do obsługi szeregów języków, od języków programowania po języki opisu danych.

Jest to narzędziem, które bierze jako wejście specyfikację gramatyki w specjalnym formacie i produkuje kod w wybranym języku programowania, który potrafi analizować ciągi symboli zgodnie z tą gramatyką. Można go użyć do tworzenia lekserów [2.2.1], parserów [2.2.2] i drzew parsowania [2.18].

Wykorzystuje się je do tworzenia i obsługi języków. Może to obejmować tworzenie kompilatorów i interpreterów dla języków programowania, tworzenie analizatorów dla języków opisu danych, a nawet tworzenie narzędzi do manipulowania kodem źródłowym dla celów refaktoryzacji lub analizy statycznej.

Gramatyki w ANTLR są zapisywane w specjalnym formacie, który zawiera produkcje gramatyczne dla nieterminali gramatyki. Produkcja gramatyczna składa się z lewej strony, która jest nieterminalem, i prawej strony, która jest sekwencją terminali i nieterminali. Oto przykład prostego pliku gramatyki ANTLR:

```
1  grammar Hello ;
2
3  expression : term (('+' | '-') term)* ;
4  term       : factor (('*' | '/') factor)* ;
5  factor     : number | '(' expression ')' ;
6  number     : DIGIT+ ;
7
8  DIGIT      : [0-9] ;
```

#### Przykład 2.5: Przykład gramatyki napisanej w ANTLR

W powyższym przykładzie mamy cztery reguły: `expression`, `term`, `factor` i `number`. Pierwszą regułą jest `expression`, opisuje ona wyrażenie arytmetyczne i składa się z `term` oraz opcjonalnych powtórzeń operatorów dodawania lub odejmowania. Reguła `term` opisuje człon wyrażenia i składa się z `factor` oraz opcjonalnych powtórzeń operatorów mnożenia lub dzielenia. Wynika to z kolejności wykonywania działań w arytmetyce liczb. Reguła `factor` opisuje liczby lub wyrażenia otoczone nawiasami. Reguła `number` opisuje pojedynczą cyfrę lub ciąg cyfr.

Wszystkie potrzebne elementy syntaktyczne tego języka zastały zawarte w tabeli poniżej [2.4].

Element	Opis
symbol : ...;	Definicja reguły
symbol	Symbol nieterminalny
'...'	Symbol terminalny
(spacja)	Operator konkatencji
	Operator alternatywy
...?	Operator opcjonalności
...*	Operator powtórzenia (0 lub więcej)
...+	Operator powtórzenia (1 lub więcej)
(...)	Grupowanie
// ...lub /* ... */	Komentarz
;	Znak końca

Tabela 2.4: Syntaktyczne elementy języka ANTLR

Oprócz dostarczenia notacji ANTLR dostarcza także zbiór narzędzi pozwalających generować kod odpowiadający za obsługę języka. Tworzenie języka w antlr skład się z takich kroków jak opisanie gramatyki w notacji ANTLR, generowanie kodu parsera oraz leksera za pomocą specjalnego narzędzia, oraz wykorzystanie leksera i parsera.

Pierwszym krokiem jest napisanie pliku gramatyki, który opisuje gramatykę języka, który chcesz analizować. Plik gramatyki zawiera definicje dla wszystkich nieterminali w gramatyce wraz z ich produkcjami gramatycznymi. Jako przykładową gramatykę wykorzystajmy przykład wyżej [2.5].

Generowanie kodu z pliku gramatyki to zazwyczaj kwestia wywołania narzędzia ANTLR4 z odpowiednimi argumentami z poziomu wiersza poleceń. Na przykład, jeśli używamy języka Java i mamy już plik gramatyki Hello.g4, jesteśmy w stanie wygenerować kod za pomocą polecenia przedstawionego na poniższym przykładzie [2.6]. Polecenie to generuje cztery pliki Java: "HelloLexer.java", "HelloParser.java", "HelloListener.java" i "HelloBaseListener.java".

```
1 antlr4 Hello.g4
```

Przykład 2.6: Polecenie generujące kod ANTLR4 dla pliku Hello.g4

Wygenerowane klasy można teraz używać do analizy ciągów symboli zgodnie z podaną przykładową gramatyką. Na przykład, możemy użyć wygenerowanego leksera i parsera w następujący sposób:

```
1 import org.antlr.v4.runtime.*;
2 import org.antlr.v4.runtime.tree.*;
3
4 public class HelloRunner {
5     public static void main(String[] args) throws Exception {
6         CharStream input = CharStreams.fromStream(System.in);
7         HelloLexer lexer = new HelloLexer(input);
8         CommonTokenStream tokens = new CommonTokenStream(lexer);
9         HelloParser parser = new HelloParser(tokens);
10        ParseTree tree = parser.expression();
11        System.out.println(tree.toStringTree(parser));
```

```

12     }
13 }

```

### Przykład 2.7: Przykład wykorzystania wygenerowanego kodu ANTLR w Javie

Do pełni działania parsera i leksera w języku programowania, potrzebne jest środowisko uruchomieniowe. Środowisko te oparte jest o bibliotekę dostarczaną przez sam zespół rozwijający technologię. Na czerwiec 2023 roku, ANTLR4 obsługuje środowiska uruchomieniowe takich języków jak: Java, JavaScript, Python, C++, TypeScript, Go, Swift, PHP oraz Dart

## 2.3.5 Porównanie elementów syntaktycznych notacji

Opis	BNF	EBNF	ABNF	ANTLR
Definicja reguły	<symbol> ::= ...	symbol = ...	symbol = ...	symbol : ...;
Symbol nieterminalny	<symbol>	symbol	symbol lub <symbol>	symbol
Symbol terminalny	...	'...' lub "..."	'...' lub "..."	'...'
Operator katenacji	(spacja)	,	,	(spacja)
Operator alternatywy			/	
Operator opcjonalności		[...]	*1...lub [...]	...?
Operator powtórzenia (0 lub więcej)		{...}	*...	...*
Operator powtórzenia (1 lub więcej)		{...}-	1*...	...+
Operator powtórzenia ( <i>n</i> powtórzeń)		n*...	n*n...	
Operator powtórzenia (od <i>n</i> do <i>m</i> powtórzeń)			n*m...	
Grupowanie		(...)	(...)	(...)
Komentarz		(* ... *)	; ...	// ...lub /* ... */
Znak końca		;		;

Tabela 2.5: Porównanie notacji języków do definiowania gramatyki

# Rozdział 3

## Implementacja

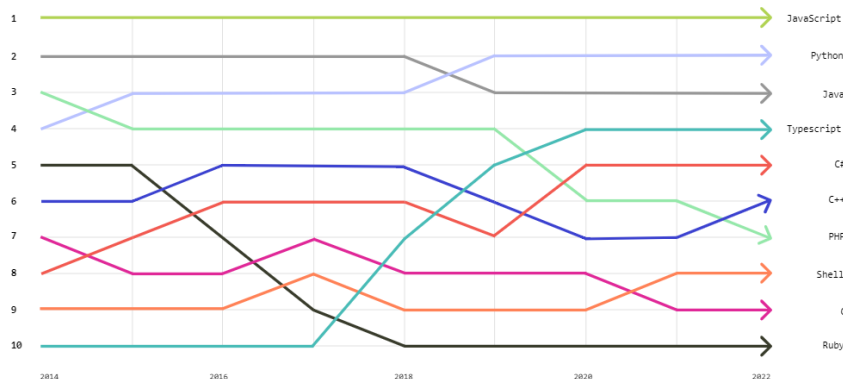
### 3.1 Wykorzystane technologie

#### TypeScript

TypeScript to język programowania, który jest rozbudowanym supersetem języka JavaScript. Zapewnia on statyczne typowanie oraz rozszerza sam język JavaScript o dodatkowe funkcjonalności. Pierwsza publikacja TypeScript przypada na rok 2012, została ona opublikowana przez Microsoft.

Głównym założeniem języka jest dostarczenie możliwości definiowania typów oraz analiza typowania statycznie i dynamicznie. Typowanie statyczne polega na dostarczeniu informacji o typach i weryfikacja ich poprawności na poziomie IDE. Dynamiczne typowanie polega na weryfikacji zgodności typów w trakcie wykonywania programu.

TypeScript staje się coraz bardziej popularną technologią, co pokazują dane przedstawione między innymi przez GitHub. Poniższy wykres 3.1 przedstawia wzrost ilości repozytoriów wykorzystujących dany język programowania. Na tym wykresie jesteśmy w stanie zauważyć wzrosty oraz spadki konkretnych języków programowania w ciągu ośmiu lat. Możemy zauważyć, że wykres przedstawia znaczący wzrost zainteresowania językiem TypeScript.



Rysunek 3.1: Wykres przedstawiający wzrost wykorzystania języków programowania

Aktualnie język ten znajduje się na czwartym miejscu najpopularniejszych języków serwisu GitHub. W kolejnych latach przewidywany jest dalszy wzrost oraz realne jest to, że TypeScript zastąpi JavaScript na miejscu najpopularniejszego języka programowania. Wynika to z faktu, iż programowanie w tym języku jest znacznie wydajniejsze niż w normalnym JavaScriptcie oraz faktu, że kod napisany w TypeScript jest znacznie łatwiejszy w utrzymaniu.

Język ten dostarcza kilka podstawowych typów, które możemy wykorzystać. Tabela 3.1 przedstawia te typy wraz z ich opisem i przykładem.

Typ	Opis	Przykład
<i>number</i>	Reprezentuje wartość numeryczną	let a: number = 10;
<i>string</i>	Reprezentuje ciąg znaków	let a: string = "example";
<i>boolean</i>	Reprezentuje wartość logiczną	let isA: boolean = false;
<i>array</i>	Reprezentuje tablicę danych	let a: number[] = [];
<i>tuple</i>	Krotki, czyli tablice o stałej długości	let a: [number, number] = [0, 10];
<i>enum</i>	Zdefiniowane przez użytkownika wartości wyliczeniowe	let a: AnEnum = AnEnum.A;
<i>any</i>	Wyłącza sprawdzanie typów dla określonego pola, może przyjąć dowolną wartość	let a: any[] = ["test", 1, false];
<i>unknown</i>	Przyjmuje dowolną wartość, lecz nie wyłącza całkowicie sprawdzania typów danego pola	let a: unknown[] = ["test", 1, false];

Tabela 3.1: Podstawowe typy danych dostępne w TypeScript

Syntaktyka języka TypeScript jest oparta syntaktykę języka JavaScript. Co za tym idzie, koda napisany w tym języku jest niemalże tożsamy z JavaScript. Język TypeScript zatem musi obsługiwać wszystkie możliwe mechanizmy występujące w rozszerzonym języku.

Aby lepiej przedstawić syntaktykę języków, zostaną przedstawione dwie aplikacje. Aplikacje będą implementowały dokładnie tę samą logikę, z tym że kod pierwszej będzie napisany w TypeScript, a drugi w JavaScript.

```

1  const nodes = [
2    {
3      "id": "4a603fb3-4e73-4548-b39c-965d9937b8da",
4      "label": "A",
5      "position": [
6        100,
7        100
8      ]
9    },
10   {
11     "id": "e317e8a9-e5de-4380-89cb-972b4d597cdd",
12     "label": "B",
13     "position": [
14       500,
15       100

```

```

16     ]
17   },
18   ...
19 ];
20
21 const edges = [
22   {
23     "id": "9d6e02cb-580f-400c-beab-5d72ba9f4d2c",
24     "source": "4a603fb3-4e73-4548-b39c-965d9937b8da",
25     "target": "e317e8a9-e5de-4380-89cb-972b4d597cdd",
26     "weight": 630
27   },
28   {
29     "id": "115e341c-786a-452e-a9ee-f29cbbea1940",
30     "source": "4a603fb3-4e73-4548-b39c-965d9937b8da",
31     "target": "a404d3ed-5376-4e7d-8418-14305ea73f3f",
32     "weight": 760
33   },
34   {
35     "id": "cf4eca1b-14f1-4219-a207-dd839764df1b",
36     "source": "4a603fb3-4e73-4548-b39c-965d9937b8da",
37     "target": "0bafee74-ec49-4979-afd3-a03e213a512c",
38     "weight": 540
39   },
40   ...
41 ];

```

Przykład 3.1: Dane wejściowe algorytmu Dijkstry w JavaScript

Prezentację różnic w składni języków zaczniemy od przedstawienia danych wejściowych do algorytmu. Jest to ważne ze względu na format danych, które możemy wprowadzić. Przykład 3.1 przedstawia przykładowe dane wejściowe i ich formę. W przypadku powszechnego JavaScriptu mamy możliwość wprowadzenia dowolnych danych wejściowych w obiekcie. Możemy także zrobić literówkę czy wprowadzić dane w złym formacie. Przykładem takiego pola jest na przykład “position” w węzłach. Wiedząc to, jesteśmy w stanie powiedzieć, że brak typowania sprawia, że kod jest znacznie trudniejszy do utrzymania.

```

1 class Dijkstra {
2   constructor(nodes, edges) {
3     this.nodes = {};
4     this.edges = {};
5     this.sourceNodeId = undefined;
6     this.isPathsAnalyzed = false;
7
8     for (const node of nodes) {
9       const relatedEdges = edges
10        .filter(
11          (edge) =>
12            edge.source === node.id
13            || edge.target === node.id
14        )
15        .map((edge) => edge.id);
16
17       this.nodes[node.id] = {

```

```

18         ...node,
19         edges: relatedEdges,
20         dist: Infinity,
21         prev: undefined,
22     };
23 }
24
25 for (const edge of edges) {
26     this.edges[edge.id] = edge;
27 }
28 }
29
30 analyzePathsFromSource() {
31     if (this.sourceNodeId === undefined) {
32         throw new Error("Select source node first.");
33     }
34
35     if (this.isPathsAnalyzed) {
36         for (const nodeId in this.nodes) {
37             this.nodes[nodeId].dist = Infinity;
38             this.nodes[nodeId].prev = undefined;
39         }
40
41         this.isPathsAnalyzed = false;
42     }
43
44     this.nodes[this.sourceNodeId].dist = 0;
45     const visited = [];
46
47     while (visited.length < Object.keys(this.nodes).length) {
48         const currentNode = this.findMinDistance(visited);
49
50         if (currentNode === undefined) break;
51
52         visited.push(currentNode.id);
53
54         for (const currentEdgeId of currentNode.edges) {
55             const currentEdge = this.edges[currentEdgeId];
56             const distance = currentEdge.weight;
57             const totalDistance = currentNode.dist + distance;
58             const neighbor = this.nodes[currentEdge.target];
59
60             if (totalDistance < neighbor.dist) {
61                 neighbor.dist = totalDistance;
62                 neighbor.prev = currentNode;
63             }
64         }
65     }
66
67     this.isPathsAnalyzed = true;
68 }
69
70 getShortestPathTo(targetNodeId) {
71     if (this.sourceNodeId === undefined || !this.isPathsAnalyzed) {

```



```

72     throw new Error("Missing source node or graph not analyzed.");
73 }
74
75 let current = this.nodes[targetNodeId];
76 const nodes = [];
77 const edges = [];
78
79 do {
80     nodes.push(current.id);
81
82     const currentEdge = current.edges
83         .map((edgeId) => this.edges[edgeId])
84         .find((edge) => edge.source === current?.prev?.id);
85
86     if (currentEdge === undefined) {
87         break;
88     }
89
90     edges.push(currentEdge.id);
91     current = current.prev;
92 } while (current !== undefined);
93
94 return {
95     from: this.sourceNodeId,
96     to: targetNodeId,
97     nodes,
98     edges
99 };
100 }
101
102 findMinDistance(visited) {
103     let minDistance = Infinity;
104     let minNodeId = undefined;
105
106     for (const nodeId in this.nodes) {
107         const node = this.nodes[nodeId];
108
109         if (!visited.includes(nodeId) && node.dist <= minDistance) {
110             minDistance = node.dist;
111             minNodeId = nodeId;
112         }
113     }
114
115     return minNodeId !== undefined
116         ? this.nodes[minNodeId]
117         : undefined;
118 }
119
120 setSourceNode(sourceNodeId) {
121     this.sourceNodeId = sourceNodeId;
122 }
123 }
124
125 const dijkstra = new Dijkstra(nodes, egdes);

```

```

126 dijkstra.setSourceNode(nodes[0].id);
127 dijkstra.analysePathsFromSource();
128 const result = dijkstra.getShortestPathTo(nodes[5].id);
129
130 console.log(result);

```

### Przykład 3.2: Przykład implementacji algorytmu Dijkstry w JavaScript

Przykład 3.2 przedstawia implementację algorytmu Dijkstry w języku JavaScript. Implementacja opiera się o deklarację klasy. Klasa ta zawiera cztery pola opisujące proces wyszukiwania najszybszej trasy na podstawie grafu. Odpowiednio, pierwsze pole zawiera zmodyfikowane węzły, drugie pole zawiera krawędzie grafu, trzecie pole oznacza identyfikator źródłowego węzła oraz ostatnie pole jest to flaga, która informuje nas czy graf został przeanalizowany. Podstawą działania algorytmu Dijkstry jest przeszukiwanie trasy z wejściowego węzła do wszystkich innych. Należy wziąć pod uwagę, że przetwarzany graf jest grafem ważonym oraz grafem skierowanym. Metody zaimplementowane w klasie odpowiadają za kolejne etapy działania algorytmu.

Przedstawienie implementacji tego algorytmu zaczniemy od przedstawienia typowania, które będzie wykorzystywane na przestrzeni implementacji. Typowanie to pomoże nam wymusić dane wejściowe w wymuszonej formie, a także przedstawić formę zwracanych danych.

```

1 export type Position = [number, number];
2
3 export interface Node {
4   id: string;
5   label: string;
6   position: Position;
7 }
8
9 export type ExtendedNode = Node & {
10   edges: string[];
11   dist: number;
12   prev?: ExtendedNode;
13 };
14
15 export interface Edge {
16   id: string;
17   source: string;
18   target: string;
19   weight: number;
20 }
21
22 export interface Path {
23   from: string;
24   to: string;
25   nodes: string[];
26   edges: string[];
27 }

```

### Przykład 3.3: Deklaracja typów dla algorytmu Dijkstry w TypeScript

W przykładzie 3.3 deklarujemy pięć typów, które pomogą nam opisać implementację. Pierwszym i zarazem najprostszym typem jest deklaracja typu “Position”, który reprezen-

tuje dwójkę uporządkowaną. Kolejne typ są powiązane, jest to interfejs “Node” oraz unia “ExtendedNode”.

Interfejs w TypeScript jest to abstrakcyjna wersja obiektu opisana polami oraz metodami o określonej charakterystyce typowania. Pełnią one bardzo ważny element tego języka i są bardzo powszechnie stosowane. Ten typ definicji typowania nie pozwala, na wprowadzenie jakiejkolwiek logiki. Finalnie, sam interfejs po przetłumaczeniu na JavaScript jest usuwany. Pełni on więc wyłącznie funkcję weryfikacji typowania obiektów w kodzie.

Typ “ExtendedNode” jest unią, czyli typem obiektu rozszerzonym o określone pola. Dzięki zastosowaniu takiego mechanizmu otrzymujemy typ “Node” rozszerzony o opcjonalne pola “edges”, “dist”, “prev”. Jest to bardzo przydatny element języka pokazujący prawdziwą naturę i modalność dostarczaną przez ten język.

Warto zwrócić także uwagę na pole “prev” podany w uni “ExtendNode”. Unia ta zawiera pole oznaczone znakiem zapytania (“prev?”). Jest to mechanizm wprowadzony przez TypeScript, który umożliwia określenie czy pole jest opcjonalne, czy nie. Efektem tego stanie się nic innego jak stworzenie definicji “prev: string | undefined”. Operator opcjonalności możemy również wykorzystać w definicji metod oraz w interfejsach.

```
1 export class Dijkstra {
2   private readonly nodes: Record<string, ExtendedNode>;
3   private readonly edges: Record<string, Edge>;
4
5   private sourceNodeId?: string;
6   private isPathsAnalyzed: boolean = false;
7
8   public constructor(nodes: Node[], edges: Edge[]) {
9     this.nodes = {};
10    this.edges = {};
11
12    for (const node of nodes) {
13      const relatedEdges = edges
14        .filter(
15          (edge) =>
16            edge.source === node.id
17            || edge.target === node.id
18        )
19        .map((edge) => edge.id);
20
21      this.nodes[node.id] = {
22        ...node,
23        edges: relatedEdges,
24        dist: Infinity,
25        prev: undefined
26      };
27    }
28
29    for (const edge of edges) {
30      this.edges[edge.id] = edge;
31    }
32  }
33
34  public analyzePathsFromSource() {
```

```

35     if (this.sourceNodeId === undefined) {
36         throw new Error("Select source node first.");
37     }
38
39     if (this.isPathsAnalyzed) {
40         for (const nodeId in this.nodes) {
41             this.nodes[nodeId].dist = Infinity;
42             this.nodes[nodeId].prev = undefined;
43         }
44
45         this.isPathsAnalyzed = false;
46     }
47
48     this.nodes[this.sourceNodeId].dist = 0;
49     const visited: string[] = [];
50
51     while (visited.length < Object.keys(this.nodes).length) {
52         const currentNode = this.findMinDistance(visited);
53
54         if (currentNode === undefined) break;
55
56         visited.push(currentNode.id);
57
58         for (const currentEdgeId of currentNode.edges) {
59             const currentEdge = this.edges[currentEdgeId];
60             const distance = currentEdge.weight;
61             const totalDistance = currentNode.dist + distance;
62             const neighbor = this.nodes[currentEdge.target];
63
64             if (totalDistance < neighbor.dist) {
65                 neighbor.dist = totalDistance;
66                 neighbor.prev = currentNode;
67             }
68         }
69     }
70
71     this.isPathsAnalyzed = true;
72 }
73
74 public getShortestPathTo(targetNodeId: string): Path {
75     if (this.sourceNodeId === undefined || !this.isPathsAnalyzed) {
76         throw new Error("Missing source node or graph not analyzed.");
77     }
78
79     let current: ExtendedNode | undefined = this.nodes[targetNodeId];
80
81     const nodes: string[] = [];
82     const edges: string[] = [];
83
84     do {
85         nodes.push(current.id);
86
87         const currentEdge = current.edges
88             .map((edgeId) => this.edges[edgeId])

```

```

89         .find((edge) => edge.source === current?.prev?.id);
90
91         if (currentEdge === undefined) {
92             break;
93         }
94
95         edges.push(currentEdge.id);
96         current = current.prev;
97     } while (current !== undefined);
98
99     return {
100         from: this.sourceNodeId,
101         to: targetNodeId,
102         nodes,
103         edges
104     };
105 }
106
107 private findMinDistance(visited: string[]): ExtendedNode | undefined
108 {
109     let minDistance = Infinity;
110     let minNodeId: string | undefined = undefined;
111
112     for (const nodeId in this.nodes) {
113         const node = this.nodes[nodeId];
114
115         if (!visited.includes(nodeId) && node.dist <= minDistance) {
116             minDistance = node.dist;
117             minNodeId = nodeId;
118         }
119     }
120
121     return minNodeId !== undefined
122         ? this.nodes[minNodeId]
123         : undefined;
124 }
125
126 public setSourceNode(sourceNodeId: string) {
127     this.sourceNodeId = sourceNodeId;
128 }
129
130 const dijkstra = new Dijkstra(nodes, edges);
131 dijkstra.setSourceNode(nodes[0].id);
132 dijkstra.analyzePathsFromSource();
133 const result = dijkstra.getShortestPathTo(nodes[5].id);
134
135 console.log(result);

```

### Przykład 3.4: Przykład implementacji algorytmu Dijkstry w TypeScript

Na pierwszy rzut oka, składnia kodu przedstawiona na przykładzie 3.4 jest bardzo zbliżona do tej z przykładu 3.2. Pojawiły się tam jednak pewne elementy, których w zwykłym JavaScriptcie nie jesteśmy w stanie wykorzystać.

Najważniejszym elementem języka TypeScript jest anotacja typu. Anotację typu możemy wykorzystać do określenia typu zmiennej, typu parametru w funkcji, a także do zwracanego typu z metody lub funkcji. Jest to pierwsze rozszerzenie powszechnej syntaktyki JavaScript. Anotacje typowania zaczynają się od znaku “:”, po czym podajemy konkretny oczekiwany typ. Do anotowania zmiennych i innych elementów możemy wykorzystać powszechne typy podane w tabeli 3.1 lub typy złożone zdefiniowane przez nas.

Interesujące może zadawać się również rozszerzenie podstawowych funkcjonalności klas. Pierwszym elementem, który możemy zauważyć, są operatory hermetyzacji. Jesteśmy w stanie określić, które pole lub która metoda będzie dostępna poza implementacją.

Ciekawym elementem jest także słowo kluczowe “readonly”. Umożliwia ono określenie, które pole może zostać zmienione. Jest to umożliwienie wykorzystania deklaracji “const” w obiektach. Oznacza to, że tak opisane pole może być tylko i wyłącznie modyfikowane w przypadku pól złożonych (opisanych referencją), zmienne opisane typem prostym są możliwe tylko do odczytu.

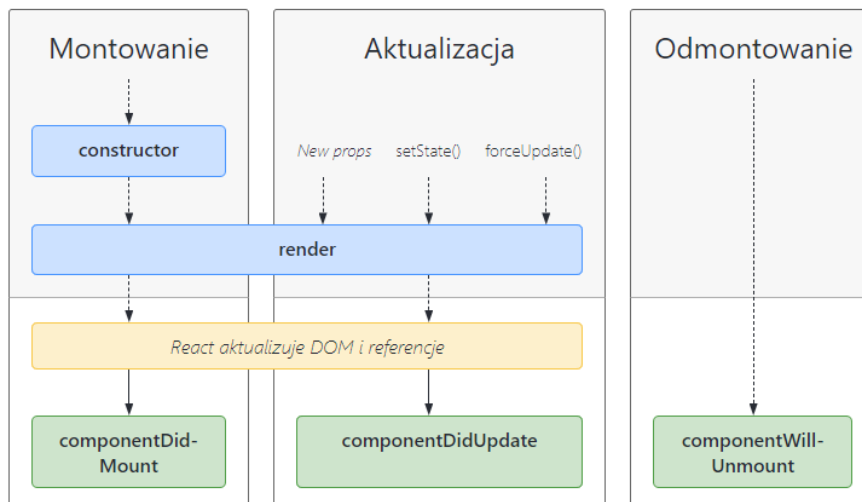
TypeScript jest bardzo dynamicznie rozwijającym się językiem, co sprawia, że jest coraz chętniej wykorzystywany w przez różne firmy na rynku. Obecnie spora część bibliotek stworzonych pod JavaScript, posiada także odpowiednik lub samo typowanie w TypeScript. W tej pracy zostały przedstawione podstawowe elementy tego języka. Wprowadza on znacznie więcej elementów, z którymi warto zapoznać się na stronie [**TypeScriptTooling**].

## React

React jest biblioteką rozszerzającą JavaScript o elementy szablonów w postaci fragmentów znaczników. Główną cechą wykorzystania Reacta jest podejście do rozbijania kodu w małe fragmenty będące komponentami. Każdy komponent ma swoją implementację oraz może zawierać określoną logikę.

Wszystkie komponenty mają określony cykl życia. Cykl życia odpowiada za opisanie sposobu, w jakiej formie odbywa się renderowanie komponentu. Pokazuje on także metody, które mogą zostać wykorzystane w komponencie. Na przełomie lat cykl życia przechodził różne transformacje do momentu aż osiągnął stan na rysunku 3.2.

Cały cykl podzielony jest na trzy kategorie. Pierwszą kategorią jest “Montowanie”, czyli moment, w którym komponent wykonuje operację “componentDidMount”. Jest to zdarzenie, które wykonuje się zaraz po pierwszym wyrenderowaniu komponentu. Wykorzystuje się je do wywoływania akcji jak ładowanie danych lub uruchamianie procesów asynchronicznych. Kolejną kategorią jest “Aktualizowanie”, czyli moment, w którym następują zmiany na wejściu komponentu lub w stanie komponentu. Aktualizowanie może być wywołane przez zmianę wartości w parametrze komponentu, w zależności czy komponent nasłuchuje na zmianę określonego parametru. Wywołane może być także w momencie, kiedy wywołamy “forceUpdate()” lub zmienimy jakąś wartość w stanie komponentu. Po wyrenderowaniu zostaje wywołana metoda “componentDidUpdate”. Ostatnią kategorią jest “Odmontowanie”. Jest to element, który odpowiada za wykrywanie, kiedy komponent przestaje być renderowany na stronie. W tym momencie jest wywoływana metoda “componentDidUnmount”.



Rysunek 3.2: Cykl życia komponentu w React

Wyróżniamy różne typy komponentów, takie jak: komponenty funkcyjne, komponenty klasowe, komponenty strukturalne.

Komponenty funkcyjne reprezentowane są za pomocą podejścia funkcyjnego wprowadzonego w nowoczesnych wersjach JavaScript. Ich założenie jest bardzo proste, komponent przyjmuje parametry jak normalna funkcja, po czym zwraca strukturę JSX [3.1]. Głównym elementem tego typu komponentu jest fakt, iż nie posiada on stanu (state) oraz metod cyklu życia, które możemy rozszerzać. Stan aplikacji, jak i cykl życia jest osiągalne dzięki wykorzystaniu haczyków (hook). Są one najczściej używane do tworzenia prostych komponentów, lecz z biegiem czasu i ich ewolucją stają się one powszechnie wykorzystywane. Stanowią one lepszą alternatywę do komponentów klasowych ze względu na mniejszą ilość wbudowanej logiki i wydajność.

### Definicja 3.1. *JSX*

*JSX (JavaScript XML) to składnia, która pozwala na tworzenie hierarchicznych struktur elementów w języku JavaScript, które reprezentują elementy interfejsu użytkownika. Składnia JSX przypomina HTML, ale jest zintegrowana z kodem JavaScript, co umożliwia tworzenie dynamicznych i reaktywnych aplikacji.*

Komponenty klasowe reprezentowane są za pomocą podejścia klasowego wbudowanego w JavaScript. Posiadają one wbudowany stan (state) oraz metody cyklu życia komponentu, które możemy nadpisywać. Ustawianie stanu początkowego, jak i definiowanie parametrów jest oparte o konstruktor. Są to komponenty znacznie bardziej zaawansowane niż komponenty funkcyjne i były wcześniej podstawowym podejściem do tworzenia komponentów.

Komponenty strukturalne są typem komponentów funkcyjnych, które w założeniu nie posiadają żadnej logiki. Głównym założeniem tego typu komponentów jest pobranie danych z parametrów i wyświetlenie ich. Z założenia, komponenty tego typu nie powinny posiadać nawet złożonej logiki renderowania.

Ważnym elementem, który został już nadmieniony, są haczyki (hooks). Przede wszystkim dostarczają one możliwość wykorzystania stanu (state) w komponentach nieklasowych.

Wykorzystujemy do tego haczy “useState”. Pozwalają one także wykonywać operacje cyklu życia oraz operacje nasłuchujące na zmianie wartości parametrów i zmiennych stanu. Wykorzystujemy do tego haczyk “useEffect”. Wraz z użyciem “useContext” możemy udostępniać i odczytywać dane zapisane globalnie w aplikacji. Możemy na przykład zapisywać tam dane, w jakim trybie wyświetlamy stronę, czy jest to tryb ciemny, czy tryb jasny. Ostatnim kluczowym haczykiem jest “useRef”, umożliwia on bezpośredni dostęp do elementów węzła DOM.

Możemy także tworzyć własne haczyki. Dzięki takiemu rozwiązaniu jesteśmy w stanie wyciągać część logiki ze skomplikowanych komponentów. Każdy haczyk musi zaczynać się od słowa “use” i reprezentowany jest przez funkcję. Funkcja może przyjmować parametry oraz zwracać określone dane lub metody.

W celu lepszego pokazania możliwości Reacta przedstawimy przykład aplikacji napisanej w nim. Rozwiniemy algorytm Dijkstry wykorzystaniu w omówieniu TypeScript i stworzymy reprezentację graficzną grafu. Zostaną wykorzystane typowania zdefiniowane w przykładzie 3.3.

```
1  const rootElement = document.getElementById("root")!;  
2  const root = ReactDOM.createRoot(rootElement);  
3  
4  root.render(  
5    <React.StrictMode>  
6      <Graph />  
7    </React.StrictMode>  
8  );
```

### Przykład 3.5: Początek aplikacji w React

Przykład 3.5 przedstawia podłączenie biblioteki React do elementu DOM o identyfikatorze “root”. Pozwala to na określenie, w którym elemencie będzie zagnieżdżona aplikacja. Jest to bardzo elastyczne, ponieważ możemy wyrenderować aplikację Reactową nawet w szablonach po stronie serwera aplikacji.

```
1  const Graph: React.FC = () => {  
2    const canvasRef = useRef<HTMLCanvasElement | null>(null);  
3    const [nodes, setNodes] = useState<Node[]>([]);  
4    const [edges, setEdges] = useState<Edge[]>([]);  
5  
6    function addNodes(...nodesData: [string, Position][]): Node[] {  
7      const newNodes = nodesData.map(  
8        ([label, position]): Node => ({  
9          id: uuid(),  
10         label,  
11         position  
12       })  
13     );  
14  
15     setNodes([...newNodes]);  
16     return newNodes;  
17   }  
18  
19   function addEdges(...nodes: [Node, Node][]): Edge[] {  
20     const newEdges = nodes.map(  
21       ([node1, node2]): Edge => ({  
22         id: uuid(),  
23         source: node1.id,  
24         target: node2.id,  
25         weight: 1  
26       })  
27     );  
28     return newEdges;  
29   }  
30  
31   return (  
32     <div>  
33       <Canvas ref={canvasRef} />  
34     </div>  
35   );  
36 }
```



```

21     ([source, target]): Edge => {
22         const weigth = Math.sqrt(
23             (source.position[0] + target.position[0]) ** 2 +
24             (source.position[1] + target.position[1]) ** 2
25         );
26
27         return {
28             id: uuid(),
29             source: source.id,
30             target: target.id,
31             weight: parseFloat(weigth.toPrecision(2))
32         };
33     }
34 );
35
36 setEdges([...newEdges]);
37 return newEdges;
38 }
39
40 useEffect(() => {
41     const addedNodes = addNodes(
42         ["A", [100, 100]],
43         ["B", [500, 100]],
44         ["C", [200, 600]],
45         ...
46     );
47
48     const addedEdges = addEdges(
49         [addedNodes[0], addedNodes[1]],
50         [addedNodes[0], addedNodes[2]],
51         [addedNodes[0], addedNodes[5]],
52         ...
53     );
54 }, []);
55
56 useEffect(() => {
57     const canvas = canvasRef.current;
58
59     if (!canvas) return;
60
61     const ctx = canvas.getContext("2d");
62     if (!ctx) return;
63
64     ctx.clearRect(0, 0, canvas.width, canvas.height);
65     ctx.fillStyle = "white";
66     ctx.fillRect(0, 0, canvas.width, canvas.height);
67
68     edges.forEach((edge) => {
69         const sourceNode = nodes.find((node) => node.id === edge.
source);
70         const targetNode = nodes.find((node) => node.id === edge.
target);
71
72         if (!sourceNode || !targetNode) {

```

```

73     return;
74 }
75
76     const sourceX = sourceNode.position[0];
77     const sourceY = sourceNode.position[1];
78     const targetX = targetNode.position[0];
79     const targetY = targetNode.position[1];
80
81     if (sourceNode && targetNode) {
82         ctx.beginPath();
83         ctx.strokeStyle = "black";
84         ctx.moveTo(sourceX, sourceY);
85         ctx.lineTo(targetX, targetY);
86         ctx.stroke();
87
88         const textX = (sourceX + targetX) / 2;
89         const textY = (sourceY + targetY) / 2;
90         ctx.fillStyle = "red";
91         ctx.font = '16px 'Arial'';
92         ctx.fillText(edge.weight.toString(), textX, textY);
93     }
94 });
95
96     nodes.forEach((node) => {
97         const nodeX = node.position[0];
98         const nodeY = node.position[1];
99
100         ctx.beginPath();
101         ctx.arc(nodeX, nodeY, 20, 0, 2 * Math.PI);
102         ctx.fillStyle = "gray";
103         ctx.fill();
104
105         ctx.fillStyle = "white";
106         ctx.textAlign = "center";
107         ctx.textBaseline = "middle";
108         ctx.font = '12px 'Arial'';
109         ctx.fillText(node.label, nodeX, nodeY);
110     });
111 }, [nodes, edges]);
112
113     return (
114         <div>
115             <h1>Graf wa ony skierowany</h1>
116             <canvas
117                 ref={canvasRef}
118                 width={800}
119                 height={800}
120                 style={{ border: "1px solid black" }}
121             />
122         </div>
123     );
124 };

```

Przykład 3.6: Podstawowe rysowanie grafu w React

Przykład 3.6 przedstawia podstawową implementację rysowania na podstawie canvas. Komponent posiada dwie zmienne w stanie (state), które odpowiadają za przechowywanie węzłów i wierzchołków. Dodawanie wierzchołków jest dosyć skomplikowane, ponieważ jako użytkownicy chcemy dodawać dane na podstawie ograniczonych danych. Na przykład w przypadku wierzchołka, jedynymi elementami, jakie chcemy podać, jest pozycja oraz etykieta. Identyfikator będzie wtedy ustawiony na unikalną wartość. Dane są wprowadzane podczas montowania komponenty, dzięki wykorzystaniu haczyka “useEffect”, z pustą listą zależności. Drugi “useEffect” nasłuchuje na zmianę zmiennych “nodes” oraz “edges” ze stanu i aktualizuje obrazek wyświetlany na canvas.

Efekt odpowiedzialny za rysowanie wykorzystuje referencję do elementu “canvas”. Dzięki temu jesteśmy w stanie odwołać się do kontekstu “2D” rysunku. Kontekst pozwala nam na rysowanie określonych elementów.

Kolejnym krokiem jest wprowadzenie interakcji z rysowanym grafem w celu wybrania wierzchołka startowego i końcowego.

```
1  const Graph: React.FC = () => {
2    ...
3    const [sourceNodeId, setSourceNodeId] = useState<string>();
4    const [targetNodeId, setTargetNodeId] = useState<string>();
5    ...
6
7    const findNodeAtPosition = useCallback(
8      (position: Position): Node | undefined => {
9        return nodes.find((node) => {
10          const nodeX = node.position[0];
11          const nodeY = node.position[1];
12          const distancePartX = (position[0] - nodeX) ** 2;
13          const distancePartY = (position[1] - nodeY) ** 2;
14          const distance = Math.sqrt(distancePartX + distancePartY);
15          return distance <= 20;
16        });
17      },
18      [nodes]
19    );
20
21    const handleCanvasClick = useCallback(
22      (event: MouseEvent): void => {
23        event.preventDefault();
24
25        const canvas = canvasRef.current;
26
27        if (!canvas) {
28          return;
29        }
30
31        const rect = canvas.getBoundingClientRect();
32        const clickX = event.clientX - rect.left;
33        const clickY = event.clientY - rect.top;
34        const clickedNode = findNodeAtPosition([clickX, clickY]);
35
36        if (clickedNode === undefined) {
37          return;
```

```

38     }
39
40     if (event.ctrlKey) {
41         setTargetNodeId(clickedNode.id);
42     } else {
43         setSourceNodeId(clickedNode.id);
44     }
45 },
46 [findNodeAtPosition]
47 );
48
49 useEffect(() => {
50     const canvas = canvasRef.current;
51
52     if (!canvas) return;
53     canvas.onclick = handleCanvaClick;
54
55     ...
56 }, [nodes, edges, handleCanvaClick]);
57
58 ...
59 };

```

Przykład 3.7: Dodawanie interakcji do narysowanego grafu w React

Przykład 3.7 przedstawia sposób, w jaki możemy dodać interakcję. Interakcja będzie polegała na dwóch akcjach, kliknięcie węzła oznacza go jako start, kliknięcie węzła z przyciśniętym klawiszem “Ctrl” oznacza go jako koniec ścieżki. W tym celu tworzymy dwie nowe zmienne w stanie komponentu. Zmienne te będą reprezentować wybrane węzły, dzięki przechowywaniu ich identyfikatorów. Kolejnym krokiem jest podpięcie akcji pod zdarzenie kliknięcia rysunku.

Podpięcie akcji kliknięcia zostanie wykonane efekcie odpowiedzialnym za renderowanie rysunku. Wykonanie podpięcia jest możliwa poprzez wykorzystanie referencji do elementu DOM. Możemy podpiąć akcję pod każde zdarzenie elementu, które jest obsługiwane przez DOM. Interakcja z grafem poprzez kliknięcie zostanie oparta o dwie metody: “handleCanvaClick” oraz “findNodeAtPosition”. Obie metody są zaimplementowane z wykorzystaniem haczyka “useCallback”.

Haczyk “useCallback” jest wykorzystywany ze względu na zmienność stanu. Wynika to z faktu, iż zmienna “nodes”, która jest wykorzystywana w metodzie “findNodeAtPosition”, może zmieniać swoją wartość asynchronicznie. Zasada działania tego haczyka polega na reimplementacji metody po zmianie wartości stanu.

Głównym założeniem metody “handleCanvaClick” jest wykrycie pozycji kursora i wybranie tego, który węzeł został kliknięty. Pierwszym krokiem jest estymacja pozycji myszy, a następnie wykorzystanie “findNodeAtPosition” w celu znalezienia węzła w tej lokacji. Po znalezieniu węzła rozpatrywane jest ustawienie wartości węzła startowego lub węzła końcowego.

Metoda “findNodeAtPosition” wyszukuje węzeł na podstawie koordynat kursora. Wyliczany jest dystans od środka węzła do pozycji myszy dla każdego węzła wpisanych w stan.

```

1  const Graph: React.FC = () => {
2    ...

```

```

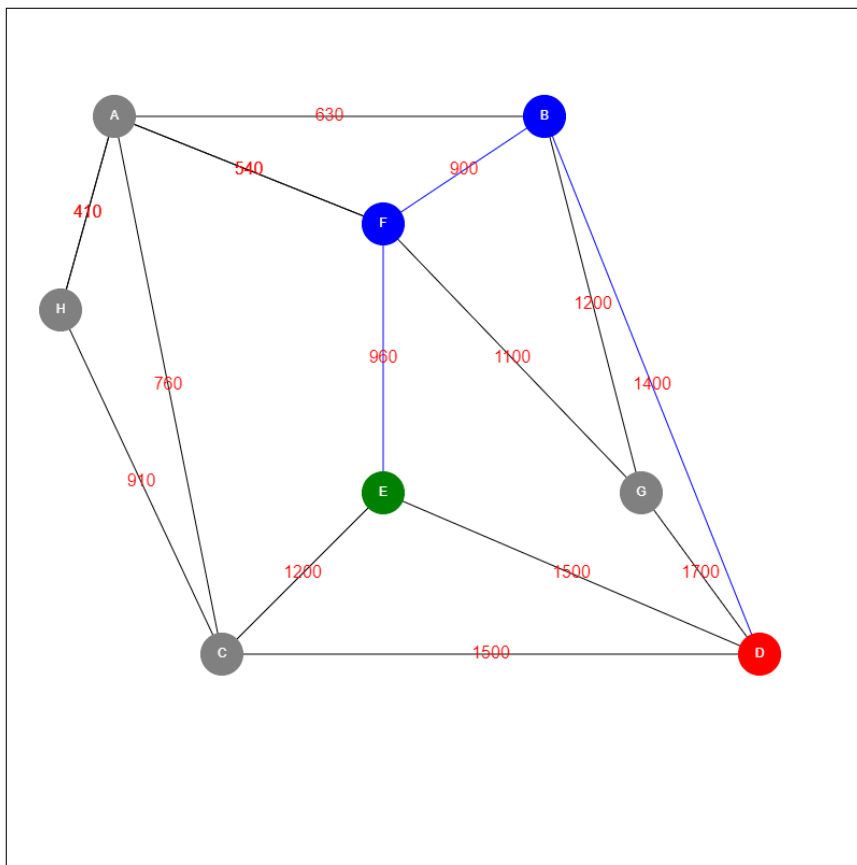
3   const [dijkstra, setDijkstra] = useState<Dijkstra>();
4   const [path, setPath] = useState<Path>();
5   ...
6
7   useEffect(() => {
8     ...
9
10    setDijkstra(new Dijkstra(addedNodes, addedEdges));
11  }, []);
12
13  useEffect(() => {
14    if (dijkstra === undefined || sourceNodeId === undefined) {
15      return;
16    }
17
18    dijkstra.setSourceNode(sourceNodeId);
19    dijkstra.analyzePathsFromSource();
20  }, [dijkstra, sourceNodeId]);
21
22  useEffect(() => {
23    if (
24      dijkstra === undefined ||
25      sourceNodeId === undefined ||
26      targetNodeId === undefined
27    ) {
28      return;
29    }
30
31    const shortestPath = dijkstra.getShortestPathTo(targetNodeId);
32    setPath(shortestPath);
33  }, [dijkstra, sourceNodeId, targetNodeId]);
34
35  ...
36  };

```

Przykład 3.8: Wyszukiwanie trasy na grafie z wykorzystaniem algorytmu Dijkstry w React

Przykład 3.8 przedstawia ostatni element implementacji grafu. Wykorzystujemy w nim zaimplementowany już algorytm Dijkstry. Obiekt implementujący algorytm Dijkstry musimy przechowywać w stanie. Ostatnim elementem jest wprowadzenie zmian do algorytmu w momencie zmiany zaznaczonych węzłów. Jest to wykonane za pomocą haczyków “useEffect”.

### Graf ważony skierowany



Rysunek 3.3: Efekt implementacji aplikacji w React

Rysunek 3.3 przedstawia wygląd zaimplementowanej aplikacji. Reprezentuje ona graf z możliwością zaznaczanie węzłów i wyszukiwania trasy.

## Jest

Jest to framework do testowania kodu JavaScript, który jest powszechnie wykorzystywany w ekosystemie React. Framework ten zapewnia środowisko do tworzenia i uruchamiania testów jednostkowych [3.2], integracyjnych [3.3] i testów komponentów [3.4]. Dostarcza on dużą ilość funkcji umożliwiających weryfikowanie określonych stanów kodu. Udostępniona jest metoda “expect”, która jest podstawą przeprowadzania testów. Testy zaimplementowane w tym frameworku charakteryzują się szybkością i poziomem izolacji testów. Każdy test uruchamiany jest w izolowanym środowisku, wynika z tego, że testy nie mogą mieć na siebie wpływu.

Framework ten dostarcza także narzędzia do tworzenia automatycznych stubów (sztucznych implementacji) i mocków (imitacji obiektów) w celu symulowania zależności i testowania jednostek kodu niezależnie od innych komponentów. Istnieje również wbudowane wsparcie dla testowania asynchronicznych operacji, takich jak zapytania sieciowe lub operacje na

bazie danych. Można używać mechanizmu `async/await` lub funkcji zwrotnych (callbacks) w celu testowania asynchronicznych funkcji.

Testy zdefiniowane w projekcie są wykrywane automatycznie. Jest to zależne od konfiguracji, którą możemy sami zdefiniować. Domyślnie wyszukiwane są pliki z rozszerzeniami `“.test.js”` lub `“.spec.js”`. Wsparcie dla TypeScript również musi zostać odpowiednio skonfigurowanie.

### **Definicja 3.2. Test jednostkowy**

*Test jednostkowy to procedura testowania, która sprawdza zachowanie pojedynczej jednostki kodu w izolacji od innych części systemu. Celem testu jednostkowego jest weryfikacja, czy dana jednostka kodu działa zgodnie z oczekiwaniami, spełnia założone warunki testowe i zwraca poprawne wyniki dla różnych scenariuszy testowych.*

### **Definicja 3.3. Test integracyjny**

*Test integracyjny, jest rodzajem testu oprogramowania, który polega na sprawdzaniu współpracy między różnymi komponentami lub modułami systemu. Celem testu integracyjnego jest weryfikacja, czy poszczególne części systemu działają poprawnie, gdy są ze sobą zintegrowane.*

### **Definicja 3.4. Test komponentów**

*Testowanie komponentów, znane również jako testowanie programu lub modułu, jest etapem testowania, który następuje po testowaniu jednostkowym. Polega ono na sprawdzaniu poszczególnych komponentów niezależnie od innych części systemu, takich jak moduły, klasy, obiekty czy programy.*

```
1  function addValues(a, b) {
2    return a + b;
3  }
4
5  describe("addValues", () => {
6    it("should return 4 when 3 and 1 provided", () => {
7      const result = addValues(3, 1);
8      expect(result).toEqual(4);
9    });
10
11    it("should not return 4 when 3 and 2 provided", () => {
12      const result = addValues(3, 2);
13      expect(result).not.toEqual(4);
14    });
15  });
```

Przykład 3.9: Przykład prostego testu w Jest

Przykład 3.9 przedstawia implementację metody `“addValues”` oraz testy jednostkowe przypisane do nich. Wykorzystanie `“describe”` pozwala na grupowanie testów względem wspólnej cechy. Wspólną cechą dla obu testów jest testowana metoda `“addValues”`. Testy definiujemy za pomocą metody `“it”`. Następnie symulujemy wykonanie metody i sprawdzenie dwóch przypadków. Pierwszy test sprawdza, czy poprawnie wprowadzone dane będą dawały poprawny wynik. Drugi test sprawdza, czy wprowadzenie błędnych danych nie będzie dawało poprawnego wyniku.

Do testowania komponentów Reactowych została wykorzystana biblioteka React Testing Library. React Testing Library jest to zestaw narzędzi umożliwiający testowanie komponentów Reactowych. Przede wszystkim biblioteka ta dostarcza narzędzia do manipulowania wirtualnym środowiskiem DOM. Testowanie z wykorzystaniem tej biblioteki stanowią dobre praktyki stosowane przez zespół React. Głównym elementem takiego podejścia do testowania jest fakt, że testy napisane z pomocą tej biblioteki są zbliżone do testów użytkowników.

```
1  const Counter = () => {
2    const [count, setCount] = useState(0);
3
4    const increment = () => {
5      setCount(count + 1);
6    };
7
8    return (
9      <div>
10        <h2>Counter</h2>
11        <p>Current count: {count}</p>
12        <button onClick={increment}>Increment</button>
13      </div>
14    );
15  };
16
17  describe('Counter', () => {
18    it('renders initial count', () => {
19      render(<Counter />);
20      const countElement = screen.getByText(/Current count:/i);
21      expect(countElement).toBeInTheDocument();
22      expect(countElement).toHaveTextContent('Current count: 0');
23    });
24
25    it('increments count on button click', () => {
26      render(<Counter />);
27      const incrementButton = screen.getByRole('button', { name: /
Increment/i });
28      fireEvent.click(incrementButton);
29      const countElement = screen.getByText(/Current count:/i);
30      expect(countElement).toHaveTextContent('Current count: 1');
31    });
32  });
```

Przykład 3.10: Przykład prostego testu komponentu Reactowego z wykorzystaniem Jest i React Testing Library

Przykład 3.10 przedstawia prosty komponent “Counter”, który renderuje licznik i przycisk do zwiększania wartości licznika. W teście używamy biblioteki “@testing-library/react” do renderowania komponentu i wykonania asercji. W pierwszym teście sprawdzamy, czy początkowa wartość licznika wynosi 0. W drugim teście symulujemy kliknięcie przycisku i sprawdzamy, czy wartość licznika została zwiększona o 1.

Testy używają funkcji render do renderowania komponentu, a następnie korzystają z funkcji screen.getByText i screen.getByRole z React Testing Library do znajdowania elementów interfejsu użytkownika na podstawie ich treści lub roli. Następnie, przy użyciu funkcji expect z biblioteki jest-dom, wykonujemy asercje na znalezionych elementach.



### Definicja 3.5. Test akceptacyjny

*Test akceptacyjny, jest procesem weryfikacji, który polega na sprawdzeniu, czy aplikacja lub system spełnia wymagania biznesowe, scenariusze użycia i oczekiwania użytkowników końcowych. Testy te są oparte na funkcjonalności i celu końcowego produktu oraz są przeprowadzane w celu zaakceptowania systemu przed jego wdrożeniem.*

## Puppeteer

Puppeteer to narzędzie do automatyzacji przeglądarki internetowej, które umożliwia programistom kontrolowanie, manipulację i testowanie stron internetowych w sposób programatyczny. Jest to oprogramowanie open source rozwijane przez zespół Chrome DevTools, które działa na silniku przeglądarki Chrome lub Chromium. Można programować interakcje z przeglądarką, takie jak otwieranie stron, klikanie, wypełnianie formularzy, nawigacja, przechwytywanie zrzutów ekranu, obsługa dialogów, manipulacja elementami strony i wiele innych. Przede wszystkim można tworzyć testy, które symulują interakcje użytkownika, sprawdzają stan strony, weryfikują poprawność renderowania elementów i asercje na zawartości strony. Puppeteer zapewnia narzędzia do wykonywania testów jednostkowych [3.2], testów integracyjnych [3.3] i testów end-to-end [3.6].

### Definicja 3.6. Test end-to-end

*Test end-to-end to proces sprawdzania, czy system lub aplikacja działa poprawnie, uwzględniając wszystkie komponenty, warstwy i zależności między nimi. Testy te symulują pełne ścieżki użytkownika, zaczynając od inicjalizacji systemu lub aplikacji, przechodząc przez różne etapy i interakcje, aż do końcowego rezultatu lub oczekiwanego stanu.*

Istnieje możliwość sterowania przeglądarką internetową, wykonując różne akcje, takie jak otwieranie stron, klikanie, wypełnianie formularzy, nawigacja, przechwytywanie zrzutów ekranu, obsługa dialogów, manipulacja elementami strony i wiele innych. Możemy tworzyć testy, które naśladują interakcje użytkownika, sprawdzają stan strony, weryfikują poprawność renderowania elementów oraz sprawdzają zawartość strony. Istnieje możliwość analizowania i pobierania treści, metadanych, obrazów, linków oraz innych informacji ze stron internetowych. Jest to szczególnie przydatne w procesach takich jak zbieranie danych, analiza, monitorowanie konkurencji i wiele innych zastosowań. Z wykorzystaniem Puppeteer możemy także przechwytywać widoki stron, zapisywać je jako obrazy lub tworzyć pliki PDF zawierające pełną zawartość stron.

```
1 const puppeteer = require('puppeteer');
2
3 (async () => {
4   const browser = await puppeteer.launch();
5   const page = await browser.newPage();
6   await page.goto('https://example.com');
7   const title = await page.title();
8   console.log('Tytuł strony:', title);
9   await page.screenshot({ path: 'screenshot.png' });
10  await browser.close();
11 })();
```

Przykład 3.11: Przykład prostego testu w Puppeteer

W przykładzie 3.11 najpierw importujemy moduł Puppeteer. Następnie w bloku “async” tworzymy funkcję anonimową i używamy metody “puppeteer.launch()” do uruchomienia przeglądarki. Następnie tworzymy nową stronę przy użyciu “browser.newPage()”. Po inicjalizacji przeglądarki, korzystamy z “page.goto()” do nawigacji do określonej strony internetowej. W tym przypadku używamy strony przykładowej “https://example.com”. Następnie sprawdzamy tytuł strony, wykorzystując “page.title()”. Tytuł jest zwracany jako obiekt “Promise”, więc używamy “await”, aby oczekiwać na wynik. Wykonujemy zrzut ekranu strony przy użyciu “page.screenshot()” i podajemy ścieżkę, pod którą chcemy zapisać plik z zrzutem. Na końcu zamykamy przeglądarkę, wywołując “browser.close()”.

## Cucumber

Cucumber to narzędzie do testowania oprogramowania, które umożliwia pisanie testów akceptacyjnych w języku naturalnym, zrozumiałym dla wszystkich zaangażowanych w projekt. Jest popularnym narzędziem w praktyce BDD (Behavior-Driven Development) i promuje współpracę pomiędzy członkami zespołu projektowego, takimi jak programiści, testerzy, analitycy biznesowi i interesariusze.

Jest to narzędzie, które używa specjalnego języka o nazwie Gherkin do pisania czytelnych i zrozumiałych testów dla wszystkich osób zaangażowanych w projekt. Gherkin koncentruje się na opisie zachowania aplikacji za pomocą prostych słów kluczowych, takich jak “Zakładając” (Given), “Jeżeli” (When) i “Wtedy” (Then). Na podstawie scenariuszy napisanych w Gherkinie, Cucumber automatycznie generuje szablony kodu testów w wybranym języku programowania, takim jak Java, JavaScript, Ruby itp. Dzięki temu możliwa jest automatyzacja tworzenia testów na podstawie specyfikacji biznesowych.

Cucumber skupia się na testowaniu aplikacji na podstawie oczekiwanego zachowania. Scenariusze testowe opisują, jak aplikacja powinna się zachowywać w różnych sytuacjach i jakie wyniki oczekujemy. To podejście pomaga skupić się na wymaganiach biznesowych i spełnianiu oczekiwań użytkowników. Mamy możliwość połączenia specyfikacji biznesowych z kodem testów, co ułatwia śledzenie i zarządzanie wymaganiami. Można tworzyć implementacje kroków opisanych w scenariuszach, które są powiązane z kodem testów. Dzięki temu testy są związane z konkretną funkcjonalnością i ułatwiają śledzenie pokrycia testowego.

Z wykorzystaniem takich narzędzi jak Cucumber promujemy współpracę i porozumienie między różnymi członkami zespołu projektowego. Scenariusze testowe napisane w języku naturalnym są zrozumiałe dla wszystkich, a proces tworzenia specyfikacji testowych staje się wspólnym wysiłkiem. Dzięki temu narzędziu możliwa jest lepsza komunikacja i zrozumienie między programistami, testerami, analitykami biznesowymi i innymi członkami zespołu, co przyczynia się do skuteczniejszego testowania i dostarczania oprogramowania zgodnego z oczekiwaniami.

```
1 Feature: User login
2   To gain access to secured assets
3   As a user
4   I would like to login in to system
5
6   Scenario: Logged in with correct credentials
7     Given User is on login page
```

```

8   When Correct input provided in login form
9   And Click "Login"
10  Then User see welcome screen
11
12  Scenario: Login with incorrect credentials
13  Given User is on login page
14  When Incorrect input provided in login form
15  And Click "Login"
16  Then User see error message

```

Przykład 3.12: Przykład prostego testu napisanego w Gherkin

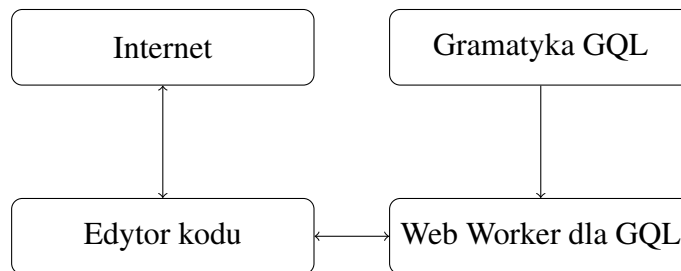
W przykładzie 3.12 mamy prostą specyfikację testową w języku Gherkin z dwoma scenariuszami. Pierwszy scenariusz dotyczy logowania poprawnymi danymi, a drugi scenariusz dotyczy logowania niepoprawnymi danymi.

W przykładzie pierwszego scenariusza test polega na sprawdzeniu logowania poprawnymi danymi. Pierwszy krok mówi nam, że jesteśmy na stronie logowania. Następnie wprowadzamy poprawne dane logowania, klikamy przycisk “Zaloguj się” i sprawdzamy, czy widzimy komunikat powitalny.

W drugim scenariuszu testujemy logowanie niepoprawnymi danymi. Kolejne kroki są podobne, ale wprowadzamy niepoprawne dane logowania i sprawdzamy, czy widzimy komunikat o błędzie logowania.

## 3.2 Omówienie zaimplementowanej aplikacji

Głównym celem pracy było dostarczenie aplikacji umożliwiającej weryfikację składni formalnej języka GQL [1.4.4]. Aplikacja została oparta o architekturę podaną na poniższym rysunku. Architektura ta składa się z edytora, gramatyki i WebWorkera. Stanowi on trzon całego rozwiązania i sposobu, w jaki aplikacja została zaimplementowana.



Rysunek 3.4: Schemat architektury implementacji

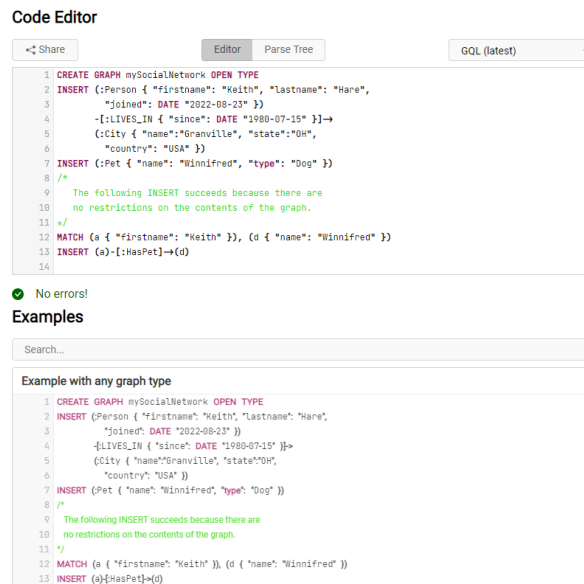
Schemat zawarty na rysunku 3.4 przedstawia przebieg komunikacji między elementami. Gramatyka GQL jest wykorzystywana w elemencie WebWorker tylko i wyłącznie w jednym kierunku. Wynika to z faktu, że gramatyka jest implementowana jako biblioteka wykorzystywana w aplikacji TypeScript. Edytor natomiast komunikuje się z WebWorkerem w dwie strony. Wysyłamy zapytanie o konkretnej formie, a następnie oczekujemy aż algorytm przetwarzający, zwróci odpowiednią odpowiedź. Ostatnimi elementami powiązanymi relacją jest sam edytor oraz internet. Relacja ta wynika z faktu, iż edytor będzie aplikacją internetową, a co za tym idzie, będzie dostępny na wielu platformach.

Wyodrębnienie implementacji analizy składniowej i leksykalnej ma swoje gruntowne powody. Dzięki takiemu rozwiązaniu jesteśmy w stanie uzyskać swego rodzaju wielowątkową implementację. Język JavaScript na podstawach, którego powstał język TypeScript, jest językiem, który nie umożliwia implementacji rozwiązań wielowątkowych. Stanowi to spory problem, ponieważ proces analizy składniowej złożonych gramatyk jak GQL, może być dosyć czasochłonny. Zmusza nas to do szukania sposobu, aby tę wielowątkowość uzyskać. Z pomocą przychodzą oddzielone fragmenty kodu, które mogą być uruchomione asynchronicznie zwane WebWorkerami.

Tym asynchronicznym fragmentem kodu jest WebWorker. Mechanizm ten działa na zasadzie uruchomienia skryptu w tle, niezależnie od głównego wątku przeglądarki. Kod ten staje się wtedy swego rodzaju instancją wątku, z którym nasza aplikacja może się komunikować. Komunikacja odbywa się poprzez dwustronną wymianę wiadomości i nasłuchiwanie odpowiedzi na te wiadomości. Wiadomości mogą przysyłać różnego rodzaju dane oraz identyfikator rodzaju wysłanej wiadomości. Szczegółowe omówienie rodzajów wspieranych wiadomości zostanie omówione w dalszej części pracy.

## Implementacja edytora

Interaktywnym sercem zaimplementowanej aplikacji jest edytor. Edytor został zaimplementowany na podstawie biblioteki React i języka TypeScript. Rysunek 3.5 przedstawia wygląd interfejsu edytora. Interfejs ten składa się na cztery najważniejsze elementy, to jest nagłówek, edytor tekstu, status parsowania i konsola błędów oraz przykłady dla wybranego języka.



Rysunek 3.5: Główny widok edytora online

Pierwszym i najważniejszym elementem jest edytor, który umożliwia podawanie treści zapytań. Zapytania są fragmentem tekstu będącego pewną reprezentacją danych wejściowych

analizy składniowej języka formalnego. Obsługuje on takie operacje jak zapamiętywanie kroków wprowadzania, cofanie i przywracanie zmian, kolorowanie składni oraz podpowiadanie słów kluczowych. Posiada on także funkcjonalność podświetlania linijki, w której znajduje się błąd.

Edytor ten oparty jest na podstawie elementu `< textarea >` dostępnego w HTML5. Oczywiście, możliwości tego elementu są bardzo ograniczone. Nie możemy w nim zaimplementować na przykład kolorowania składni czy podpowiadania słów kluczowych. Do zaimplementowania większości elementów interakcji z edytorem zostały wykorzystane zdarzenia DOM tego elementu.

Kolorowanie składni zostało zaimplementowane w elemencie `< pre >`, dzięki któremu mamy możliwość nadpisania wyglądu edytora. Tag ten umożliwia wyświetlanie sformatowanego tekstu. Tekst w tym elemencie jest wyświetlany czcionką o stałej szerokości, a tekst zachowuje zarówno spacje, jak i podziały wierszy. W implementacji edytora tekst jest dzielony na poszczególne elementy na podstawie szeregu wyrażeń regularnych. Wyrażenia te są budowane po stronie web workera i oparte na danych uzyskanych z parsera i leksera. Struktura wygenerowanego elementu `< pre >` oparta jest o podejście tabelaryczne. Jedna kolumna odpowiada za zawartość linii na pasku numerycznym, druga natomiast za zawartość określonej linijki. Elementy wyodrębnione za pomocą wyrażeń regularnych są umieszczane w elementach `< span >`, którym to podaje się odpowiednie klasy. Zastosowanie różnych klas daje nam możliwość dowolnego kolorowania składni. Aplikacja ma założone kilka podstawowych klas, które są wykorzystywane do opisanie kolorami określonych składni. Tabela 3.2 przedstawia podstawowe elementy składni, które można wykorzystać do kolorowania elementów języka.

Nazwa elementu	Identyfikator	Przykład
Zwykły tekst	-	przykład
Słowo kluczowe	Keyword (0)	<b>przykład</b>
Znaki specjalne	Punctuation (1)	<b>przykład</b>
Komentarz	Comment (2)	przykład
Ciąg znaków	String (3)	przykład
Liczba	Number (4)	przykład
Symbol logiczny	Boolean (5)	przykład
Element specjalny 1	Custom1 (5)	przykład
Element specjalny 2	Custom2 (6)	przykład
Element specjalny 3	Custom3 (7)	przykład
Element specjalny 4	Custom4 (8)	przykład

Tabela 3.2: Spis dostępnych elementów gramatyki w edytorze

Kolejnym elementem wprowadzonym do edytora jest podpowiadanie słów kluczowych. Jest to prosty mechanizm, który operuje na zdarzeniach edytora. Główne zdarzenia, na których oparte jest wyświetlanie okna podpowiedzi to zmiana pozycji kursora oraz prośba o pokazanie menu kontekstowego. Operowanie pomiędzy możliwymi opcjami odbywa się na zasadzie podłączenia globalnych skrótów klawiszowych. Zasada działania wyświetlania okna działa na podstawie pokazywania kontenera za pomocą opcji stylowania zawartych w pliku

CSS. Rysunek 3.6 przedstawia wygląd menu podpowiadania.

```

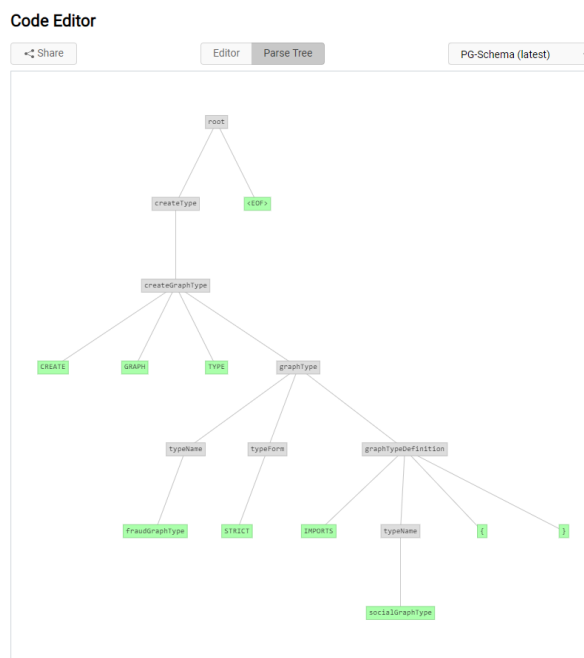
1 CREATE GRAPH mySocialNetwork OPEN TYPE
2 INSERT (:Person { "firstname": "Keith", "lastname": "Hare",
3             "joined": DATE "2022-08-23" })
4       -[:LIVES_IN { "since": DATE "1980-07-15" }→
5       (:City { "name": "Granville", "state": "OH",
6               "country": "USA" })
7 INSERT (:Pet { "name": "Winnifred", "type": "Dog" })
8 /*
9  The following INSERT succeeds because there are
10 no restrictions on the contents of the graph.
11 */
12 MATCH (a { "firstname": "Keith" }, (d { "name": "Winnifred" }))
13 INSERT (a)-[:HasPet]→(d)
14 INSE

```

### Rysunek 3.6: Podpowiadanie składni w edytorze

Podkreślanie linijek zostanie przedstawione w dalszej części, wraz z omówieniem konsoli przedstawiającej status parsowania oraz listy błędów.

Ważnym elementem, a przede wszystkim bardzo użytecznym jest ilustracja drzewa parsowania. Rysunek 3.7 przedstawia przykładowe drzewo parsowania dla wprowadzonych danych. Przedstawia on przykład napisany w PG-Schema.

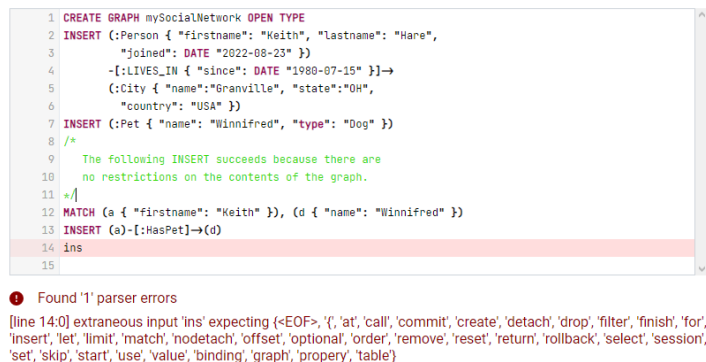


Rysunek 3.7: Widok drzewa parsowania

Drzewo parsowania [2.18] w przypadku implementacji aplikacji jest wynikiem działania parsowania. Dane otrzymane w wyniku parsowania musiały zostać odpowiednio zmodyfikowane w celu osiągnięcia prostszej struktury. Uproszczona struktura znacząco ułatwia wyświetlanie. Upraszczanie modelu odbywa się na poziomie web workera, więc zostanie

to omówione w dalszej części. Najważniejsze elementy to przedstawienie symboli terminalnych i nieterminalnych. Symbole nieterminalne są oznaczone kolorem szarym, natomiast kolorem zielonym są oznaczone symbole terminalne.

Status procesu parsowania oraz ewentualne błędy są pokazywane w konsoli błędów. Konsola ta jest umieszczona bezpośrednio pod edytorem. Zawiera ona dwa elementy, pierwszym elementem jest status procesu parsowania. Jako iż jest to proces asynchroniczny, istnieją trzy statusy: parsowanie, brak błędów składni oraz wykryto określoną liczbę błędów. Statusy posiadają także reprezentujące je kolory, odpowiednio szary, zielony oraz czerwony. W przypadku statusu parsowania pokazują się animowana kontrolka reprezentująca trwający proces. Natomiast w pozostałych statusach odpowiednio jest to znak odhaczenia oraz znak wykrzyknika. Pod spodem statusu pojawia się lista błędów. Każdy element list zaczyna się od linii oraz kolumny, w której wystąpił dany błąd. Elementy kończą się opisem błędu oraz podpowiedzią, w jaki sposób można naprawić problem. Dodatkowo aplikacja została wyposażona w mechanizm podkreślania afektowanych linii. Wszystko to zostało oparte na danych z web workera, które generowane są podczas parsowania. Model błędów w uproszczonej formie jest dostarczany do edytora, po czym odpowiednio mapowany na tekst, a także na pozycję linii. Rysunek 3.8 przedstawia formę, w jaki sposób prezentowane są błędy składni język formalnego.



```
1 CREATE GRAPH mySocialNetwork OPEN TYPE
2 INSERT (:Person { "firstname": "Keith", "lastname": "Hare",
3   "joined": DATE "2022-08-23" })
4   -[:LIVES_IN { "since": DATE "1980-07-15" }]→
5   (:City { "name": "Granville", "state": "OH",
6     "country": "USA" })
7 INSERT (:Pet { "name": "Winnifred", "type": "Dog" })
8 /*
9   The following INSERT succeeds because there are
10  no restrictions on the contents of the graph.
11 */
12 MATCH (a { "firstname": "Keith" }), (d { "name": "Winnifred" })
13 INSERT (a)-[:HasPet]→(d)
14 ins
15
```

Found '1' parser errors

[line 14:0] extraneous input 'ins' expecting {<EOF>, '(', 'at', 'call', 'commit', 'create', 'detach', 'drop', 'filter', 'finish', 'for', 'insert', 'let', 'limit', 'match', 'nodetach', 'offset', 'optional', 'order', 'remove', 'reset', 'return', 'rollback', 'select', 'session', 'set', 'skip', 'start', 'use', 'value', 'binding', 'graph', 'property', 'table'}

Rysunek 3.8: Przedstawienie błędów składni

Przedostatnim elementem są przykłady języka formalnego. Jest to lista dostępnych przykładów zależna od wybranego języka formalnego. Każdy z przykładów składa się z kodu przykładu oraz tytułu. Dostarczona została także możliwość wyszukania interesujących nas elementów. Wyszukiwanie działa na zasadzie wyszukiwania podobnych słów w tytule, jaki w zawartości języka. Najlepiej dopasowane przykłady pokazują się wyżej, a mniej pasujące niżej. Po kliknięciu przykładu aplikacja automatycznie ustawia ten przykład jak aktualną treść, którą zawiera edytor. Rysunek 3.9 przedstawia interfejs listy przykładów.



## Examples

Search...

Example with any graph type

```

1 CREATE GRAPH mySocialNetwork OPEN TYPE
2 INSERT (Person { 'firstname': 'Keith', 'lastname': 'Hare',
3               'joined': DATE '2022-08-23' })
4   -[LIVES_IN { 'since': DATE '1980-07-15' }->
5     (City { 'name': 'Granville', 'state': 'OH',
6            'country': 'USA' })
7 INSERT (Pet { 'name': 'Winnifred', 'type': 'Dog' })
8 /*
9 The following INSERT succeeds because there are
10 no restrictions on the contents of the graph.
11 */
12 MATCH (a { 'firstname': 'Keith' }), (d { 'name': 'Winnifred' })
13 INSERT (a)-[HasPet]->(d)

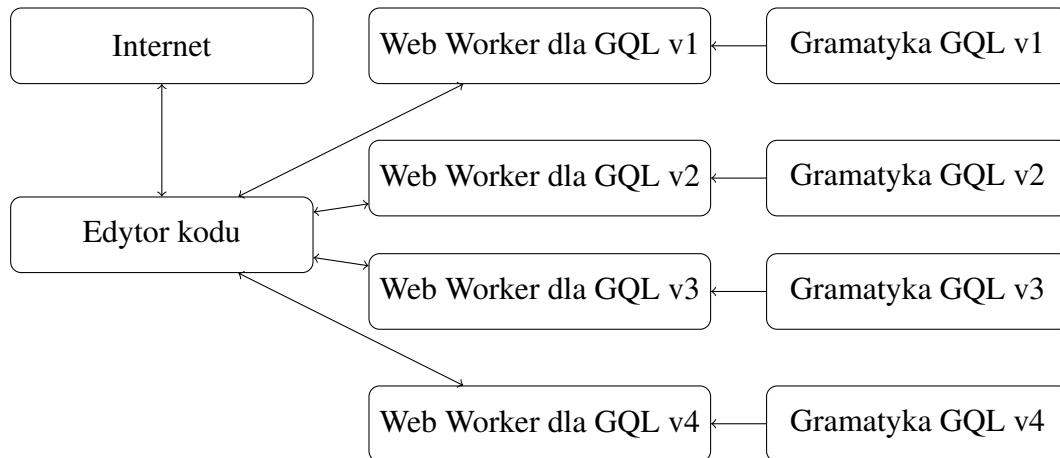
```

Example with closed graph type

Rysunek 3.9: Przykłady języka i wyszukiwanie przykładów

Ostatnim elementem implementacji jest możliwość zmiany języków formalnych. Do-myślnie funkcjonalność ta miała za zadanie umożliwić zmianę wersji języka formalnego. Fi-nalnie aplikacja może obsługiwać zupełnie różne języki formalne, przy zachowaniu wszyst-kich funkcjonalności. Kluczowym elementem dzięki, któremu było to możliwe, było wydzie-lenie procesu parsowania i ekstrakcji meta danych języka do WebWorkera. O możliwościach i implementacji WebWorkera będziemy pisać w dalszej części pracy.

Na rysunku 3.10 został przedstawiony rozszerzony schemat architektury przedstawionej na początku omówienia implementacji. Schemat ten pokazuje kilka możliwych rozgałęzień będących różnymi wersjami WebWorkera.



Rysunek 3.10: Rozszerzony schemat architektury aplikacji

W przypadku edytora, patrząc na schemat, widzimy cztery różne wersje języka, które możemy wybrać z poziomu interfejsu. Po zmianie web workera następuje podmiana całego jądra parsowania, a co za tym idzie cała aplikacja, dostosowuje swoje funkcjonalności do wybranego języka formalnego.



## Implementacja web workera

Web worker jest elementem wprowadzającym asynchroniczność w kodzie aplikacji. W tym elemencie został zawarty proces analizy leksykalnej oraz parsowania. Dodatkowo umożliwia on ekstrakcję meta danych języka formalnego w celu dostarczenia takich funkcjonalności jak podpowiadanie słów kluczowych czy kolorowanie składni języka.

Rozwiązanie to obsługuje trzy kluczowe zdarzenia. Pierwszym obsługiwanym typem przyjmowanych wiadomości jest inicjalizacja całego procesu. Drugim obsługiwanym typem przyjmowanych wiadomości jest zdążenie dostarczające narzędzia języka formalnego. Trzecim obsługiwanym typem wiadomości jest wykonanie analizy leksykalnej i składniowej na podanym zakresie danych.

Pierwsza forma wiadomości dostarcza możliwość pobrania podstawowych informacji o języku. W tym kroku ekstraktowane są słowa kluczowe z wygenerowanego parsera. Kolejnym elementem dostarczonym przez ten krok jest wygenerowanie i częściowe zaimplementowanie meta danych języka formalnego. Dane te po uprzednim przygotowaniu zwracane są do edytora.

Druga forma wiadomości jest tylko i wyłącznie formalną opcją, która może umożliwić rozszerzenie aplikacji. Generuje ona podstawowy parser i analizator leksykalny oraz zwraca go do edytora. Edytor może wtedy wykorzystać te narzędzia do przeprowadzenia określonych akcji.

Trzecia forma wiadomości dostarcza możliwość wykonania analizy składni dla podanego zakresu danych. Podstawowy element, czyli parsowanie jest wykonywane w pierwszy momencie działania tego kroku. Następnie z wyniku parsowania wyodrębniane jest drzewo parsowania. Ważnym elementem jest także ekstrakcja błędów z parsera. Proces ekstrakcji jest oparty na implementacji tak zwanego słuchacza (listenera) zdarzeń parsowania. W momencie, kiedy zdarzenie zostanie zauważone, dodawany jest nowy rekord błędu w klasie listenera. Błędy po wykonaniu parsowania zostają przesłane do edytora wraz z drzewem parsowania.

Ekstrakcja drzewa parsowania polega na uproszczeniu modelu zwracanego przez wynik parsowania. Po ówczesnym odpowiednim skonfigurowaniu parsera zwraca on wszystkie potrzebne dane do umożliwienia rysowania. Uproszczony model będzie składał się z dwóch rodzajów elementów węzłów. Węzeł nieterminalny zawiera nazwę oraz tablicę potomków. Węzeł terminalny zawiera typ numeru oraz nazwę danego symbolu.

## Implementacja gramatyki i narzędzi do analizy języka GQL

Implementacja gramatyki składa się z dwóch elementów, z implementacji gramatyki w ANTLR oraz z wygenerowanego kodu przez ANTLR w TypeScript. Opis gramatyki formalnej składa się z dwóch plików. Pierwszy plik opisuje gramatykę parsera, która będzie wykorzystywana do definiowania reguł symboli nieterminalnych. Drugi plik opisuje gramatykę analizatora leksykalnego, która będzie wykorzystana do opisu symboli terminalnych oraz reguł opisujących symbol terminalny.

Przykład 3.13 przedstawia początek implementacji gramatyki parsera. Jest to przedstawieniem fragmentu składni samego języka GQL w formie formalnej definicji. Charakterystyczną cechą gramatyki parsera jest nazewnictwo reguł. Każda nazwa reguły zaczyna się z małej litery.

```

1 parser grammar GqlParser;
2
3 options {
4     language = JavaScript;
5     tokenVocab = GqlLexer;
6 }
7
8 // program grammar
9 gqlProgram: activity* EOF;
10
11 activity: programActivity sessionCloseCommand?;
12
13 programActivity: sessionActivity | transactionActivity;
14
15 // session management grammar
16 sessionActivity: sessionSetCommand | sessionResetCommand;
17
18 sessionSetCommand:
19     SESSION SET (
20         sessionSetSchemaClause
21         | sessionSetGraphClause
22         | sessionSetTimeZoneClause
23         | sessionSetParameterClause
24     );
25
26 sessionSetSchemaClause: SCHEMA schemaRef;
27
28 sessionSetGraphClause: PROPERTY? GRAPH graphExpr;
29
30 sessionSetTimeZoneClause: TIME ZONE setTimeZoneValue;
31
32 setTimeZoneValue: stringValueExpr;
33
34 sessionSetParameterClause:
35     sessionSetGraphParameterClause
36     | sessionSetBindingTableParameterCalues
37     | sessionSetValuesParameterClause;
38
39 sessionSetGraphParameterClause:
40     PROPERTY? GRAPH sessionSetParameterName optTypedGraphInit;
41
42 sessionSetBindingTableParameterCalues:
43     BINDING? TABLE sessionSetParameterName optTypedBindingTableInit;
44
45 sessionSetValuesParameterClause:
46     VALUE sessionSetParameterName optTypedValueInit;
47
48 sessionSetParameterName: IF_NOT_EXISTS? parameterName;
49
50 sessionResetCommand: SESSION? RESET sessionResetArguments;
51
52 sessionResetArguments:
53     ALL? (PARAMETERS | CHARACTERISTICS)
54     | SCHEMA

```

```

55 | PROPERTY? GRAPH
56 | TIME ZONE
57 | PARAMETER parameterName;
58
59 sessionCloseCommand: SESSION? CLOSE;
60
61 ...

```

### Przykład 3.13: Fragment implementacji parsera GQL

Przykład 3.14 przedstawia początek implementacji leksera. Ważnym elementem jest to, że w pliku definiującym lexer definiujemy symbole terminalne. Reguły opisujące symbole terminalne zazwyczaj pisze się dużymi literami, a słowa oddziela się znakiem .

```

1 lexer grammar GqlLexer;
2
3 options {
4   caseInsensitive = true;
5 }
6
7 // arrows
8 LEFT_ARROW: '<-';
9 LEFT_ARROW_TILDE: '<~';
10 LEFT_ARROW_BRACKET: '<-[';
11 LEFT_ARROW_TILDE_BRACKET: '<~[';
12 LEFT_MINUS_RIGHT: '<->';
13 LEFT_MINUS_SLASH: '<-/';
14 LEFT_TILDE_SLASH: '<~/';
15 MINUS_LEFT_BRACKET: '-[';
16 MINUS_SLASH: '-/';
17 RIGHT_ARROW: '->';
18 RIGHT_BRACKET_MINUS: ']-';
19 RIGHT_BRACKET_TILDE: ']~';
20 BRACKET_RIGHT_ARROW: ']->';
21 BRACKET_TILDE_RIGHT_ARROW: ']~>';
22 SLASH_MINUS: '/-';
23 SLASH_MINUS_RIGHT: '/->';
24 SLASH_TILDE: '/~';
25 SLASH_TILDE_RIGHT: '/~>';
26 TILDE_LEFT_BRACKET: '~[';
27 TILDE_RIGHT_ARROW: '~>';
28 TILDE_SLASH: '~/';
29
30 ...

```

### Przykład 3.14: Fragment implementacji leksera GQL

Definicja gramatyki w notacji ANTLR jest pierwszym krokiem. Kolejnym krokiem jest wygenerowanie kodu źródłowego dla odpowiedniego środowiska uruchomieniowego. W przypadku aktualnej implementacji środowiskiem tym było środowisko przeglądarki oraz środowisko języka TypeScript.

Narzędzia dostarczane przez ANTLR generują trzy podstawowe pliki. Odpowiednio pliki zawierają implementację parsera, implementację listenera parsera oraz implementację analizatora leksykalnego.

Kod wygenerowany dla parsera zdecydowanie jest najbardziej skomplikowany. Wynika to z faktu, iż jego implementacja rozpatruje tokeny i rozбивa na odpowiedni konteksty. Dla każdej reguły zaimplementowanej w notacji parsera generowany jest odpowiedni kontekst. Kontekst ten zawiera różne przejścia stanów w zależności od implementacji.

Kod wygenerowany dla listenera jest klasą abstrakcyjną zawierającą metody, które jesteśmy w stanie nadpisać. Metody te odpowiadają każdemu możliwemu stanowi, w jakim może znaleźć się parser. Możemy wtedy zaimplementować dodatkową logikę w momencie, kiedy analiza składniowa wejdzie w określone reguły. Dla przykładu jesteśmy w stanie zapisywać nazwy zmiennych wpisanych w danych wejściowych oraz określenie ich wartości. Jest to bardzo pomocne w wykorzystaniu przez interpretatory zdefiniowanego języka formalnego.

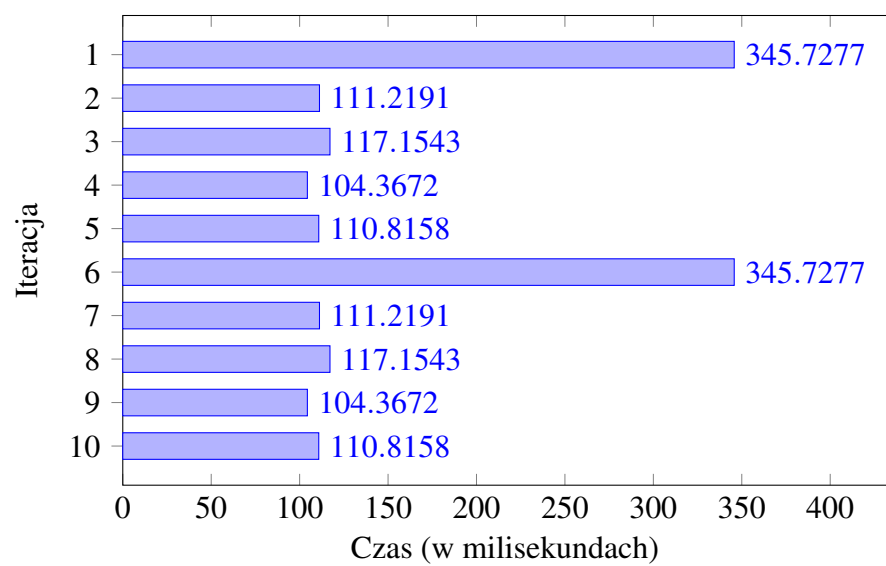
Ostatnim wygenerowanym kodem jest kod analizatora leksykalnego. Jest to klasa umożliwiająca wykonanie tokenizacji na wprowadzanych danych. Jesteśmy w stanie skonfigurować wiele rzeczy takich jak na przykład wykrywanie błędów analizy leksykalnej.

Implementacja gramatyki w środowisku wymagała także dodatkowych implementacji związanych z biblioteką ANTLR dla TypeScript. W najnowszej wersji ANTLR4 została dostarczona wbudowana obsługa języka TypeScript. Niestety spora część implementacji posiadała błędne typowanie. Zostało to naprawione i dostosowane do implementacji gramatyki wykorzystanej w edytorze.

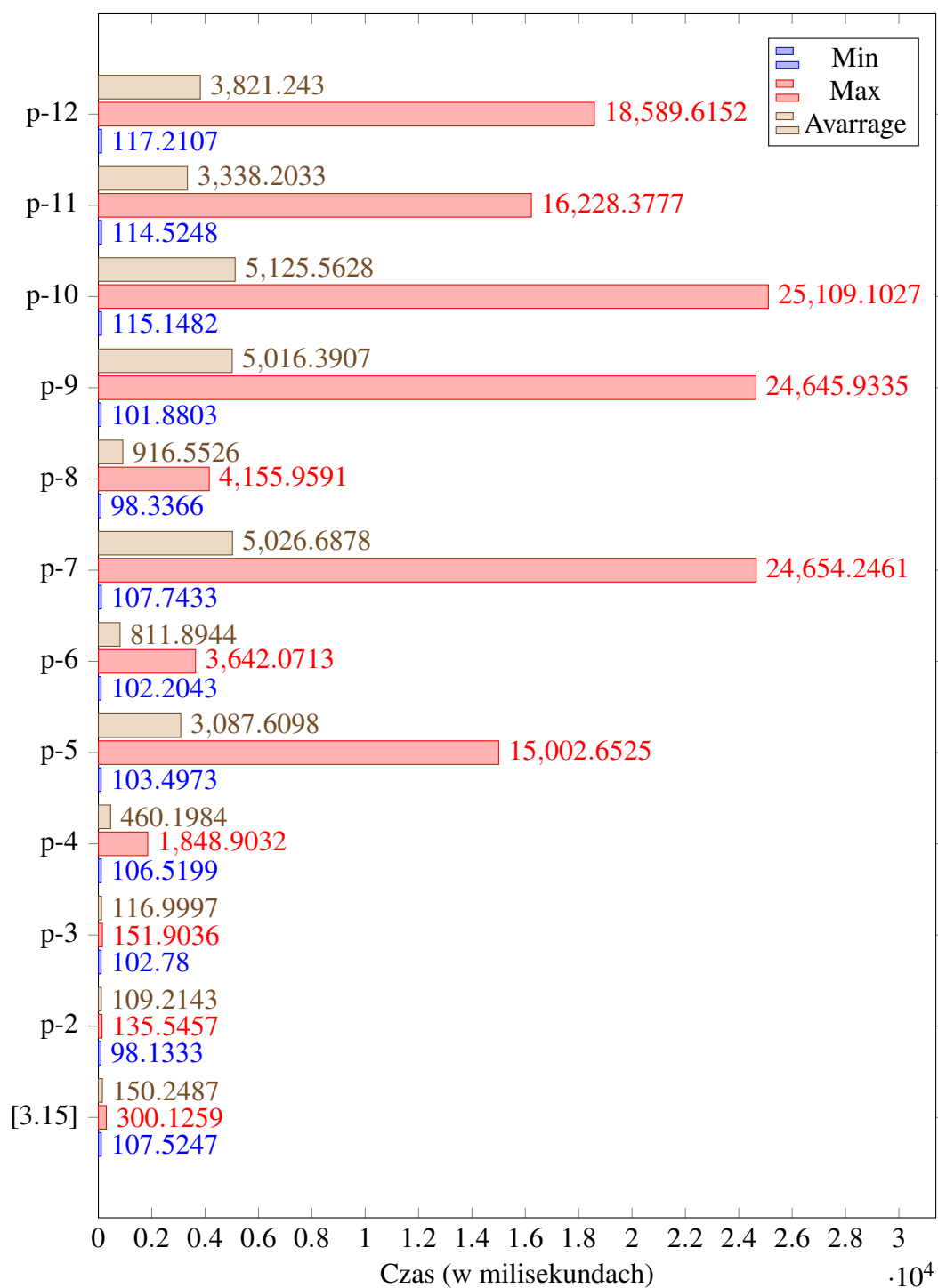
### 3.3 Weryfikacja wydajności i testy

Przykład 3.15: Przykład GQL 1 (p-1)

```
CREATE GRAPH TYPE IF NOT EXISTS example_graph_type
AS {
  (Person :Person { "lastname" STRING, "firstname" STRING, "joined" DATE },
  (City :City { "name" STRING, "state" STRING, "country" STRING}),
  (Pet :Pet { "name" STRING, "type" STRING}),
  (Person)-[LivesIn :LIVES_IN { "since" DATE }]->(City),
  (Person)-[Knows :KNOWS]->(Person)
}
```



Rysunek 3.11: Wynik testów przykładu 1 [3.15]



Rysunek 3.12: Ogólne wyniki wydajności gramatyki

## **Podsumowanie**

# Spis rysunków

1	Wykres wzrostu zainteresowania frazą "nosql" . . . . .	4
2	Wykres przedstawiający popularność poszczególnych typów systemów baz danych . . . . .	5
3	Wykres przedstawiający najpopularniejsze grafowe systemy baz danych . . .	6
1.1	Przykład grafu nieskierowanego . . . . .	14
1.2	Przykład grafu skierowanego . . . . .	15
1.3	Przykład relacji przechodniej . . . . .	17
1.4	Przykład relacji nieprzechodniej . . . . .	17
1.5	Przykład relacji jednostronnej . . . . .	17
1.6	Przykład relacji dwustronnej . . . . .	18
1.7	Przykład hiper grafu na relacjach rodzinnych . . . . .	20
1.8	Przykładowy graf właściwości przedstawiający relacje społeczne . . . . .	22
1.9	Przykład grafu RDF . . . . .	24
2.1	Przebieg tłumaczenia języków formalnych . . . . .	55
3.1	Wykres przedstawiający wzrost wykorzystania języków programowania . . .	68
3.2	Cykl życia komponentu w React . . . . .	78
3.3	Efekt implementacji aplikacji w React . . . . .	85
3.4	Schemat architektury implementacji . . . . .	90
3.5	Główny widok edytora online . . . . .	91
3.6	Podpowiadanie składni w edytorze . . . . .	93
3.7	Widok drzewa parsowania . . . . .	93
3.8	Przedstawienie błędów składni . . . . .	94
3.9	Przykłady języka i wyszukiwanie przykładów . . . . .	95
3.10	Rozszerzony schemat architektury aplikacji . . . . .	95
3.11	Wynik testów przykładu 1 [3.15] . . . . .	100
3.12	Ogólne wyniki wydajności gramatyki . . . . .	101



# Spis tabel

1.1	Przykładowa tabela atrybutów w grafowej bazie danych. . . . .	18
1.2	Symbole dostępne we wzorcach ze zmienną długością (PGQL) . . . . .	30
1.3	Obsługiwane języki zapytań przez bazy danych . . . . .	43
1.4	Obsługiwane języki programowania przez bazy danych . . . . .	44
2.1	Syntaktyczne elementy języka BNF . . . . .	62
2.2	Syntaktyczne elementy języka EBNF . . . . .	63
2.3	Syntaktyczne elementy języka ABNF . . . . .	64
2.4	Syntaktyczne elementy języka ANTLR . . . . .	66
2.5	Porównanie notacji języków do definiowania gramatyki . . . . .	67
3.1	Podstawowe typy danych dostępne w TypeScript . . . . .	69
3.2	Spis dostępnych elementów gramatyki w edytorze . . . . .	92