Informatik
Hauptcampus

H O C H
S C H U L E
T R I E R

gedcom7.js

Realisierung einer JavaScript-Bibliothek für das genealogische Austauschformat FamilySearch GEDCOM Version 7

Marius Müller & David Gruber

Bachelor-Projektarbeit

Betreuer: Christian Bettinger

Trier, 28.02.2023

Kurzfassung

In dieser Arbeit wurde die JavaScript-Bibliothek *gedcom7.js* für das genealogische Austauschformat FamilySearch Gedcom Version 7 entwickelt und getestet. Das grundlegende Element der Bibliothek stellt dabei ein Parser dar, der Dateien im Format Gedcom7 lesen und schreiben kann. Beim Lesen wird der Inhalt der Datei, nach der in der Gedcom7 Spezifikation definierte Syntax überprüft und in geeignete Datenstrukturen überführt. Diese Datenstrukturen stellen Funktionalitäten zu Veränderung und Erweiterung der Dateien bereit. Bei allen Operationen finden Syntaxüberprüfungen statt, sodass zu jedem Zeitpunkt garantiert ist, dass eine Gedcom7-konforme Struktur vorliegt. Der Hauptfokus der Implementierung liegt auf einfacher Erweiterbarkeit, weshalb ein Grammatik Generator entwickelt wurde, der die Grammatiken zur Syntaxüberprüfung automatisiert generiert.

Abstract

In this work, the JavaScript library gedcom for the genealogical exchange format FamilySearch Gedcom Version 7 was developed and tested. The basic element of the library is a parser that can read and write files in the Gedcom7 format. During reading, the content of the file is checked according to the syntax defined in the Gedcom7 specification and converted into suitable data structures. These data structures provide functionalities for changing and expanding the files. Syntax checks are performed in all operations, so that a Gedcom7-compliant structure is guaranteed at all times. The main focus of the implementation is on easy extensibility, which is why a grammar generator has been developed that automatically generates grammars for syntax checking.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1

Einleitung und Problemstellung

Jeder Zweite würde gerne mehr über seine Vorfahren wissen. [fDA07]

Diese Erkenntnis geht aus einer Umfrage des Instituts für Demoskopie Allensbach hervor. Das Sammeln von Informationen über Vorfahren und das Erkunden der eigenen Wurzeln stellt für viele Menschen eine Faszination dar. Anhand von verschiedenen Quellen, können neben dem Wohnort und der Lebensspanne von Ahnen auch Berufe, Lebensweisen und Familienverhältnisse gefunden werden. Außerdem bietet die Familienforschung die Möglichkeit bisher unbekannt Verwandte zu finden. [Mal]

Besonders das Aufkommen des Internets brachte einen Aufschwung für die Familienforschung hervor. Die einfachere Kommunikation auch über Landesgrenzen hinweg, ermöglicht es genealogische Informationen auszutauschen und so Gleichgesinnte und Verwandte auf der ganzen Welt zu finden [Ahn]. Um diese Kommunikation zu ermöglichen, werden Datenformate und Standards benötigt. Eines dieser Datenaustauschformate ist FamilySearch Gedcom Version 7, das 2021 von der Kirche Jesu Christi der Heiligen der Letzten Tage entwickelt wurde. Im Rahmen dieser Arbeit wird die JavaScript-Bibliothek *gedcom7.js* entwickelt, mit der Dateien im Datenformat Gedcom7 gelesen, verändert und geschrieben werden können.

1.1 Anforderungsanalyse & Ziele

Das Ziel dieser Arbeit ist es die grundlegenden Komponenten einer Bibliothek für die Verarbeitung von Dateien im genealogische Austauschformat FamilySearch Gedcom Version 7 zu entwickeln. Da es sich dabei um einen komplexen Datenstandard handelt, liegt der Fokus nicht auf Vollständigkeit, sondern darauf eine Grundlage für weiterführende Arbeiten zu schaffen. Wichtig ist, dass die fundamentalen Funktionalitäten für die Arbeit mit Gedcom7 Dateien bereitgestellt werden und dass diese in einer solchen Form vorliegen, dass einfache Erweiterungen und Vervollständigungen möglich sind.

Folgende Anforderungen werden an die Implementierung gestellt:

- AF01: Dateien im Format Gedcom7 sollen eingelesen werden können
- AF02: Eingelesene Dateien im Format Gedcom7 sollen ausgegeben werden können
- AF03: Eingelesene Dateien sollen gemäß der Gedcom7-Spezifikation verändert und erweitert werden können
- AF04: Neue Dateien im Format Gedcom7 sollen erstellt werden können
- AF05: Die Syntax von Dateien oder Strings soll gemäß der Gedcom7-Spezifikation überprüfbar sein
- AF06: Die in der Gedcom7-Spezifikation definierten Datentypen sollen unterstützt werden
- AF07: Die Bibliothek soll so implementiert und dokumentiert werden, dass sie in weiterführenden Arbeiten erweitert werden kann

1.2 Struktur der Arbeit

Diese Arbeit ist in sechs Kapitel unterteilt. Nach der Einleitung wird der Leser in Kapitel 2 in die Thematik eingeführt und die theoretischen Grundlagen, die für das Verständnis der Arbeit notwendig sind, werden erklärt. Anschließend werden verwandte Implementierungen von Bibliotheken für das Format Gedcom7 vorgestellt, um die Arbeit in den Kontext des aktuellen Entwicklungsstandes zu setzen.

In den folgenden Hauptkapiteln wird das Konzept beschrieben, das im Rahmen dieser Arbeit umgesetzt wurde. Dazu werden die Hauptkomponenten der Bibliothek vorgestellt und die Architektur des entwickelten Systems erklärt. Anschließend wird im Kapitel Implementierung aufgezeigt, wie das zuvor beschriebene Konzpet konkret umgesetzt wurde und wie die fertige Implementierung getestet wurde, um eine korrekte Funktionsweise zu garantieren.

Zum Schluss wird die Ausarbeitung in Kapitel 6 kurz zusammengefasst, indem die wichtigsten Ergebnisse aufgeführt werden und ein Ausblick für zukünftige Arbeiten gegeben wird.

2

Theoretische Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen für das Verständnis dieser Projektarbeit beschrieben. Dazu wird eine kurze Einführung in Genealogie, FamilySearch und Gedcom im speziellen gegeben, um ein Verständnis für die Thematik aufzubauen. Zudem wird auf Quellen verwiesen, aus denen tiefergehende Informationen erschlossen werden können. Im Anschluss dazu wird kurz auf die JavaScript Bibliothek *Nearley* und das JavaScript Testframework *Mocha* eingegangen, die für die Umsetzung dieser Projektarbeit relevant werden.

2.1 Genealogie und FamilySearch

Genealogie ist ein Überbegriff für die Familien- und Ahnenforschung und beschäftigt sich mit der historischen Herkunft und der Geschichte von Menschen weltweit [Ahn]. Dabei sind insbesondere Abstammungs- und Verwandtschaftsverhältnisse von Bedeutung, die anhand valider Quellen in Stammbäumen zusammengefasst werden, die aufzeigen, wie eine Generation mit der nächsten verbunden ist. Auf Basis der so erlangten Erkenntnisse kann eine Familiengeschichte erstellt werden, die eine biographische Studie einer genealogisch nachgewiesenen Familie und der Gemeinde in der sie lebten, darstellt [Gen]. FamilySearch ist eine öffentliche genealogische Datenbank als Teil des Projekts der *Genealogical Society of Utah*, die Menschen dabei hilft, ihre Familiengeschichte durch Webseiten, Apps und persönliche Hilfestellung in über 5000 Family Search Centern auf der ganzen Welt zu erkunden [Fam].

Das Aufkommen des Internets stellte einen Wendepunkt in der Genealogie dar. Die einfachere Kommunikation auch über Landesgrenzen hinweg, ermöglicht es genealogische Informationen auszutauschen und so Gleichgesinnte und Verwandte auf der ganzen Welt zu finden [Ahn]. Um einen standardisierten Austausch genalogischer Informationen zu ermöglichen, entwickelte die Kirche Jesu Christi der Heiligen der Letzten Tage das Datenformat Gedcom, das im folgenden Kapitel vorgestellt wird.

2.2 GEDCOM Version 7

Das Datenformat FamilySearch Gedcom Version 7.0 ist ein einheitliches, flexibles Format für den Austausch von genealogischen Daten, das 2021 von der Kirche Jesu Christi der Heiligen der Letzten Tage entwickelt wurde. Das Ziel besteht darin, eine langfristige Speicherung von genealogischen Informationen zu ermöglichen, die für zukünftige Genealogen und die von ihnen verwendeten System zugänglich und verständlich sind [Fam22]. Die im Rahmen dieser Arbeit verwendete Version 7.0.11 wurde am 01.11.2022 veröffentlicht und stellt die aktuelleste Version des Standards dar[1].

Gedcom7 ist ein UTF-8 kodiertes hierarchisches Containerformat, das die Dateinamenserweiterung *.ged* verwendet. Der erste Character einer Gedcom7-Datei sollte das Byte-Order-Mark (U+FEFF) sein. Der Inhalt einer Gedcom7-Datei ist in sog. *Structures* unterteilt, die aus einem *Structuretype* und einem optionalen *Payload* bestehen und mehrere Substrukturen besitzen können. Hat eine Structure eine *Substructure*, dann ist die Structure die *Superstructure* der Structure. Jede *Substructure* hat genau eine *Superstructure* und ist so eindeutig zugeordnet. Eine Structure, die keine Supertructure besitzt, heißt *Record*. Alle Records zusammen mit einem *Header* und einem *Trailer* bilden ein *Dataset*, das den Inhalt einer Gedcom7-Datei darstellt. [Fam22]

Der Payload einer Structure ist eine Zeichenkette eines bestimmten Datentyps, die entweder Informationen für die Supertructure bereithält, oder einen Zeiger auf eine andere Structure repräsentiert und somit auf diese verweist. Gedcom Version 7.0 definiert 11 verschiedene Datentypen mit denen Namen, Daten, Uhrzeiten, Texte und vieles mehr dargestellt werden können. Der Structuretype ist eindeutig definiert durch eine URI und gibt an, welche Bedeutung und welchen Datentyp die Structure besitzt, welche Substructures enthalten sein können und mit welcher Kardinalität diese auftreten können. [Fam22]

Kodiert wird der Inhalt einer Gedcom7-Datei in sog. *Lines*, die eine Zeichenkettenrepräsentation einer Structure (bzw. eines Teils einer Structure) darstellen und wie in Listing 2.1 aufgebaut sind. Die Bedeutung der einzelnen Bestandteile einer Line ist in Tabelle 5.1 spezifiziert.

```
Level D [Xref D] Tag [D LineValue] EOL
```

Listing 2.1: Aufbau einer Gedcom Line. Eckige Klammern repräsentieren optionale Inhalte.

[1] Stand 28.02.2023

Level	Eine Line beginnt mit einem *Level*, das die Verhältnisse der Structures untereinander beschreibt. Alle Structures mit dem kleinstmöglichen Level 0 sind Records - Level ≥ 1 repräsentieren Substructures. Eine Structure mit dem Level x ist also die Superstructure aller folgenden Structures mit dem Level $x + 1$.
D	*D* steht für *Delimiter*, was englisch für Trennzeichen ist und repräsentiert in diesem Fall das Leerzeichen mit dem Unicode $u + 0020$.
Xref	*Xref* ist die Abkürzung für *Cross-Reference Identifier* und fungiert als Adresse für eine Structure. Möchte man von einer Structure auf eine andere Structure verweisen, kann dies über einen Zeiger-Payload auf die entsprechende Structure realisiert werden.
Tag	Der *Tag* kodiert den *Structuretype* einer Structure.
LineValue	Im *LineValue* einer Struktur ist der Payload kodiert.
EOL	*EOL* steht für *End-Of-Line* und kodiert das Ende einer Line. Im Format Gedcom7 kann dies entweder durch einen *Carriage-Return* (Unicode U+000D), *Line-Feed*(Unicode U+000A) oder einen *Carriage-Return* gefolgt von einem *Line-Feed* repräsentiert werden.

Tabelle 2.1: Bestandteile einer Gedcom Line und ihre Bedeutung

Ein Ausschnitt aus einer Gedcom7-Datei ist in Listing 2.2 dargestellt. Dieser Ausschnitt zeigt einen Record vom Typ *Family*, in dem Informationen über eine Familie gespeichert werden können. Der Familie wurde der Cross-Reference Identifier *@F1@* zugewiesen, sodass im Dataset auf dieses verwiesen werden kann. Der Ehemann und die Ehefrau der Familie (engl. Husband und Wife) sind die Individuen *I1* und *I2*, die ebenfalls in der Gedcom7-Datei definiert sind. Dieser Zusammenhang wird über die Cross-Reference Identifier *@I1@* und *@I2@* ausgedrückt. Außerdem wird ein Family-Event, nämlich die Hochzeit der beiden Ehepartner, aufgeführt und auf den 1.März 1951 datiert. Als letzte Information ist die Anzahl der Kinder (NCHI: Number of Children) mit 2 spezifiziert.

```
0 @F1@ FAM
1 HUSB @I1@
1 WIFE @I2@
1 MARR
2 DATE 1 MAR 1951
1 NCHI 2
```

Listing 2.2: Beispiel für einen Family Record

Detaillierte Erklärungen, alle Informationen zu Structuretypes, Datentypen, usw. und viele weitere Beispiele können in [Fam22] nachgelesen werden.

2.3 Nearley

Nearley.js ist eine JavaScript-Bibliothek zum Parsen kontextfreier Grammatiken (CFGs). Sie bietet einen vielseitigen und effizienten Parsing-Algorithmus, der auf dem Algorithmus von Earley basiert und es ermöglicht, mehrdeutige und rekursive Grammatiken effizient zu behandeln [Chab]. Die Bibliothek ist modular aufgebaut, sodass Benutzer ihre eigenen Parser und Lexer definieren und Parser aus BNF- und EBNF-Grammatiken erzeugen können. Nearley.js ist in reinem JavaScript implementiert und kann in jeder Umgebung ausgeführt werden, die JavaScript unterstützt, einschließlich Webbrowsern und serverseitigen Umgebungen.

Es bietet eine Reihe nützlicher Features, darunter JavaScript-Funktionen, genannt *Postprocessor*, bei denen Benutzer Code angeben können, der ausgeführt wird, wenn bestimmte Teile der Eingabe erkannt werden [Chab]. Diese Funktionen können verwendet werden, um die Ausgabe zu formatieren oder die Eingabe zu validieren.

Kontextfreie Grammatiken (CFGs)

Innerhalb der Theorie der formalen Sprachen wird eine kontextfreie Grammatik, auch als "context-free grammar"(CFG) bezeichnet. Sie enthält nur Ersetzungsregeln, bei denen genau ein Nichtterminalsymbol auf eine beliebig lange Abfolge von Nichtterminal- und Terminalsymbolen abgebildet wird. Die Regeln haben dabei stets die Form $V \rightarrow w$, wobei V ein Nichtterminalsymbol und w eine Zeichenkette aus Nichtterminal- und/oder Terminalsymbolen ist [Chaa].

Die Anwendbarkeit einer Regel auf eine Zeichenkette ist lediglich vom Vorkommen des Nichtterminalsymbols V abhängig, nicht jedoch vom Kontext, in dem es sich befindet. Das bedeutet, dass es irrelevant ist, welche Zeichen sich links oder rechts von dem Nichtterminalsymbol befinden. Somit handelt es sich bei den Ersetzungsregeln um sogenannte "kontextfreie"Regeln.

Algorithmus von Earley

Der Algorithmus von Earley ist ein Parsing-Algorithmus, der kontextfreie Grammatiken parsen kann. Er wurde 1970 von Jay Earley entwickelt. Im Allgemeinen hat das Parsen mit diesem Parsing-Algorithmus eine zeitliche Komplexität, die proportional zu n^3 ist (wobei n die Länge der zu parsenden Zeichenkette ist). Für eindeutige Grammatiken hat der Parser eine Komplexität von n^2. Auf einer großen Menge von Grammatiken, zu der die meisten praktischen kontextfreien Grammatiken von Programmiersprachen gehören, läuft der Parser sogar in linearer Zeit [Ear70]. Der Algorithmus von Earley ist ein Algorithmus der Typ-2-Chomsky-Hierarchie, der kontextfreie Grammatiken mit mehrdeutigen und rekursiven Regeln parsen kann.

Um festzustellen, ob eine kontextfreie Grammatik ein gegebenes Wort erzeugen kann, benötigt der Algorithmus als Eingabe sowohl die Grammatik selbst als auch das Wort, welches aus demselben Alphabet besteht.

Metasyntax und EBNF

Metasyntax und Metasprache beziehen sich auf eine Art von Sprache, die verwendet wird, um die Struktur und Syntax einer anderen Sprache zu beschreiben. Die Metasprache selbst hat ihre eigene Syntax und Grammatik, die zur Beschreibung der Struktur und Syntax der Zielsprache verwendet wird. [Fey16]

Im Kontext der Erweiterten Backus-Naur-Form (EBNF) wird diese als eine formale Metasyntax bezeichnet, da sie dazu dient, kontextfreie Grammatiken darzustellen. Die EBNF ist eine Erweiterung der Backus-Naur-Form (BNF), welche von John Naur, nach seiner Hilfe bei der Entwicklung der Programmiersprache FORTRAN, während seiner Arbeit an ALGOL entwickelt wurde. Basierend auf der Arbeit des Logikers Emil Post erfand Backus eine Notation, , welche präzise, einfach, aber auch performant genug war, um die Syntax jeder Programmiersprache zu beschreiben [Fey16]. EBNF erlaubt es, komplexe Syntax-Regeln in einer leichter verständlichen und kompakteren Schreibweise auszudrücken, indem sie zusätzliche Symbole wie Klammern und Wiederholungszeichen einführt.

EBNF wird ebenfalls in der Spezifikation von Gedcom Version 7.0 verwendet, um z.B Mulitplizitäten auszudrücken.

Beispiel

In Listing 2.3 wird ein einfaches Beispiel einer Nearley Grammatik gezeigt. Die Grammatik besteht aus sogenannten *Token*, in diesem Fall *reihenfolge* und *ergebnis*. Da *reihenfolge* der erste Token ist, ist dies der Einstieg in die Grammatik. Ein Pfeil ($\rightarrow$) wird verwendet, um die Regel eines Tokens zu definieren, das |-Zeichen wird benutzt, um mehrere Alternativen für einen Token zu definieren.

Dementsprechend kann der Token *reihenfolge* den String *Kein Muenzwurf* oder eine beliebige Anzahl von durch Leerzeichen getrennten *ergebnis*-Token enthalten.

Die beliebige Anzahl von *ergebnis*-Token wird durch den EBNF-Operator *:** am Ende der Regel gekennzeichnet. Es gibt neben *:** (0 oder mehr) noch weitere Operatoren, wie z.B. *:+* (1 oder mehr), und *:?* (optional), die in der EBNF-Spezifikation beschrieben werden.

Der Token *ergebnis* kann entweder den String *kopf* oder *zahl* enthalten.
Die Grammatik kann somit entweder den Text *Kein Muenzwurf* oder z.B. die Zeichenkette *kopf kopf zahl kopf* oder Ähnliche parsen.

```
reihenfolge
 -> "Kein Muenzwurf"
    | ergebnis (" " ergebnis):*


ergebnis
 -> "kopf"
    | "zahl"
```

Listing 2.3: Beispiel für eine Nearley Grammatik

2.4 Mocha

Im Rahmen der Bibliotheksimplementierung war es ein zentrales Anliegen, das Testen dieser zu gewährleisten. Zu diesem Zweck musste ein geeignetes JavaScript-Testframework gefunden werden. Mocha ist für diesen Einsatz geeignet, da es eine API bereitstellt, die das einfache Erstellen von Tests ermöglicht. Wie bei vielen anderen Testframeworks können Assertion-Funktionen genutzt werden, um die Tests zu überprüfen. Hierbei bietet Mocha die Möglichkeit, verschiedene Assertion-Frameworks zu nutzen [Moc]. In diesem Projekt wurde das Framework *Chai* verwendet.

Ein einfaches Beispiel für einen Mocha-Test mit Chai-Assertion ist in Listing 2.4 dargestellt. Es wird lediglich überprüft, ob der Wert der Konstante *value* gleich ihrem zugewiesenem Wert 1 ist. Die Funktion *expect().to.equal()* stammt aus der Chai-Bibliothek und überprüft ob der Parameter der Funktion *expect()* dem Parameter der *equal()* Funktion entspricht.

```
describe('test example', function () {
    it('should assert true', function (done) {
        const value = 1;
        expect(value).to.equal(1);
        done();
    });
});
```

Listing 2.4: Beispiel für einen Mocha-Test

Durch die Verwendung von Mocha und Chai können noch weitere Testarten durchgeführt werden, um die Funktionalität von *gedcom7.js* zu überprüfen. Beispielsweise können Fehlerfälle überprüft werden, um zu kontrollieren, dass das Programm korrekt auf unerwartete Eingaben reagiert. Auch die Länge von Arrays kann getestet werden, um zu prüfen, ob das Programm die erwartete Anzahl von Elementen zurückgibt.

Eine weitere wichtige Testart ist die Überprüfung der Gleichheit von Objekten und Dateien. Dies ist besonders wichtig, um sicherzustellen, dass das Programm die erwarteten Ergebnisse liefert und dass Änderungen an der Software nicht zu unerwarteten Fehlern führen.

Durch diese effiziente Möglichkeit Tests zu schreiben, können potenzielle Fehler in der Software schnell erkannt und behoben werden, was zu einer höheren Zuverlässigkeit und Qualität des Programms führt.

3

Related Work

Eine umfassende Untersuchung der bestehenden Gedcom7-Bibliotheken wurde vor Beginn der theoretischen Ausarbeitung der Gedcom7-Bibliothek durchgeführt. Dabei wurden auch einige der beliebtesten Bibliotheken, die in Python oder JavaScript entwickelt wurden, untersucht. Zwei Bibliotheken wurden dabei als Orientierung für die Arbeit herangezogen.

3.1 gedcom7code/js-parser

Der js-parser, welcher von Luther Tychonievich, einem Managing Editor von FamilySearch Gedcom, entwickelt wurde, dient als minimaler Parser für Gedcom7-Zeilen. Dieser Parser, der in JavaScript geschrieben wurde, basiert auf einer Regular Expression und liefert die einzelnen Bestandteile einer Line zurück. Die grundlegende Struktur einer GEDCOM-Line kann damit ermittelt werden.

3.2 python-gedcom

Die zweite Bibliothek, welche interessante Konzepte bezüglich der Handhabung der Gedcom7-Strukturen liefert, ist die *python-gedcom*-Bibliothek. Die an die Gedcom7-Spezifikation angelehnte Klassenhierarchie bot für die Zwecke der Gedcom7-Bibliothek eine ideale Vorlage. Jeder Gedcom-Record besitzt eine eigene Klasse, die von einer Elternklasse Structure erbt.

Ein weiterer Ansatz dieser Bibliothek ist das Speichern der einzelnen Felder in den Klassen für *Record*s oder *Substructures*, die eine Gedcom-Line ausmachen. Dies ermöglicht das konsistente Lesen, Interpretieren, Manipulieren und Schreiben von Gedcom7-Dateien

Bei der python-gedcom-Bibliothek wurde die Prüfung der genauen Spezifikationsvorgaben wenig beachtet. Records können somit beliebig hinzugefügt werden und nicht spezifikationskonforme Records können erstellt und gespeichert werden.

Da die Einhaltung der Spezifikation jedoch sehr wichtig ist, wurde in *gedcom7.js* eine sehr detaillierte Grammatik mittels Nearley.js erstellt. Diese Grammatik wird beim Parsen der Gedcom-Datei und nach jeder Operation an einem Dataset verwendet, um das Dataset auf Korrektheit zu prüfen und somit auch eine korrekte Gedcom-Datei zu gewährleisten.

4

Konzept

Die Bibliothek *gedcom7.js* lässt sich wie in Abbildung 4.1 dargestellt in vier logische Teile gliedern. Das zentrale Element ist der Gedcom Parser, mit dem Dateien oder Strings im in Abschnitt 2.2 vorgestellten Format *Gedcom7* eingelesen werden und mit Hilfe von *Nearley* auf Korrektheit der Syntax überprüft werden können. Die dafür zugrundeliegende Grammatik wird mit Hilfe eines *Grammatik Generators* generiert, der die in [Fam22] definierte Spezifikation in eine Nearley-konforme Syntax überführt. Die so eingelesenen Informationen werden in Gedcom Datenstrukturen gespeichert, die verändert und erweitert werden und anschließend im Format Gedcom7 ausgegeben werden können. In den folgenden Abschnitten werden die vier Teile und das Zusammenspiel dieser in detaillierter Form vorgestellt.

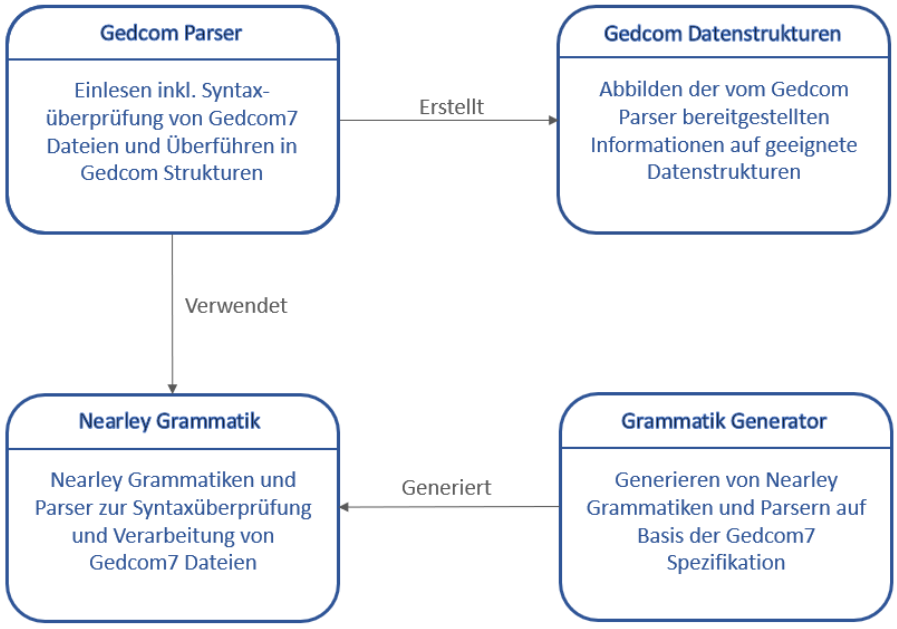

Abbildung 4.1: Allgemeiner Aufbau

4.1 Gedcom Grammatik

Eine wichtige Anforderung an die Bibliothek *gedcom7.js* ist, dass die Syntax von Dateien oder Strings, die eingelesen werden, gemäß der Gedcom7-Spezifikation überprüfbar sein soll. Da die Gedcom Datenstrukturen veränderbar und erweiterbar sein sollen, ist es wichtig, dass eine Syntaxüberprüfung nach Änderungen auf einfache Weise möglich ist. Umgesetzt wird diese Syntaxüberprüfung mit Hilfe des in Kapitel 2.3 vorgestellten JavaScript-Parser-Toolkits *Nearley*. Mit *Nearley* können auf einfach Weise, menschenlesbare Grammatiken erstellt und zu einem *Nearley-Parser* kompiliert werden. Von großem Vorteil ist dabei, dass Features wie Postprozessoren und die Implementierung eines Lexers unterstützt werden.

4.1.1 Pre- und Postprozessor

Standardmäßig gibt ein *Nearley-Parser* jedes Zeichen, das mit einer Regel übereinstimmt, als Array zurück [Chab]. Bei komplexeren Grammatiken wie der Gedcom7 Spezifikation führt dies dazu, dass sehr viele Arrays ineinander verschachtelt werden, sodass schnell zweistellige Verschatlungsgrade erreicht werden, was ein Weiterarbeiten mit den Ergebnissen erschwert. Mit Hilfe von Postprozessoren können jeder *Nearley Regel* Verarbeitungsanweisungen zugewiesen werden, sodass die Ergebnisse beispielsweise im JSON-Format zurückgegeben werden. Auf diese Weise können die eingelesenen Dateien bereits bei der Syntaxüberprüfung in eine passende Darstellungsform gebracht werden, sodass eine leichte Überführung in die passende Gedcom Datenstruktur möglich ist.

Desweiteren kann ein Lexer verwendet werden, um die Arbeit mit *Nearley* zu optimieren. Ein *Nearley-Parser* teilt die Eingabedaten standardmäßig in einen Strom von einzelnen Zeichen, die sequentiell abgearbeitet werden, was auch als *Scannerless Parsing* bezeichnet wird [Chab]. Ein Lexer ist eine Art Preprozessor, der die Eingabedaten in größere Einheiten, die sog. *Tokens* zusammenfasst [Chab]. Auf diese Weise wird der Aufwand beim Parsen verringert und die Interpretation der Eingabedaten fällt oft leichter. Ein einfaches Beispiel hierfür ist eine Regel die einen Zahlenwert erwartet. Ist der Eingabewert beispielsweise "137", würde ein *Nearley-Parser* standardmäßig jede Ziffer einzeln einlesen und im Postprozessor müsste definiert werden, dass die aufeinanderfolgenden Ziffern als ein Zahlenwert interpretiert werden sollen. Mit Hilfe eines Lexers könnte eine einfache Regel definiert werden, die den kompletten Zahlenwert als ein Token vorverarbeitet. Im Rahmen dieser Arbeit wurde der JavaScript Lexer *Moo.js* [Rad] verwendet. *Moo.js* zeichnet sich durch seine Geschwindigkeit aus und wird von *Nearley* als Lexer unterstützt.[1]

[1] Laut den Entwicklern ist Moo.js der schnellste JavaScript-Lexer und ∼2-10 mal schneller als herkömmliche Lexer [Rad].

4.1.2 Nearley-Parser für Gedcom7

Da sowohl die Gedcom7- als auch die Nearley Syntax auf EBNF-Sprachkonzepten basieren, lässt sich die Gedcom7 Spezifikation ohne weiteres in eine Nearley Grammatik übersetzten, die dann zu einem Nearley-Parser kompiliert werden kann. Wird diesem Nearley-Parser eine Gedcom7-Datei (.ged) kodiert als UTF-8 Zeichenkette übergeben, erfüllt dieser die folgenden zwei Aufgaben:

1. Überprüfung der Gedcom7 Syntax

Der Nearley-Parser überprüft den übergebenen Gedcom7-String Line für Line, indem er alle Zeichen (bzw. Tokens) sequentiell liest, bis ein End-Of-Line (EOL) Token gefunden wird. Nach jedem Zeichen das eingelesen wird, überprüft der Parser, welche in der Grammatik definierten Regeln durch das neu eingelesene Zeichen nicht mehr mit der Zeichenkette übereinstimmen und verwirft diese. Wird ein EOL Token gelesen werden die Postprozessoren aller übereinstimmenden Regeln ausgeführt und ein Array mit den Ergebnissen dieser Postprozessoraufrufe als Ergebnis der Line zurückgegeben. Da die Gedcom7 Grammatik nicht mehrdeutig ist, findet der Parser bei korrekter Gedcom7 Syntax immer ein eindeutiges Ergebnis (d.h. beim Erreichen des EOL Tokens ist maximal eine übereinstimmende Regel übrig). Werden bei diesem Prozess alle Regeln der Grammatik ausgeschlossen, bevor ein EOL Token gelesen wird, ist die Syntax des übergebenen Gedcom7-Strings nicht korrekt und das Parsen wird mit einem Syntaxfehler abgebrochen. Da die Zeichenkette sequentiell abgearbeitet wird, kann bei auftretendem Fehler genau aufgezeigt werden, welche Line und welches Zeichen fehlerhaft sind.

2. Extrahieren der Structure-Informationen

Eine weitere Aufgabe des Nearley-Parsers ist es, die Structure-Informationen des Gedcom7-Strings zu extrahieren, sodass im nächsten Schritt eine einfache Überführung in entsprechende Gedcom Datenstrukturen möglich ist. Durch den sequentiellen Aufbau einer Gedcom7 Datei wird eine Structure immer vor ihren Substructures definiert. Da das erste Token jeder Line stets das Level der Line repräsentiert, kann der Nearley-Parser die Abhängigkeiten der Lines zueinander zuordnen und es ist zu jedem Zeitpunkt eindeutig, welcher Superstructure eine Structure zugeordnet werden soll. Folgende Informationen können also durch den Nearley-Parser extrahiert werden:

- **Inhalt der Line**: Die in Abschnitt 2.2 vorgestellten Bestandteile einer Line können ausgelesen und zugeordnet werden.
- **Datentyp**: Sofern ein Payload in der Line vorhanden ist, kann mit Hilfe der URI der Datentyp des Payloads bestimmt werden
- **Superstructure**: Zu jeder Line kann die entsprechende Superstructure angegeben werden, sofern es sich nicht um einen Record (Structure mit Level 0) handelt, die keine Superstructure besitzen
- **Substructures**: Hat eine Sturktur eine oder mehrere Substructures, können diese auf Basis des Levels der Lines bestimmt werden

4.2 Grammatik Generator

In der Gedcom7 Spezifikation werden 181 Structuretypes verteilt auf 7 Records definiert, die alle in einer Line der Form

```
Level D [Xref D] Tag [D LineVal] EOL
```

dargestellt werden. Sollen diese Structuretypes in eine Nearley Grammatik überführt werden, muss für jede dieser Structures und jede mögliche Kombination an Substructures eine Regel erstellt werden. Da dies eine sehr repetitive Aufgabe ist und sich die Regeln nur an bestimmten Stellen unterscheiden, lässt sich die Grammatikerstellung durch einen Grammatik Generator automatisieren. Dazu können Definitionsdateien erstellt werden, die die für alle Structuretypes die folgenden Informationen bereithalten:

- **URI**: Die URI des Structuretypes wird benötigt, um eine Structure eindeutig zuordnen zu können
- **LineType**: Der LineType gibt an, wie die Line aufgebaut ist - also ob bspw. ein Cross-Reference-Identifier oder ein Payload vorhanden sind
- **Datatype**: Sofern ein Payload in der Line vorhanden ist, kann über den Datatype die Syntax des Payloads ermittelt werden
- **Tag**: Der Tag wird benötigt, um die Line einer Nearley Regeln zuordnen zu können
- **Substructures**: In der Gedcom7 Spezifikation sind für alle Structuretypes alle zulässigen Substructures definiert. Mit dieser Information können alle syntaktisch korrekten Structures in Nearley Regeln abgebildet werden.
- **Level**: Um eine eindeutige Grammatik zu generieren, müssen die Level mit denen ein jeweiliger Structuretype auftreten kann, zwingend mit angegeben werden. Da in der Gedcom7 Spezifikation Tags mehrfach für verschiedene Structuretypes verwendet werden, kann nicht einfach ein generisches Level für die Regeln verwendet werden, das ganzzahlige Werte akzeptiert, da die entstehende Grammatik damit mehrdeutig wäre. Ein Beispiel hierfür sind die Structures *g7:HEAD-DATE* und *g7:DATE-exact* im Gedcom Header. Mit einem generischen Level wären die Regeln für beide Structuretypes identisch mit

  ```
  Level D "DATE" D DateExact EOL
  ```

  Wird eine solche Line als Substructure eines Header Records von dem Nearley Parser gelesen, kann dieser nicht entscheiden, ob es sich um ein *g7:HEAD-DATE* oder ein *g7:DATE-exact* handelt und würde somit zwei Ergebnisse aufrecht erhalten. Um diese Mehrdeutigkeit zu verhindern, wird das Level in der Definition angegeben. Da die Structure *g7:HEAD-DATE* mit dem Level +1 und *g7:DATE-exact* mit Level +3 bezogen auf den zugrundeliegenden Header Record vorliegen, können die Lines vom Parser eindeutig unterschieden werden.

Anhand dieser Informationen kann der Grammatik Generator automatisiert Nearley Regeln formulieren. Diese Regeln können zu einer Grammatik zusammengefasst und anschließend vom Generator zu einem Nearley-Parser kompiliert werden. Anhand des LineTypes kann der Generator den Regeln die passenden Postprozessoren zuweisen, die für das Extrahieren der Structure Informationen zuständig sind. Auf diese Weise kann ein voll funktionaler Nearley-Parser automatisiert generiert werden, der die Gedcom7-Syntax vollständig parsen und alle für die weitere Verarbeitung benötigten Informationen extrahieren kann.

Ein weiterer großer Vorteil an dieser Automatisierung ist, dass zur Erfüllung der Anforderung der einfachen Erweiterbarkeit der Bibliothek beigetragen wird. Sollte die Grammatik in zukünftigen Projekten erweitert werden, bspw. wenn in einer neuen Version des Gedcom Standards weitere Structuretypes definiert werden, ist dies auf einfache und verständliche Weise durch das Hinzufügen neuer Einträge in die Definitionsdatei des Grammatik Generators möglich.

Desweiteren bildet der Grammatik Generator ein Fundament für einen wichtigen Use-Case, der in weiterführenden Arbeiten adressiert werden sollte: der Möglichkeit Extensions zu definieren. Die Gedcom7 Spezifikation definiert die wichtigsten Strukturen zur Speicherung genealogischer Informationen - für alle Informationen, die über diese Standardstrukturen hinausgehen, müssen Extensions definiert werden. Da genealogische Informationen sehr vielfältig sein können, sind Extensions ein probates Mittel, dass in vielen Anwendungen genutzt wird. Mit Hilfe des Grammatik Generators kann die Definition von Extensions umgesetzt werden, indem eine Schnittstelle zum Generator entwickelt wird, die dem Benutzer zur Verfügung gestellt wird. Über diese Schnittstelle kann die Definitionsdatei erweitert und anschließend die Grammatik neu generiert und kompiliert werden. Auf diese Weise könnte die Bibliothek auf die Anforderung aller Benutzer angepasst werden.

4.3 Gedcom Datentrukturen

Die zentrale Struktur in einer Gedcom7 Datei ist das sog. *Dataset*. Jedes Dataset muss mit einer *Header-Structure* beginnen, der Metadaten über das gesamte Dataset beinhaltet und dabei u.a. Aussagen über den Ort und Zeitpunkt der Erstellung und den Ersteller des Datasets selbst machen kann. Die Mindestanforderung an den Header ist, dass die verwendete Gedcom Version in einer dafür vorgesehenen Structure spezifiziert ist. Abgeschlossen wird jedes Dataset mit einer *Trailer Line*, die das Ende des Datasets repräsentiert. Eine minimales Gedcom7 Dataset sieht also wie folgt aus:

```
0 HEAD
1 GEDC
2 VERS 7.0
0 TRLR
```

Listing 4.1: Minimales Gedcom7 Dataset

Alle genealogischen Informationen können in einem oder mehreren Records festgehalten werden. Folgende Records sind in der Gedcom7 Spezifikation definiert:

- **Family (FAM)**: Der Family Record wurde ursprünglich so strukturiert, dass er eine Familie mit einem männlichen Ehemann und einer weiblichen Ehefrau repräsentiert. Um die Migration von bestehenden Gedcom-Dateien auf Gedcom7 zu erleichtern, wurde die Benamung der Strukturen beibehalten. Trotzdem sollen in Gedcom7 Familien, Heirat, Zusammenleben und Adoption unabhängig vom Geschlecht der Partner angegeben werden können und daher das Geschlecht und die Rollen von Partnern nicht aus der Husband- bzw. Wife Struktur abgeleitet werden.
- **Individual (INDI)**: Zusammenstellung von Fakten und Hypothesen über eine Person. Diese können aus verschiedenen Quellen stammen, die durch Quellenangaben dokumentiert werden können.
- **Multimedia (OBJE)**: Eine Referenz zu einer oder mehrerer digitaler Dateien, angereichert mit Informationen über den Inhalt und den Typ der Datei.
- **Repository (REPO)**: Beinhaltet Informationen über Personen oder Institutionen, die eine Sammlung von Quellen besitzen.
- **Shared Note (SNOTE)**: Eine Sammlung von Informationen, die nicht vollständig in andere Strukturen passen. Beispiele wären Forschungsnotizen, alternative Interpretationen oder Argumentationen
- **Source (SOUR)**: Beschreibt eine Quelle, indem auf bestimmte Dokumente oder Verzeichnisse verwiesen wird.
- **Submitter (SUBM)**: Beschreibt eine Person oder eine Institution, die im Dataset enthaltene Informationen beigesteuert hat.

In der Gedcom7 Spezifikation ist festgelegt, welche Structures Teil eines bestimmten Records sein dürfen. Beispielsweise darf eine *HUSB*-Structure nur in einem Family Record vorliegen - ein *CREATION_DATE* dahingegen kann für alle Records definiert werden. Alle Lines, die in einem Gedcom7 Dataset vorliegen sind Structures, die über Super- und Substructures zueinander ins Verhältnis gesetzt werden. Auch ein Record ist eine Structure, jedoch mit der Besonderheit, dass ein Record im Gegensatz zu allen anderen Structures keine Superstructure besitzt. Die genauen Verhältnisse dieser Datenstrukturen ist in Abbildung 4.2 dargestellt. Im folgenden werden die Datenstrukturen, mit denen eine Gedcom7 Datei in der Bibliothek *gedcom7.js* abgebildet wird, vorgestellt.

4.3.1 Structure

In der Bibliothek *gedcom7.js* wird jede Line, die vom Parser gelesen wird, in Form einer Structure abgebildet (siehe Abbildung 4.2). Diese Datenstruktur hält alle Informationen wie Level, Tag und LineValue der Line bereit und enthält zudem Verweise auf die Superstructure und alle Substructures. Eine wichtige Anforderung ist zudem, dass die eingelesene Gedcom7 Datei veränderbar und erweiterbar sein soll. Daher enthält die *Structure* Datenstruktur Methoden mit der Substructures hinzugefügt bzw. entfernt werden können und mit denen der Payload der Structure verändert werden kann. Um die veränderte Datenstruktur als Gedcom7 Datei speichern zu können, muss jede Structure als Gedcom7 konforme Line ausgegeben werden können. Außerdem werden zwei spezielle Structures definiert, die *Datatype Structures* und *Records*.

1. Records

Die in der Einleitung dieses Kapitels aufgeführten Records, wie z.B. der Family Record, sind die Superstructures aller weiteren Structuretypes und haben somit besondere Anforderungen. Daher werden in *gedcom7.js* alle Records in Form einer eigenen Record Datenstruktur abgebildet, die *Structure* erweitert. Dabei sollen Methoden bereitgestellt werden, mit denen die Informationen, die in den Substructures enthalten sind, extrahiert werden können. Ein Beispiel hierfür wäre eine Methode, die alle Informationen über die Residenz einer Familie, die über viele Substructures verteilt sind, zusammenfasst und in übersichtlichem Format zurückgibt. In einem Gedcom7 Record kann zudem über eine *Restriction Structure* der Zugriff auf Informationen dieses Records eingeschränkt werden. Die Record Datenstruktur sollte diese Einschränkungen verwalten und die Ausgabe von Informationen dementsprechend anpassen. Außerdem sollen alle Records Möglichkeiten zur Syntaxüberprüfung des Records und all seiner Substructures besitzen, sodass nach dem Hinzufügen einer Structure in einen Record, die Syntax des Records überprüft werden kann, um zu entscheiden, ob das Hinzufügen syntaktisch korrekt war. Dies bietet den Vorteil, dass nicht jedes Mal die Syntax des gesamten Datasets überprüft werden muss, obwohl sich nur ein Record verändert hat.

2. Datatype Structures

Der Payload von Lines kann verschiedene Datentypen haben, die in der Gedcom7 Datei als Zeichenkette kodiert sind. Handelt es sich dabei um einen einfachen LineValue vom Datentyp *Text* ist es ausreichend, diesen als Zeichenkette in der Structure zu hinterlegen. Bei komplexeren Datentypen wie *Date* ist es jedoch notwendig, die Structure um Methoden und Eigenschaften zu erweitern, die die weitere Verarbeitung erleichtern. Daher werden für komplexere Datentypen wie *Date* oder *Personal Name* spezielle *Datatype Structures* bereitgestellt, mit denen beispielsweise das Datum eines Events in ein JavaScript konformes Date-Objekt überführt werden kann.

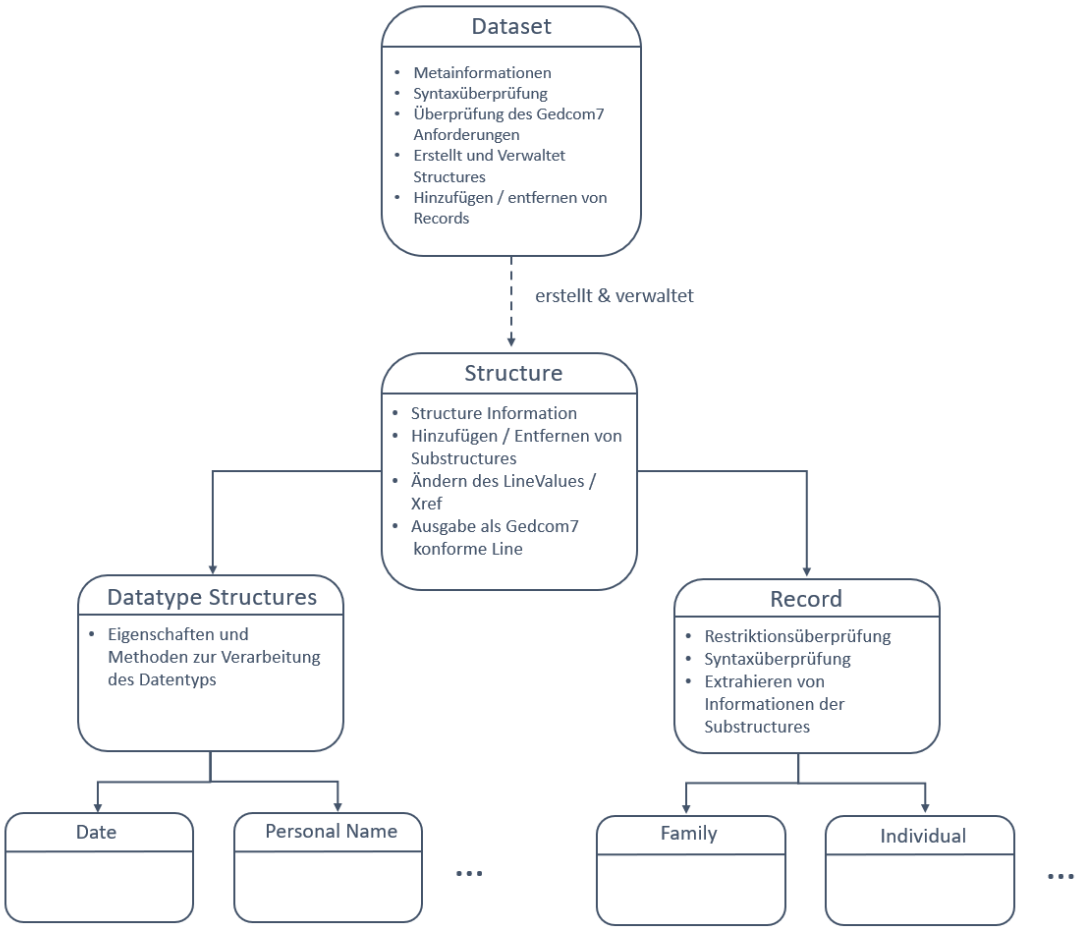

Abbildung 4.2: Gedcom Strukturen

4.3.2 Dataset

Die in Abschnitt 4.3.1 beschriebenen Structures werden in einer *Dataset* Datenstruktur zusammengefasst, die alle genealogischen Informationen einer Gedcom7 Datei enthält. Die Hauptaufgabe des Datasets besteht darin, Structures zu erstellen und zu verwalten. Wird eine Gedcom7 Datei mit korrekter Syntax mit dem in 4.1.2 vorgestellten Nearley Parser eingelesen, extrahiert dieser alle Structure Informationen. Anschließend kann ein *Dataset* erstellt werden, das all diese Informationen einliest, daraus Structures erstellt und die Zusammenhänge zwischen diesen Structures modelliert, sodass, eine Art Baumstruktur mit allen Records entsteht. Um ein Dataset mit neuen genealogischen Informationen anzureichern, sollen Methoden zum Hinzufügen bzw. zum Entfernen von Records bereitgestellt werden. Da das Hinzufügen bzw. Entfernen von Strukturen zu einer inkorrekten Gedcom7 Syntax führen kann, müssen Methoden zur Syntaxüberprüfung implementiert werden. Des Weiteren stellt das Dataset Metainformationen über eine Gedcom7 Datei zur Verfügung und sollte bestimmte Anforderungen überprüfen, die in der Ged-

com7 Spezifikation angegeben werden. Beispiele hierfür sind, dass jede Gedcom7 Datei mit dem *Byte-Order-Mark* beginnen sollte oder dass alle Structure, auf die über einen Cross-Reference-Identifier verwiesen wird, definiert seien müssen, bevor auf diese verwiesen wird.

4.4 Gedcom Parser

Die in diesem Kapitel vorgestellten Konzepte und Datenstrukturen werden alle im Gedcom Parser vereinigt, der die zentrale Instanz der Bibliothek *gedcom7.js* darstellt. Abbildung 4.3 zeigt ein Sequenzdiagramm, das den allgemeinen Ablauf beim Einlesen einer Gedcom7 Datei mit dem Gedcom Parser zeigt.
Der Gedcom Parser ließt eine Gedcom7 Datei ein und konvertiert diese in eine Zeichenkette. Die Zeichenkette kann dann an einen Nearley Parser übergeben werden, der diese Line für Line ließt, die Syntax überprüft und dabei die Structure Informationen extrahiert. Ist die Syntax der Gedcom7 Datei korrekt, werden die gesammelten Structure Informationen an den Gedcom Parser zurückgegeben - anderenfalls wird das Einlesen mit einer Fehlermeldung beendet. Anschließend überprüft der Gedcom Parser die Kardinalität der eingelesenen Structures (beispielsweise darf nur eine *HUSB*-Struktur pro Family Record enthalten sein).[2]. Sofern keine Fehler bei der Kardinalitätsüberprüfung gefunden werden, wird ein neues Dataset erstellt und die von Nearley extrahierten Structure Informationen an das Dataset übergeben.

Das Dataset generiert den Header und den Trailer, der in jedem Dataset vorhanden sein muss und erstellt anschließend alle Structures auf Basis der Structure Informationen. Dazu wird jeder Eintrag der Structure Informationen auf den Structure Type untersucht (Record, Datatype Structure oder allgemeine Structure) und auf Basis dessen eine Structure mit allen Informationen erstellt. Diese Structure wird dann in das Dataset eingegliedert, indem die entsprechende Superstructure und alle Substructures zugewiesen werden. Sind alle Structures erstellt, wird überprüft, ob dass alle Cross-Reference-Identifier, auf die im Dataset verwiesen wird auch innerhalb des Datasets definiert sind. Ist dies der Fall, wird das Dataset zurückgegeben. Dieses Dataset kann dann wie in Abschnitt 4.3 beschrieben verändert und erweitert werden.

[2] Die Kardinalitätsüberprüfung wurde in den Gedcom Parser ausgelagert, da Nearley ein Streaming-Parser ist und somit zu keinem Zeitpunkt weiß, ob noch weitere Eingaben zu erwarten sind. Daher werden Konzepte wie Kardinalitätsüberprüfungen nicht von Nearley unterstützt. [Chab]

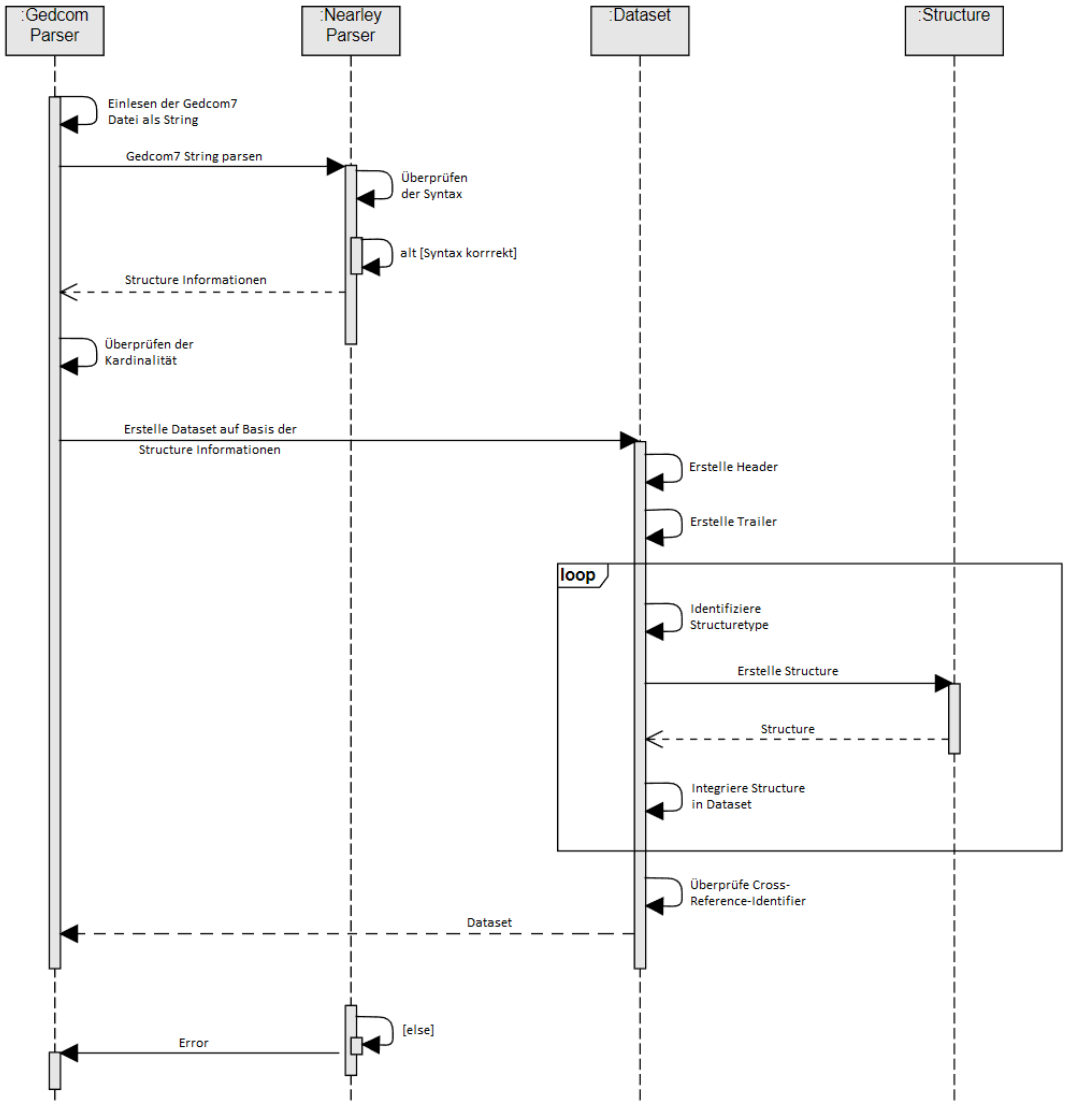

Abbildung 4.3: Ablauf Gedcom Parser

5

Implementierung & Test

In diesem Kapitel wird beschrieben wie das im Kapitel 4 vorgestellte Konzept in der Bibliothek *gedcom7.js* implementiert wird. Dazu wird zuerst darauf eingegangen, wie die Gedcom7 Spezifikation in der Nearley Grammatik abgebildet wird und wie diese Grammatikerstellung automatisiert werden kann. Anschließend wird dargestellt, wie die in Abschnitt 4.3 vorgestellten Gedcom Datenstrukturen implementiert wurden. Es wird gezeigt, wie diese Komponenten im Gedcom Parser vereinigt werden und in einem detaillierten Beispiel wird die Verwendung des Parsers demonstriert. Zum Schluss wird darauf eingegangen, wie die Bibliothek und die darin enthalten Funktionen getestet wurden.

5.1 Gedcom Grammatik

Da die Gedcom7- sowie die Nearley Syntax beide auf EBNF-Sprachkonzepten basieren, lässt sich die Gedcom7 Spezifikation ohne weiteres in eine Nearley Grammatik übersetzten. Im folgenden wird gezeigt, wie die Grammatik erstellt wurde und welche Postprozessoren verwendet wurden.

5.1.1 Gedcom7 Syntax in Nearley

Um Nearley Regeln für eine Gedcom Line zu definieren, können die folgenden Tokens für das Leerzeichen, den *Cross-Reference Identifier* und die End-Of-Line Zeichenfolge in Form von regulären Ausdrücken definiert werden.

```
D    : /[ ]/
Xref : /\@[A-Z0-9\_]+\@/
EOL  : /(?:\r\n?|\n)/
```

Listing 5.1: Tokens für eine Gedcom Line, definiert als regulärer Ausdruck

Diese regulären Ausdrücke werden in der Vorverarbeitungsphase vom Moo-Lexer verwendet, um zusammenhängende Zeichen zu Tokens zu gruppieren, die dann in der Nearley Grammatik über den Tokennamen mit einem vorangestellten %-Zeichen angesprochen werden können. Soll nun die erste Line eines Family-Records geparsed werden, könnte dies mit der folgenden Nearley-Regel umgesetzt werden.

```
record_FAM -> "0" %D %Xref %D  "FAM"  %EOL
```

Listing 5.2: Nearley Regel zum parsen eines Family Records

Diese Regel akzeptiert eine Line mit dem Level 0, einem syntaktisch korrekten Cross-Reference-Identifier, dem Tag *FAM* gefolgt von einem EOL-Token. Getrennt werden die Bestandteile durch ein Leerzeichen. Sollen nun ebenfalls HUSB- und WIFE Structures als Substructures des Family Records akzeptiert werden, könnte die Nearley Grammatik wie folgt erweitert werden.

```
record_FAM
    -> "0" %D  %Xref %D  "FAM"  %EOL
    |  record_FAM record_FAM_Substructs:+


record_FAM_Substructs
    -> "1" %D  "HUSB" %D  %Xref %EOL
    |  "1" %D  "WIFE" %D  %Xref %EOL
```

Listing 5.3: Nearley Regel zum parsen eines Family Records mit HUSB- und WIFE Substructures

Auf diese Weise würde der Nearley Parser einen Family Record ohne Substructures und einen Family Record mit beliebig vielen HUSB- und WIFE Structures als Substructures als Eingabe akzeptieren. Für die HUSB- und WIFE Structure ist als Payload ein Cross-Reference-Identifier angegeben, da in diesen Structures auf ein *Individual* Record verwiesen wird.

Sollen nun die weiteren Lines aus Listing 2.2 ebenfalls in die Grammatik aufgenommen werden, müssen Regeln für die Datentypen der Payloads des MARR-Events und der NCHI-Structure definiert werden. Die Anzahl der Kinder wird als *Integer* Datentyp kodiert, also ein Folge von Dezimalziffern. Nach der Gedcom7 Spezifikation dürfen *Integer* Werte nicht leer sein und führende Nullen sind erlaubt, sollten aber vermieden werden. Eine Regel für den Datentyp *Integer* kann also wie in Listing 5.4 dargestellt, formuliert werden.

```
digit   -> [0-9]
Integer -> digit:+
```

Listing 5.4: Nearley Regel für den Datentyp *Integer*

Für das *MARR*-Event, also die Hochzeit der Ehepartner der Familie, ist eine *DATE*-Structure zur Spezifikation des Datums der Hochzeit hinterlegt. Dieses Datum wird mit dem Datentyp *DateValue* kodiert, der im Gegensatz zum *Integer* wesentlich mehr Regeln umfasst. Ein *DateValue* kann auf vier verschiedene Weisen dargestellt werden:

1. *date*: "JULIAN 13 MAR 1998"
   Ein mehr oder weniger genau spezifiziertes Datum
2. *datePeriod*: "FROM 15 FEB 2001 TO 23 MAR 2001"
   Ein Zeitintervall, dass von einem Startdatum bis zu einem Enddatum angegeben wird
3. *dateRange*: "BET 15 FEB 2001 AND 23 MAR 2001"
   Ein ungenaueres Zeitintervall, bei dem nur Grenzen angegeben werden
4. *dateApprox*: "ABT 15 FEB 2001"
   Eine Schätzung des Datums (ABT x: genaues Datum unbekannt, aber nahe x)

Diese Zusammenhänge ergeben die in Listing 5.5 dargestellten Nearley Regeln für die Definition des Datentyps *DateValue*.

```
DateValue  -> (date | DatePeriod | dateRange | dateApprox):?

date       -> (calendar D):?
              ((day D):? month D):?
              year
              (D epoch):?
datePeriod -> ("FROM" D date D):? "TO" D date
dateApprox -> ("ABT" | "CAL" | "EST") D date
dateRange  -> "BET" D date D "AND" D date
              |  "AFT" D date
              |  "BEF" D date

calendar   -> "GREGORIAN" | "JULIAN" | "FRENCH_R" | "HEBREW"
day        -> Integer
year       -> Integer
month      -> Tag
```

```
epoch       ->  "BCE" | Tag
Tag         ->  upperCaseLetter | digit |  underscore
```

Listing 5.5: Nearley Regel für den Datentyp *DateValue*

Werden all diese Regeln zusammengefasst lässt sich eine Grammatik definieren, die den Family Record aus Listing 2.2 als Eingabe akzeptiert. Diese Grammatik ist in Listing 5.6 dargestellt.

Mit diesem Vorgehen können Nearley Regeln für alle Datentypen, Structures und Records definiert werden, die zu einer Grammatik für die Syntaxüberprüfung von Gedcom7 Dateien zusammengesetzt werden können.

```
record_FAM_Substructs
    -> "0" %D  %Xref  %D  "FAM"  %EOL
    |  record_FAM record_FAM_Substructs:+

record_FAM_Substructs
    -> "1" %D  "HUSB" %D  %Xref  %EOL
    |  "1" %D  "WIFE" %D  %Xref  %EOL
    |  "1" %D  "NCHI" %D  Integer %EOL
    |  structure_MARR

structure_MARR
    -> "1" %D  "MARR" %EOL
    |  structure_MARR

structure_DATE
    -> "2" %D "DATE" %D  DateValue %EOL
```

Listing 5.6: Nearley Grammatik für den Family Record aus Listing 2.2

5.1.2 Nearley Postprozessoren

Mit Hilfe von Postprozessoren können jeder Nearley Regel Verarbeitungsanweisungen zugewiesen werden. Für die Bibliothek *gedcom7.js* werden die folgenden drei Nearley Postprozessoren implementiert.

1. joinAndUnpackAll():

Wie in Abschnitt 4.1.1 beschrieben, überführt ein *Nearley-Parser* jedes Zeichen, das mit einer Regel übereinstimmt, in ein Array. Bei komplexeren Grammatiken wie der Gedcom7 Spezifikation führt dies dazu, dass sehr viele Arrays ineinander verschachtelt werden, sodass schnell hohe Verschachtelungsgrade erreicht werden. Ein Beispiel hierfür wäre der in Abschnitt 5.1.1 definierte Datentyp *DateValue*. Hier würde jeder Bestandteil eines DateValues in ein eigenes Array verschachtelt werden. Wird beispielsweise das Datum

GREGORIAN 13 MAR 1998

ohne Postprozessoren verarbeitet, wird das Array

[GREGORIAN, , [13, , [MAR, , [1998]]]]

zurückgegeben, dass eine Weiterverarbeitung sehr umständlich macht. Daher wird der Postprozessor *joinAndUnpackAll()* implementiert, der über die JavaScript Funktion *flat()* alle Elemente des Arrays rekursiv verkettet und anschließend über die Funktion *join()* zu einer Zeichenkette zusammenfügt. Wird dieser Postprozessor einem Datentyp wie *DateValue* zugewiesen, wird jedes syntaktisch korrekte Datum in passendem Format zurückgegeben und kann so direkt als LineValue für die weitere Verarbeitung verwendet werden. Die in Listing 5.5 definierte Regel würde sich ergeben zu

```
DateValue
    -> (date | DatePeriod | dateRange | dateApprox):?
        {% postprocessor.joinAndUnpackAll %}
```

Listing 5.7: Erweiterung der Nearley Regel für den Datentyp *DateValue*

2. createStructure():

Der Postprozessor *createStructure()* wird verwendet, um die gelesene Line mit Structure Informationen anzureichern. In der Nearley Regel wird die Line selbst, der Typ, die in der Gedcom7 Spezifikation definierte URI der Line und die Structures bei denen eine Kardinalitätsüberprüfung notwendig ist an den Postprozessor übergeben. Für den in Listing 5.6 vorgestellten Family Record ergibt sich die Nearley Regel mit Postprozessoraufruf wie folgt:

```
record_FAM
    -> "0" %D %Xref %D "FAM" %EOL
        {% (line) => postprocessor.createStructure({
            line: line,
```

```
            uri: "g7_record_FAM",
            type: "FAM_RECORD",
            checkCardinalityOf: {
                "1_g7_FAM_HUSB":"0:1",
                "1_g7_FAM_WIFE":"0:1",
            }
        }) %}
```

Listing 5.8: Nearley Regel zum parsen eines Family Records mit Postprozessor

Mit dem Parameter *checkCardinalityOf* werden die URIs aller Structures angegeben, bei denen eine Kardinalitätsüberprüfung notwendig ist und ihnen wird die in der Gedcom7 Spezifikation definierte Kardinalität als Wert zugewiesen. Kardinalitätsüberprüfungen sind bei allen Substructures erforderlich, die für die Superstructure als notwendig definiert wurden (1:1 und 1:M) oder für die eine Maximale Anzahl festgelegt ist (also 0:1 und 1:1). In der Funktion *createStructure()* werden alle Informationen abhängig vom übergebenen Type zusammengefasst und als JavaScript Objekt an den Parser zurückgegeben. Für einen Family Record ergibt sich die Funktion zu:

```
createStructure: (params) => {
    // create line object depending on type of line
    let lineObject = {};
    lineObject = {
        level: line[0],
        xref: line[2],
        tag: line[4],
        lineVal: '',
        EOL: line[5]
    };

    // return data object with structure information
    return {
        uri: params.uri,
        line: lineObject,
        type: params.type,
        lineValType: params.lineValType || null,
        superstructFound: false,
```

```
            substructs: [],
            checkCardinalityOf: params.checkCardinalityOf
        };
    }
```

Listing 5.9: Postprocessor *createStructure()* für einen Family Record

3. addSubstructure():

Der Postprozessor *addSubstructure()* wird verwendet, um die mit *createStructure()* erstellten Structures miteinander zu verbinden und die Verhältnisse zwischen Superstructures und Substructures abzubilden. Im Falle des Family Records würde sich die in Listing 5.10 dargestellte Regel ergeben.

```
record_FAM
    -> "0" %D %Xref %D "FAM" %EOL
        {% (line) => postprocessor.createStructure({
                line: line,
                uri: "g7_record_FAM",
                type: "FAM_RECORD",
                checkCardinalityOf: {
                    "1_g7_FAM_HUSB":"0:1",
                    "1_g7_FAM_WIFE":"0:1",
                }
            })
        %}

    | record_FAM record_FAM_Substructs:+
        {% (line) => postprocessor.addSubstructure({
                superstruct: line[0],
                substructs: line[1]
            })
        %}
```

Listing 5.10: Vollständige Nearley Regel zum parsen eines Family Records mit Substructures

Äquivalent zu Listing 5.8 wird für die erste Line des Family Records der *createStructure()* Postprozessor aufgerufen. Für alle Substructures die gefunden werden, wird ebenfalls der *createStructure()* Postprozessor aufgerufen (die Regel für die Substructures ist aus Platzgründen nicht in Listing 5.10 aufgeführt, kann aber äquivalent zu der Regel für den Family Record mit leicht veränderten Parametern definiert werden). Sind alle Substructures gefunden, wird der Postprozessor *addSubstructure()* aufgerufen, der die Abhängigkeitsverhältnisse der Structures zueinander abbildet. Dazu werden beim Family Record alle gefundenen Subtructures in der Eigenschaft Substructures und äquivalent dazu bei allen Substructures der Family Record als Superstructure hinterlegt. Die Implementierung des Postprozessors *addSubstructure()* ist in Listing 5.11 aufgeführt.

```
// connecting the superstructures and substructures
addSubstructure: (params) => {
    let superstruct = params.superstruct;
    let substruct = params.substructs;

    // superstructFound is set, when substruct is already present in parsing
        tree
    if (!substruct.superstructFound) {
        // level of substructure must be the increment of level of
            superstructure
        if (parseInt(substruct.line.level) !==
            parseInt(superstruct.line.level) + 1) throw new
            GedcomLevelError(superstruct, substruct);
        // put substruct in gedcom parsing tree
        substruct.superstructFound = true;
        superstruct.substructs.push(substruct);
    }

    return superstruct;
};
```

Listing 5.11: Postprozessor *addSubstructure()*

Im Falle des beispielhaften Family Records aus der Einleitung würden 4 Substructures gefunden werden - der Parameter *substructs* der Funktion *addSubstructure()* wäre also ein Array der Länge 4. Die *MARR*-Structure wiederum hätte ebenfalls *eine* Substructure.

5.2 Grammatik Generator

Der in Abschnitt 4.2 vorgestellte Grammatik Generator wird über die Klasse GrammarGenerator, wie in Abbildung 5.1 dargestellt, implementiert. Über die Klassenmethode *build(path)* kann eine Instanz von GrammarGenerator erstellt werden, der die Gedcom Grammatik an dem mit dem Parameter *path* spezifizierten Pfad erzeugt. Bei der Instanzerzeugung wird im ersten Schritt der Nearley-Header, der in der Nearley Datei *NearleyHeader.ne* spezifiziert ist, eingelesen und als Instanzvariable in Form einer Zeichenkette gespeichert. Dieser Nearley-Header stellt den obersten Eintrag jeder Nearley-Datei dar, die vom GrammarGenerator erzeugt wird und enthält die Include-Statements für Datentypen, Postprozessoren, etc. und den Aufruf des Moo-Lexers. Anschließend wird die Gedcom Grammatik Definition, die in Form von JavaScript Objekten gespeichert ist, gelesen und gespeichert. Diese Gedcom Grammatik Definition kann dann mit der Funktion *generateGrammar()* in Nearley-Dateien überführt werden, die dann zu Nearley Parsern kompiliert werden können. In den folgenden Kapitel wird auf diese Schritte im Detail eingegangen.

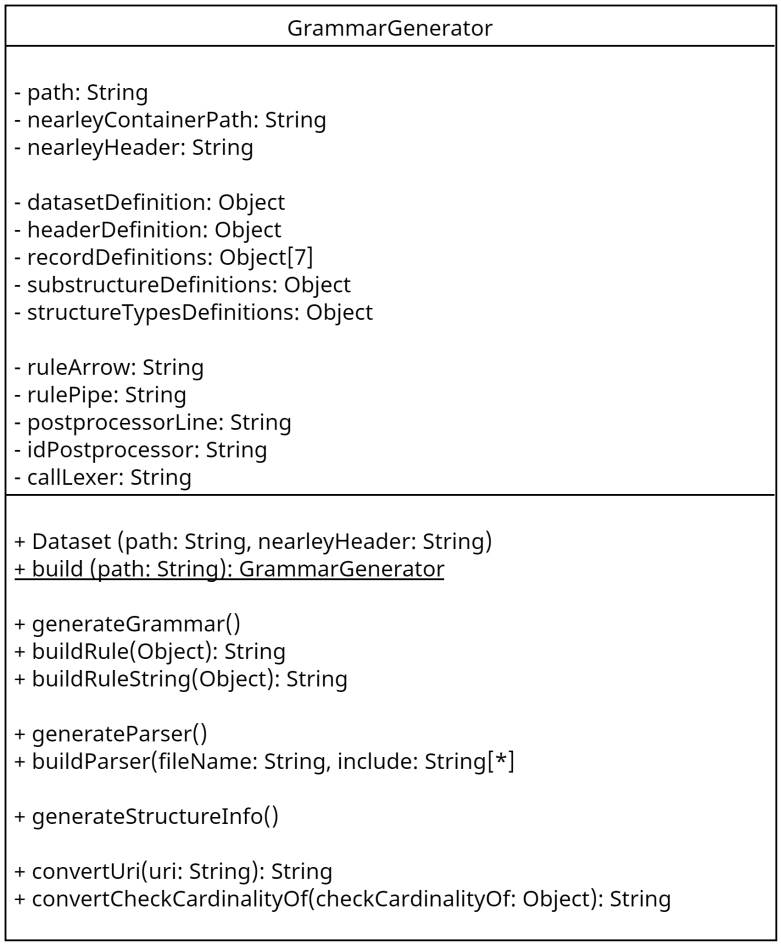

Abbildung 5.1: UML Klassendiagramm GrammarGenerator

5.2.1 Definition der Grammatik

Die Definition der Gedcom Grammatik erfolgt in Form von JavaScript Objekten. Für jede Structure, die in der Gedcom7 Spezifikation definiert ist, wird eine Grammatik Definition erstellt. Folgende Parameter sind in diesen Objekten hinterlegt.

URI	Die in der Gedcom7 Spezifikation für diese Structure hinterlegte URI.
lineType	Der Typ der Line einer Structure (hier wird beispielsweise hinterlegt, ob die Structure einen Cross-Reference-Identifier enthält oder auf andere Structuren verweisen darf).
Info	In der Info wird ein Informationstext zu jeder Strucutre hinterlegt. Dieser kann verwendet werden um bei Verwendung der Bibliothek dem Benutzer Informationen über die Bedeutung der Structures zukommen zu lassen.
Level	Die Levels unter denen die Structure in einer Gedcom7 Datei auftauchen kann.
Tag	Der in der Gedcom7 Spezifikation definierte Tag der Structure.
Substructs	Alle Structures, die als Substructure für eine Structure auftauchen können, inkl. der Kardinalität dieser.

Das Definitionsobjekt für einen Family Record ist in Listing 5.12 dargestellt. Anders als bei den Beispielen aus Abschnitt 5.1 bei denen nur ein Teil der Substructures betrachtet wurde, handelt es sich hierbei um eine vollständige Definition.

```
{
    uri: 'g7:record-FAM',
    lineType: lineTypes.FAM_RECORD,
    info: 'The Family record is a compilation of facts or hypothesized facts
        about a family.',
    level: [0],
    tag: 'FAM',
    substructs: {
        'g7:RESN': '0:1',
        FAMILY_ATTRIBUTE_STRUCTURE: '0:M',
        FAMILY_EVENT_STRUCTURE: '0:M',
```

```
            NON_EVENT_STRUCTURE: '0:M',
            'g7:FAM-HUSB': '0:1',
            'g7:FAM-WIFE': '0:1',
            'g7:CHIL': '0:M',
            ASSOCIATION_STRUCTURE: '0:M',
            'g7:SUBM': '0:M',
            LDS_SPOUSE_SEALING: '0:M',
            IDENTIFIER_STRUCTURE: '0:M',
            NOTE_STRUCTURE: '0:M',
            SOURCE_CITATION: '0:M',
            MULTIMEDIA_LINK: '0:M',
            CHANGE_DATE: '0:1',
            CREATION_DATE: '0:1'
        }
    }
```

Listing 5.12: Grammatik Definition eines Family Records

5.2.2 Grammatikgenerierung mit generateGrammar()

Nach dem in Abschnitt 5.2.1 beschriebenen Vorgehen werden Grammatik Definitionen für alle Structuretypes, Substructures und Records, sowie für das gesamte Dataset erstellt. Mit der Funktion *generateGrammar()* werden diese eingelesen, in eine nearley-konforme Zeichenkette konvertiert und anschließend in Form einer Nearley Datei (*.ne*) gespeichert. Die Regeln werden mit der Funktion *buildRuleString()* erzeugt und mit den in der Klasse GrammarGenerator definierten Building-Konstanten zusammengefügt. Ein Beispiel für eine solche Konstante ist der *ruleArrow* der zur Definition einer Regel verwendet wird und als Zeichenkette

”\n\t ->“

kodiert ist. Die so erzeugten Nearley Dateien sind einfach lesbar und liegen in der in Abschnitt 5.1.1 Form vor. Alle so erstellten Nearley Dateien werden im mit *path* spezifizierten Pfad im Verzeichnis ”*path/nearley/*“ abgelegt.

5.2.3 Parsergenerierung mit generateParser()

Mit der Funktion *generateParser()* werden die Nearley Grammatiken für alle Records und das gesamte Dataset zu Nearley Parsern kompiliert. Dazu stellt Nearley die Funktion

nearleyc *inputPath* -o *outputPath*

bereit, mit eine Nearley Datei eingelesen und im spezifizierten Pfad kompiliert werden kann. Das Erstellen von Parsern für die Records ist notwendig, da die Nearley Parser für die Syntaxüberprüfung nach Änderung eines Records verwendet werden. Würde nur ein allgemeiner Dataset-Parser erstellt werden, müsste nach jeder Änderung das komplette Dataset überprüft werden, obwohl nur ein Record verändert wurde.

Im ersten Schritt der Funktion *generateParser()* werden die Include-Statements vorbereitet, die z.B. die Definition der Datentypen enthalten. Anschließend wird für die Record- und Dataset-Grammatiken die Funktion *buildParser()* aufgerufen, die in Listing 5.13 dargestellt ist. Hier werden die mit *generateGrammar()* erzeugte Grammatik, die Include-Statements und der Nearley Header in einer Container Datei *NearleyContainer.ne* zusammengefügt. Diese Container Datei wird anschließend zu dem entsprechenden Parser kompiliert und im mit *path* spezifizierten Pfad im Verzeichnis "*path/parser/*" abgelegt.

```
// build nearley-file with include statements and NearleyHeader
async buildParser (fileName, include) {
    // string representation of the grammar to be compiled
    let fileStr = '';
    // add given include statements
    for (const file of include) {
        fileStr += `@include "../grammar/nearley/${file}"\n`;
    }
    // add nearley header
    fileStr += this.nearleyHeader;

    // overwrite content of NearleyContainer file
    await fs.writeFile(this.nearleyContainerPath, fileStr);
    // read grammar of given file
    const grammar = await fs.readFile(`${this.path}nearley/${fileName}.ne`,
        { encoding: 'utf8' });
    // append grammar to NearleyContainer file
    fs.appendFile(this.nearleyContainerPath, grammar);
    // compile composed NearleyContainer.ne file to nearley parser
    await exec(`npx nearleyc ${this.nearleyContainerPath} -o
        ${this.path}parser/${fileName}Parser.js`);
}
```

Listing 5.13: Funktion buildParser() des Grammatik Generators

5.3 Gedcom Struktur

Die Struktur einer Gedcom7 Datei wird in der Bibliothek *gedcom7.js* mit Hilfe der Klasse DATASET abgebildet, die alle Gedcom Structures verwaltet, die in Form der gleichnamigen Klasse STRUCTURE vorliegen. Structures werden in *gedcom7.js* entweder als allgemeine Instanz der Klasse STRUCTURE (z.B. eine HUSB-Structure), als RECORD (z.B. ein Family Record) oder als DATATYPE STRUCTURE (z.B. eine DATE-Structure) repräsentiert.

5.3.1 Klasse Structure

Die Klasse STRUCTURE ist die Überklasse aller Klassen zur Structure-Verwaltung, d.h. Records und Datatyp Structures erben alle Methoden und Eigenschaften von STRUCTURE. Diese Methoden und Eigenschaften sind in Abbildung 5.2 abgebildet. Eine Instanz der Klasse STRUCTURE hält alle Informationen über eine Line bereit (URI, Level, Tag, Xref, lineValue, Typ des LineValues, EOL-Zeichen). Es werden Referenzen zu allen Substructures, der Superstructure, dem Record mit dem die Structure assoziiert wird, sowie zum Dataset in dem die Structure enthalten ist bereitgestellt. Außerdem übernimmt die Klasse STRUCTURE vier zentrale Aufgaben zur Verwaltung von Gedcom7 Informationen, die von der Klasse DATASET angestoßen werden können.

1. Finden von Substructures

Eine der wichtigsten Aufgaben der Klasse STRUCTURE ist es, eigene Substructures zu suchen und zu finden. Dazu werden die Methoden *getSubstructuresByUri()*, *getSubstructuresByTag()* und *getSubstructuresByLineVal()* bereitgestellt, die alle Substructures zurückgeben, die einem Suchkriterium genügen, das abhängig von der Methode eine Gedcom7 URI, ein Gedcom7 Tag oder einen LineValue darstellen. Über den Parameter *recursive* kann spezifiziert werden, ob nur direkte Substructures (also mit einem um 1 inkrementierten Level) gesucht oder ebenfalls alle Substructures von Substructures rekursiv durchsucht werden sollen. Sollen einfach alle Substructures einer Structure ohne Suchkriterium zurückgegeben werden, kann die Methode *getSubstructures()* verwendet werden.

In vielen Anwendungsfällen kann es zudem von Interesse sein, welche Structures als Substructure in Frage kommen (also welche Structures in der Gedcom7 Spezifikation als potentielle Substructures definiert sind). Ein Beispiel hierfür wäre ein Benutzer, der einen Family Record verwaltet und herausfinden möchte, welche weiteren Informationen angegeben werden können. Für diesen Fall wird die Methode *getPossibleSubstructs()* bereitgestellt, die die Gedcom7 URIs aller Structures zurückgibt, die als Substructure auftreten können. Über den boolschen Parameter *checkCardinalityFlag* kann spezifiziert werden, ob die Kardinalität überprüft werden soll, d.h. ob nur diejenigen URIs bereitgestellt werden sollen, die beim Hinzufügen nicht zu einem CardinalityError führen.

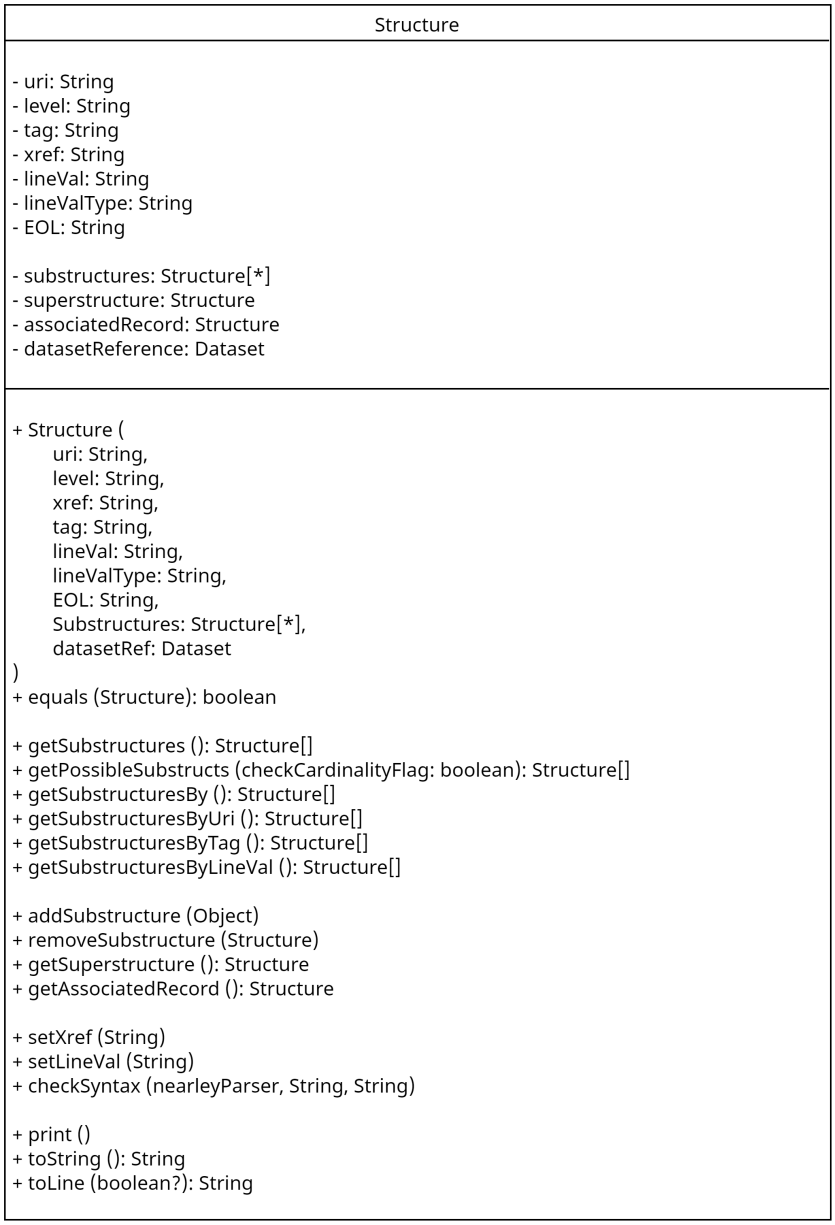

Abbildung 5.2: UML Klassendiagramm Structure

2. Hinzufügen und Entfernen von Substructures

Die Klasse Structure stellt die Methode *addSubstructure()* zur Verfügung, um einer Instanz der Klasse eine Substructure hinzuzufügen. Der Ablauf dieser Operation ist in einem Aktivitätsdiagramm in Abbildung 5.3 dargestellt. Alle benötigten Informationen über die Substructure werden als Parameter *StructureParameter* übergeben. In der Methode werden diese Informationen extrahiert und auf Basis dessen eine neue Instanz der Klasse Structure erstellt. Dieses Objekt wird in das Dataset eingefügt indem alle nötigen Referenzen angepasst werden und das Objekt Teil so der Gedcom Struktur wird. Anschließend wird die Syntax des Re-

cords überprüft, in den die neue Structure eingefügt wurde, um zu überprüfen, ob immer noch ein Gedcom7 konformes Dataset vorliegt. Ist dies der Fall wird im Dataset überprüft, ob undefinierte Cross-Reference-Identifier vorliegen.

Falls eine dieser Überprüfungen fehlschlägt, wird die Substructure über die Methode *removeSubstructure()* aus dem Dataset entfernt, indem die gleichnamige Methode der Klasse DATASET aufgerufen wird (siehe Abschnitt 5.3.5).

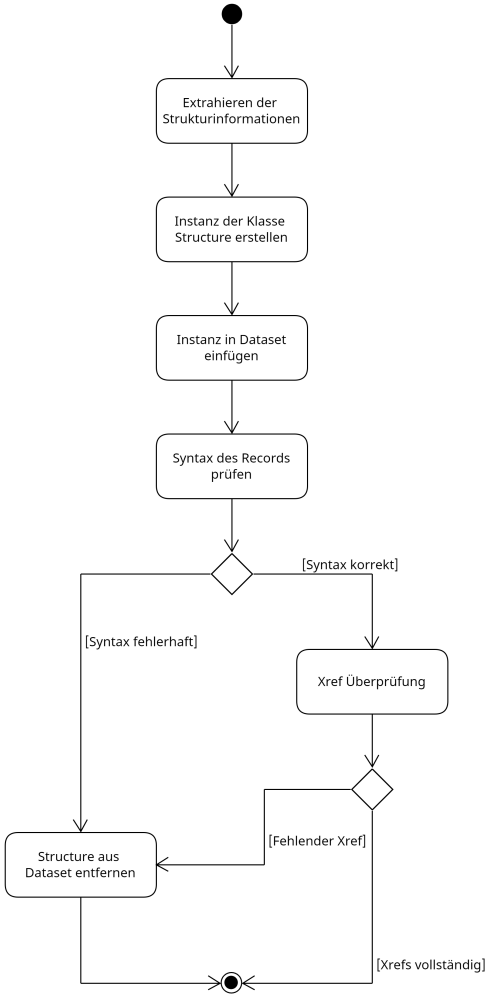

Abbildung 5.3: UML Aktivitätsdiagramm der Methode *addSubstructure()*

3. Ändern des LineValues

Der LineValue einer Instanz der Klasse STRUCTURE kann über die Methode *setLineVal()* verändert werden (siehe Abbildung 5.4). Die Eigenschaft *lineVal* des STRUCTURE Objekts wird auf den als Parameter übergebenen Wert gesetzt und anschließend wird die Syntax des entsprechenden Records überprüft, um zu kontrollieren, dass der neue LineValue syntaktisch korrekt ist. Handelt es sich um eine Structure, die einen LineValue des Typs *Xref*[1] besitzt, also auf eine andere

[1] Ein Beispiel hierfür wäre die HUSB-Structure, die auf einen Individual Record verweist.

Structure verweist, muss ebenfalls *Xref-Map* des Datasets aktualisiert werden. In dieser Map werden alle Verweise über Cross-Reference-Identifier, die im Dataset vorkommen, verwaltet. Werden nach der Veränderung des LineValues ein Syntaxfehler oder undefinierte Cross-Reference-Identifier gefunden, muss der LineValue auf den Ursprungswert zurückgesetzt werden.

Da auch ein leerer LineValue syntaktisch korrekt sein kann, wird im letzten Schritt das Dataset nach leeren Structures durchsucht, die dann entfernt werden. Ein Beispiel hierfür wäre die MARR-Structure aus dem Family Record in Listing 2.2. Würde hier der DATE-Structure ein leerer Wert zugewiesen werden, würde dies in der Structure

1 MARR

2 DATE

resultieren. Diese Structure enthält keinerlei Informationen und kann daher entfernt werden.

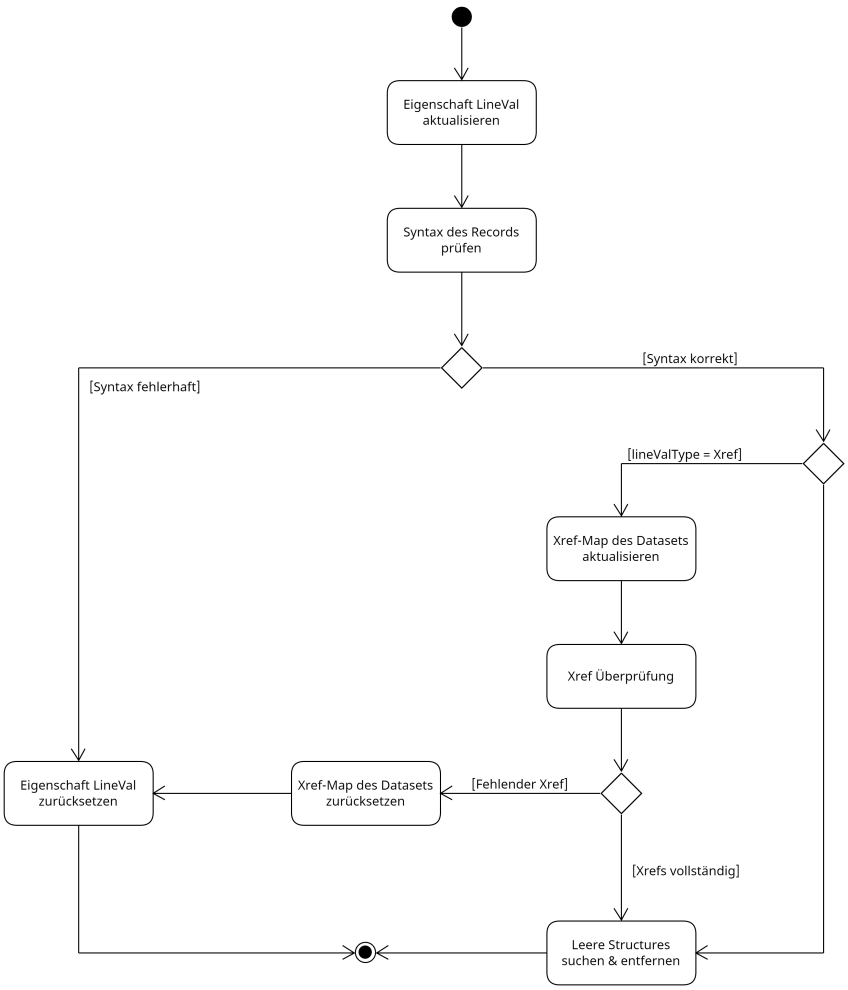

Abbildung 5.4: UML Aktivitätsdiagramm der Methode *setLineVal()*

5.3.2 Klasse Record

Die Klasse RECORD ist die Vaterklasse für Family, Individual, Header, Multimedia, Repository, SharedNote, Source und Submitter und definiert eine Methode zur Syntaxüberprüfung eines Records. Dazu wird der entsprechende Nearley-Parser, der mit dem in Abschnitt 5.2 beschriebenen GRAMMARGENERATOR erstellt wurde, eingebunden. Dieser bekommt alle Lines des Records kodiert als Zeichenkette als Eingabe und überprüft, ob alle Structures syntaktisch korrekt sind.

Des Weiteren stellt die Klasse RECORD Methoden zur Verfügung, um Informationen aus Structures, die von mehreren Records geteilt werden, zu extrahieren und in gebündelter Form auszugeben. Ein Beispiel hierfür sind die Identifier-Structures, die genutzt werden, um Structures oder ihre Inhalte eindeutig zu identifizieren. Die Klasse RECORD stellt dafür die Methode *extractIdentifierStructures()* zur Verfügung, mit der alle Identifier-Structures eines Records gesucht und zurückgegeben werden. Wie in Listing 5.15 dargestellt, wird die Methode *getSubstructuresByUri()* (siehe Abschnitt 5.3.1) verwendet, um alle Identifier-Structures des Records zu finden. Anschließend werden alle Informationen extrahiert und im JSON-Format zurückgegeben.

```
// extracts content of IDENTIFIER_STRUCTURE
//  -> Each value provides an identifier for a structure or its subject, and
     each is different in purpose
extractIdentifierStructures () {
    // REFN (Reference) is user-defined number or text that the submitter
        uses to identify the superstructure
    const references = this.getSubstructuresByUri('g7:REFN', false);
    // UID is a globally-unique identifier of the superstructure, to be
        preserved across edits
    const UIDs = this.getSubstructuresByUri('g7:UID', false);
    // EXID is an identifier maintained by an external authority that
        applies to the subject of the structure.
    const externalIdentifier = this.getSubstructuresByUri('g7:EXID', false);

    if (references || UIDs || externalIdentifier) {
        return {
            references: references?.map((ref) => {
                return {
                    reference: ref.lineVal || null,
                    type: ref.getSubstructuresByUri('g7:TYPE',
                        false)[0]?.lineVal || null
```

```
                };
            }) || null,
            uniqueIdentifier: UIDs?.map((uid) => uid.lineVal),
            externalIdentifier: externalIdentifier?.map((exid) => {
                return {
                    id: exid.lineVal || null,
                    type: exid.getSubstructuresByUri('g7:EXID-TYPE',
                        false)[0]?.lineVal || null
                };
            }) || null
        };
    }
    return null;
}
```

Listing 5.14: Methode *extractIdentifierStructures()* der Klasse RECORD

Alle Eigenschaften und Methoden der Klasse RECORD sind in Abbildung 5.5 dargestellt.

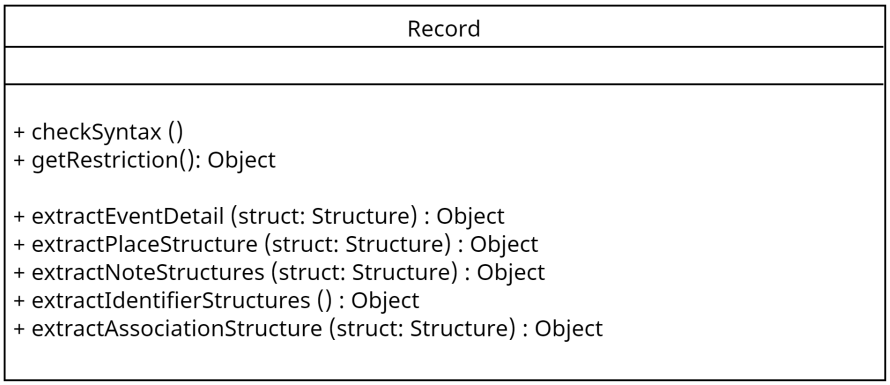

Abbildung 5.5: UML Klassendiagramm Record

5.3.3 Klasse Family

Ein Ziel bei der Erstellung der Bibliothek *gedcom7.js* war es, eine Grundlage für die Verarbeitung von Gedcom7 Dateien zu kreieren, die in zukünftigen Arbeiten erweitert werden kann. Daher wurde für diese Arbeit nur die Klasse FAMILY für den Family Record vollständig implementiert. Die Klassen für alle weiteren Records können analog zu dem hier beschrieben Vorgehen erstellt werden, um die Gedcom7 Spezifikation komplett abzubilden.

Die Klasse FAMILY ruft die Methode *checkSyntax()* der Vaterklasse RECORD mit der Family Grammatik auf. Des Weiteren werden Methoden zur einfachen Verarbeitung von Informationen aus einem Family Record bereitgestellt. Ist ein Anwender beispielsweise an Informationen über die Kinder einer Familie interessiert, müsste er die Gedcom7 Spezifikation studieren, alle Structures die Informationen über ein Kind bereithalten können nacheinander suchen und dann alle Informationen zusammenfügen. Um diese Arbeit zu erleichtern, werden in der Bibliothek *gedcom7.js* die in Abbildung 5.6 aufgeführten Convenience-Methoden bereitgestellt. Ein Beispiel hierfür ist die in Listing 5.15 abgebildete Methode *getChildrenInformation()*, mit der alle Informationen über die Kinder einer Familie extrahiert werden können. Dazu werden die entsprechenden Strukturen über die Methode *getSubstructuresByUri()* gesucht und im JSON-Format zurückgegeben. Alle Methoden und Eigenschaften der Klasse FAMILY sind in Abbildung 5.6 aufgeführt.

```
// returns information about the children of this family
getChildrenInformation () {
    const famNCHI = this.getSubstructuresByUri('g7:FAM-NCHI', false)[0];
    const famEventDetail = this.extractFamilyEventDetail(famNCHI);

    if (famNCHI) {
        return {
            numberOfChildren: Number.parseInt(famNCHI.lineVal) || null,
            type: famNCHI.getSubstructuresByUri('g7:TYPE', false)[0]?.lineVal
                || null,
            parentInformation: famEventDetail?.parentInformation || null,
            eventDetails: famEventDetail?.eventDetails || null
        };
    }
    return null;
}
```

Listing 5.15: Methode *extractIdentifierStructures* der Klasse RECORD

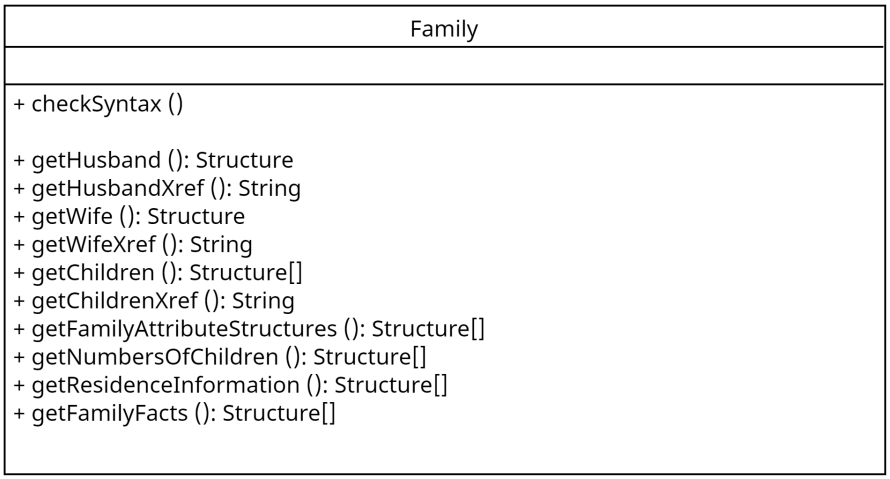

Abbildung 5.6: UML Klassendiagramm Family

5.3.4 Datatype Structures: GedcomDate Klasse

Um besser mit den Datentypen der Gedcom7-Spezifikation innerhalb der Bibliothek umgehen zu können, wurden Datentyp-Strukturen eingeführt und die Klasse GedcomDate exemplarisch implementiert.

Diese soll als Vorlage für weitere Gedcom7-Datentypen, oder erweiterte Datentypen, genutzt werden. Nach dem erfolgreichen Parsen, werden die Klassen für die *Records* und *Structures*, sowie auch die Datentyp-Strukturen erstellt.

Wenn vom Nearley-Parser die Datentypen *DateValue*, *DatePeriod* oder *DateExact* erkannt werden, wird die Klasse GedcomDate mit den entsprechenden Line-Parametern instanziiert. Beim Initialisieren der GedcomDate-Klasse wird das spezifikationskonforme Datum, mit Hilfe der Datums-Bibliothek *date-fns*, in ein JavaScript-Date-Objekt umgewandelt. Je nach Date-Typ wird das Datum nur in das Attribut *startDate* oder auch und in das Attribut *endDate* der Klasse GedcomDate, gespeichert.

Die Klasse GedcomDate bietet eine Methode *getDateObject()*, in der das GEDCOM Datum als JavaScript-Date-Objekt mit Metadaten zum Date-Typ zurückgegeben wird. Diese Methode ist in Listing 5.16 abgebildet. Es wird zunächst kontrolliert, ob eine Datumsspanne oder ein einzelnes Datum vorliegt. Anschließend wird das result-Objekt konditional mit den vorhandenen Properties gefüllt und zurückgegeben. Das *start/endDate* beeinhaltet ebenfalls die Uhrzeit, welche als Substructure der Date-Structure angegeben werden kann.

```
// public method to get the date as an object
// with a descriptor (in case of a special date)
getDateObject() {
    // determine if the date is a single date or a date range
    const isDateRange = this.startDate !== null && this.endDate !== null;
```

```
        // create result object with conditionally added properties
        const result = {
            type: this.lineValType,
            ...(isDateRange && { startDate: this.startDate, endDate:
                this.endDate }),
            ...(!isDateRange && { date: this.startDate }),
            ...(this.descriptor && { descriptor: this.descriptor }),
            ...(this.description && { description: this.description }),
            ...(this.calendar && { calendar: this.calendar }),
            ...(this.epoch && { epoch: this.epoch })
        };
        return result;
    }
```

Listing 5.16: Funktion getDateObject() der Klasse GedcomDate

Beispiel

Der folgende Ausschnitt einer Gedcom7 Datei zeigt ein erweitertes Datum, welches nach erfolgreichem Parsen in der Klasse GedcomDate repräsentiert wird.

```
0 @F1@ FAM
1 HUSB @I1@
1 MARR
2 DATE AFT JULIAN 13 MAR 1998 BCE
```

Wird die Methode *getDateObject()* der Klasse GedcomDate für dieses Datum aufgerufen, wird das Datum in folgendes JavaScript-Objekt umgewandelt:

```
{
    type: "DateValue",
    date: 1998-03-13T00:00:00,
    descriptor: "AFT",
    description: "After date: Exact date unknown, but no earlier than x",
    calendar: "JULIAN",
    epoch: "BCE"
}
```

Die Datatyp-Strukturen stellen somit eine Schnittstelle zwischen den Gedcom7 Datentypen und JavaScript dar und ermöglichen somit direkten Zugriff auf die Daten, ohne zum Beispiel den in der Gedcom Spezfikation definierte Datentypen kennen zu müssen. Andere Datentypen oder durch *Custom Tags* erweiterte Datentypen können auf die gleiche Weise implementiert werden.

5.3.5 Klasse Dataset

Die Hauptaufgabe der Klasse DATASET besteht darin, Structures zu erstellen und zu verwalten. Dazu werden die in Abbildung 5.7 dargestellten Eigenschaften und Methoden verwendet. In den Eigenschaften jeder Instanz der Klasse DATASET werden Header und Trailer, alle Records die enthalten sind, Datenstrukturen zur Verwaltung von Cross-Reference-Identifiern, sowie Informationen über die Verwendung des Byte-Order-Mark und End-Of-Line Characters gespeichert. Bei der Erstellung einer Instanz der Klasse DATASET über den Konstruktor können Header-Trailer- und Record Informationen übergeben werden, aus denen Instanzen der Klasse STRUCTURE, bzw. RECORD generiert werden. Zudem kann ein leeres Dataset über die Methode *createEmptyDataset()* erstellt werden. Die Hauptaufgaben der Klasse DATASET können zu den folgenden vier Punkten zusammengefasst werden.

1. Erstellen von Structures

Mit Hilfe der Methode *createStructure()* können Structures auf Basis von Informationen, die vom Nearley Parser zurückgegeben wurden, erstellt werden. Zusätzlich werden Informationen zur Superstructure und dem Record zu dem die Structure gehört, benötigt. Auf Basis dieser Informationen kann die passende Structure erstellt werden, Referenzen andere Structures angepasst und bei Bedarf Einträge in die Xref-Map erstellt werden.

2. Hinzufügen/Entfernen von Records

Die Methoden *addRecord()* kann verwendet werden um neue Records auf Basis von übergebenen Structure Informationen zu erstellen und ins Dataset einzugliedern. Benötigt werden dazu die Gedcom7 URI, der LineValue (sofern dieser vorhanden ist) und die Substructures, die enthalten sein sollen. Anschließend werden die Referenzen im Dataset so angepasst, dass der Record an der richtigen Stelle eingegliedert wird. Mit *removeRecord()* kann ein Record mit der gegebenen Referenz aus dem Dataset entfernt werden. Dabei wird der Eintrag aus der Xref-Map entfernt. Außerdem wird überprüft, ob eine Structure im Dataset über einen Cross-Reference-Identifier auf den entfernten Record verweist. Wenn dies der Fall ist, muss die entsprechende Referenz zu einem @VOID@-Pointer geändert werden, da nicht auf nicht-definierte Xrefs verwiesen werden darf [Fam22]. Zusätzlich wird eine Warnung für den Benutzer ausgegeben, um darauf hinzuweisen, dass das Entfernen eines Records dazu führt, dass ebenfalls alle Substructures entfernt werden.

3. Hinzufügen/Entfernen von Structures

Mit den gleichen Informationen wie beim Hinzufügen von Records, können auch Structures zum Dataset hinzugefügt werden. Zusätzlich werden die Superstructure und der Record benötigt, in den die Structure eingefügt werden soll, um ein richtiges Eingliedern ins Dataset zu ermöglichen.

Die Methode *removeStructure()* kann verwendet werden, um Structures aus dem Dataset zu entfernen. Hierbei sind im Gegensatz zum Entfernen von Records weitere Überprüfungen notwendig. Wenn das Entfernen einer Structure dazu führt, dass die Superstructure weder einen LineValue, noch Substructures hat, sollte die Superstructure ebenfalls entfernt werden [Fam22]. Außerdem kann das Entfernen einer Structure dazu führen, dass die Syntax der Superstructure nicht mehr korrekt ist, z.B. weil die Structure in der Gedcom7 Spezifikation als erforderlich (Kardinalität 1:1 oder 1:M) definiert wurde. In diesem Fall wird die Structure beibehalten und ein dem Datentyp entsprechender leerer Wert wird als LineValue eingetragen [Fam22]. Auch hier wird eine Warnung für den Benutzer ausgegeben, um darauf hinzuweisen, dass das Entfernen eines Records dazu führt, dass ebenfalls alle Substructures entfernt werden.

4. Suchen von Record

Die Klasse DATASET implementiert verschieden Methoden um Records im Dataset zu suchen. Beispielsweise können über die Methode *getFamilyRecords()* alle Family Records des Datasets zurückgegeben werden. Sucht man einen bestimmten Record, kann dieser über *getRecordByXref()* über den Cross-Reference-Identifier gefunden werden.

5. Ausgabe als Gedcom7 konformer String

Eine wichtige Anforderung für die Bibliothek *gedcom7.js* war es, dass die eingelesenen Gedcom7 Dateien wiederausgegeben werden können. Dazu implementiert die Klasse DATASET die Methode *toString()*. In dieser Methode werden für alle Records, Header und Trailer die *toString()*-Methode aufgerufen, die in der Klasse Structure so definiert ist, dass alle Structure-Informationen in Form einer Gedcom7-konforme Line zurückgegeben werden. Diese Lines werden in der richtigen Reihenfolge zusammengehangen und ergeben so eine Zeichenkette, die mit Hilfe der asynchronen Methode *write()* am spezifizierten Pfad in Form einer Gedcom7 Datei gespeichert werden kann.

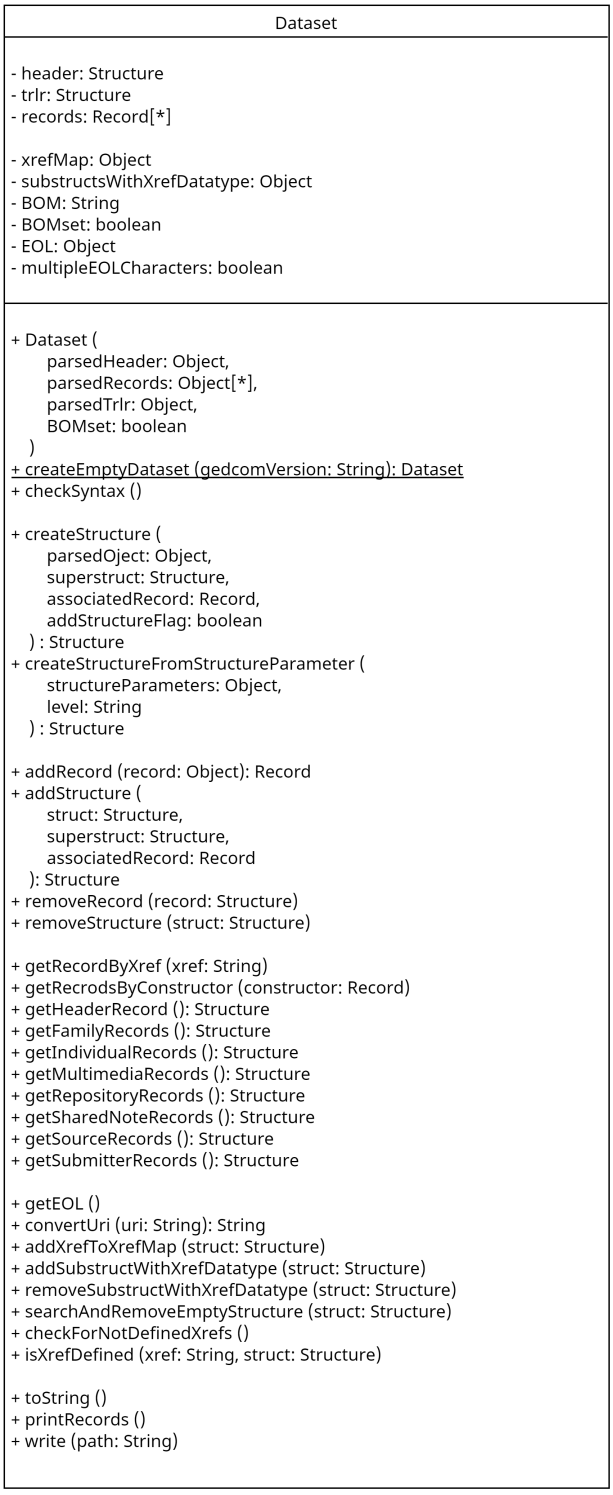

Abbildung 5.7: UML Klassendiagramm Dataset

5.4 Gedcom Parser

Alle in dieser Arbeit vorgestellten Konzepte und Implementierungen werden im GEDCOM PARSER vereinigt, der die zentrale Klasse der Bibliothek *gedcom7.js* darstellt. Die Klasse GEDCOMPARSER verfügt über die Methode *parseGedFile()* mit der Gedcom7 Dateien mit Hilfe der Node.js FileSystem Bibliothek als UTF-8 kodierte Zeichenkette eingelesen und anschließend mit der Methode *parseString()* geparsed werden kann. Wie im Konzept in Abschnitt 4.4 dargestellt, wird ein Nearley Parser erstellt, der String geparsed und anschließend aus den extrahierten Informationen eine Instanz der Klasse DATASET erzeugt die zurückgegeben wird. Das Sequenzdiagramm dieser Methode ist in Abbildung 4.3 dargestellt.

Im Folgenden wird ein beispielhafter Ablauf für die Verwendung der Bibliothek *gedcom7.js* dargestellt. Dazu wird eine Gedcom7 Datei eingelesen, die den FamilyRecord aus Listing 2.2 enthält:

Im ersten Schritt wird die Gedcom7 Datei eingelesen, geparsed und in ein Dataset überführt. Dazu wird eine Instanz der Klasse GEDCOMPARSER erzeugt.

```
gedcomParser = new GedcomParser();
```

Anschließend wird der Pfad zur Gedcom7 Datei an die Methode *parseGedFile()* übergeben. In diesem Beispiel liegt die Datei im selben Verzeichnis, wie die ausgeführte JavaScript-Datei unter dem Namen "FamilyExample.ged".

```
const dataset = await gedcomParser.parseGedFile('./FamilyExample.ged');
```

Dann kann die Family aus Listing 2.2 über den Cross-Reference-Identifier gesucht werden.

```
const famF1 = dataset.getRecordByXref('@F1@');
```

Sollen Informationen über die Kinder der Family ausgegeben werden, kann dies über

```
const nchi = famF1.getChildrenInformation();
console.log(nchi);
// Ausgabe:
  {
    numberOfChildren: 2,
    type: null,
```

```
    parentInformation: null,
    eventDetails: null
  }
```

realisiert werden. Außerdem kann das Datum der Hochzeit herausgefunden werden mit:

```
const marrInfo = famF1.getMarriageInformation();
console.log(marrInfo.marriages[0].eventDetails.date);
// Ausgabe:
  {
    type: "DateValue"
    date: 1951-03-01T00:00:00.000TZ
  }
```

Soll nun die Information zum Family Record hinzugefügt werden, dass die Ehe wieder geschieden wurde, kann dies über die DIV-Structure ausgedrückt werden. Mit Hilfe der Methode *addSubstructure()* kann eine DIV-Structure zum Family Record hinzugefügt werden:

```
const div = {
    uri: 'g7:DIV',
    substructs: [{
        uri: 'g7:DATE',
        lineVal: '28 DEC 1963',
    }]
};
famF1.addSubstructure(div);
```

Wird der Family Record nun ausgegeben, ist die neue DIV-Structure in der Ausgabe mit korrekter Gedcom7 Line-Syntax enthalten:

```
console.log(famF1.toString());
// Ausgabe:
  0 @F1@ FAM
  1 HUSB @I1@
  1 WIFE @I2@
  1 MARR
```

```
2 DATE 1 MAR 1951
1 NCHI 2
1 DIV
2 DATE 28 DEC 1963
```

5.5 Test der Implementierung

Die Implementierung der Bibliothek *gedcom7.js* wurde im Entwicklungsprozess ausgiebig getestet. Mit Hilfe des in Kapitel 2.4 vorgestellten JavaScript Test-Frameworks *Mocha* wurden insgesamt 176 Test spezifiziert, die die wichtigsten Use-Cases der Bibliothek abdecken. Im Besonderen wurden die Klassen Gedcom Parser, Dataset und Structure getestet. Da der Gedcom Parser intern die vom Grammar Generator erstellte Nearley Grammatik verwendet, wird der Grammar Generator beim Test des Gedcom Parser indirekt mitgetestet. In zukünftigen Arbeiten sollten jedoch zusätzliche Tests für den Grammatik Generator implementiert werden, da eine korrekte Syntaxüberprüfung die Grundlage für eine fehlerfreie Nutzung der Bibliothek darstellt. Besonders wenn eine Schnittstelle zum Grammar Generator zur Erstellung von Extensions eingeführt wird, sind ausgiebige Testfälle unabdingbar. Außerdem sollten in zukünftigen Arbeiten Tests für die *Datatype Structures* und alle Records implementiert werden.

Im Folgenden wird anhand eines exemplarischen Tests gezeigt, wie die Komponenten der Bibliothek *gedcom7.js*getestet wurden. Zusätzlich zu *Mocha* wurde dazu die *Node.js* Assertion-Bibliothek *Chai* verwendet, mit der ausdrucksstarke und leicht verständliche Testfälle in englischer Sprache definiert werden können. Als Testgrundlage wurden die sieben Gedcom7 Dateien verwendet, die von *Family Search* als Beispieldateien bereitgestellt wurden. Diese Dateien decken die komplette Gedcom7 Spezifikation ab und versuchen alle Structuretypes, Datatypes und Sonderfälle abzudecken. Funktioniert das Parsen und Verarbeiten dieser Dateien korrekt, kann davon ausgegangen werden, dass die Gedcom7 Spezifikation korrekt in der Bibliothek abgebildet wurde.

Mit den in Listing 5.17 aufgeführten Testfällen, wird das Lesen und Schreiben von Gedcom7 Dateien getestet. Dazu werden die oben angesprochenen, von *Family Search* als Beispieldateien bereitgestellten, Gedcom7 Dateien mit einer Instanz des Gedcom Parser eingelesen und als Dataset zurückgegeben. Dieses Dataset wird über die Methode *write()* temporär gespeichert und anschließend mit der ursprünglichen Gedcom7 Datei verglichen. Dieser Vergleich kann auf einfache Weise mit der Textvergleichsimplementierung *diff*[2] umgesetzt werden, mit der die Inhalte der Dateien Zeichen für Zeichen verglichen werden. Auf diese Weise wird ebenfalls sichergestellt, dass Gedcom7 Dateien die eingelesen aber unverändert bleiben, nach der Ausgabe identisch bleiben. Dies ist eine wichtige Anforderung,

[2] Nähere Informationen sind unter https://www.npmjs.com/package/diff angegeben.

da textbasierte Dateiformate wie der Gedcom Standard gerne über Versionsverwaltung administriert werden. Für diesen Use-Case wäre es nicht annehmbar, wenn sich Änderungen in Dateien ergeben, die nicht angepasst wurden. Damit eine identische Ausgabe möglich ist, müssen im DATASET Angaben über die verwendeten EOL-Character und das Byte-Order-Mark gespeichert werden.

```
describe('test if gedcom file is equivalent before and after parsing', () =>
    {
    const gedcomParser = new GedcomParser();
    const path = 'test/sampleData/ExampleFamilySearchGEDCOMFiles/';

    // Family Search Example Files
    const gedFiles = [
        'escapes.ged',
        'long-url.ged',
        'maximal70_without_extensions.ged',
        'minimal70.ged',
        'remarriage1.ged',
        'same-sex-marriage.ged',
        'voidptr.ged'
    ];

    forEach(gedFiles)
    .it('#%s', async (fileName) => {
        // read Gedcom file as String
        const beforeParsing = await readGedFile(path + fileName);
        // parse Gedcom String and write it to temp.ged
        const dataset = gedcomParser.parseString(beforeParsing);
        await fs.writeFile(path + 'temp.ged', '');
        await dataset.write(path + 'temp.ged');
        // read temp.ged as String and compare it with original file
        const afterParsing = await readGedFile(path + 'temp.ged');
        // expect files to be equal
        expect(diffChars(beforeParsing, afterParsing)).to.have.lengthOf(1);
    });
});
```

Listing 5.17: Testfälle für das Lesen und Schreiben von korrekten Gedcom7 Dateien

In der folgenden Auflistung sind alle weiteren Testfälle aufgeführt, die im Rahmen dieser Arbeit implementiert wurden. Alle Testfälle sind im Repository der Bibliothek im Ordner *Test* abgelegt.

Dataset	• Vergleich von geschriebenen Gedcom7 Dateien mit dem Original • Suchen von Records im Dataset • Fehler finden bei Eingabe von Gedcom7 Dateien mit nicht definiertem Xref • Fehler finden bei Eingabe von Gedcom7 Dateien mit mehrfach verwendetem Xref • Warnungsausgabe bei fehlendem Byte-Order-Mark • Verwendung der richtigen EOL-Character
Parser	• Parsen von korrekten Gedcom7 Dateien ohne Fehler • Syntaxfehler finden beim Parsen von Gedcom7 Dateien mit fehlerhafter Syntax • Syntaxfehler finden beim Parsen von Gedcom7 Dateien mit fehlendem Header und/oder Trailer
Structure	• Finden von Substructures anhand des Tags • Finden von Substructures anhand des LineValues • Ändern des LineValues • Ändern des Xref • Hinzufügen von einer/mehrere Substructures mit keiner/einer/mehreren Substructures

Tabelle 5.1: Bestandteile einer GEDCOM Line und ihre Bedeutung

6

Zusammenfassung und Ausblick

In dieser Arbeit wurde die JavaScript-Bibliothek *gedcom7.js* für das genealogische Austauschformat FamilySearch GEDCOM Version 7 entwickelt und getestet. Dabei wurden dem Leser alle für das Verständnis benötigten theoretischen Grundlagen an die Hand gegeben und anschließend dem Entwicklungsprozess entsprechend zuerst die Konzeption und danach die konkrete Implementierung präsentiert.

Die Bibliothek setzt sich aus vier Hauptkomponenten zusammen. Das zentrale Element ist der GEDCOM PARSER, mit dem Dateien im Format Gedcom7 eingelesen werden und in ein DATASET überführt werden können. Dabei wurde ein besonderer Fokus auf die spezifikationskonforme Handhabung von Gedcom7-Dateien gelegt. Die Syntaxüberprüfung wurde mit Hilfe der JavaScript-Bibliothek *Nearley* umgesetzt, die auf Grund der vielen nützlichen Features wie z.B. die Möglichkeit Postprozessoren für Regeln anzugeben, eine perfekte Wahl für diese Aufgabe darstellt. Die für die Überprüfung zugrundeliegende Grammatik wird mit Hilfe eines *Grammatik Generators* generiert, der die Gedcom7 Spezifikation in eine Nearley-konforme Syntax überführt. Dies bietet neben der Arbeitserleichterung den großen Vorteil, dass so eine unkomplizierte Erweiterbarkeit der Bibliothek garantiert wird und in zukünftigen Arbeiten Features wie Gedcom Extensions auf einfache Weise implementiert werden können.

Die Datenstrukturen DATASET und STRUCTURE stellen die grundlegenden Datenstrukturen für die Verwaltung von eingelesene Gedcom7 Dateien in der Bibliothek *gedcom7.js* dar. Die Klassen implementieren die wichtigsten Methoden zur Administration von Gedcom7 Dateien und können in zukünftigen Arbeiten einfach erweitert werden, da alle grundlegenden Strukturen bereits implementiert sind. Außerdem wurden Klassen für Records und spezielle Datentypen bereitgestellt. Anhand der Klassen GEDCOM DATE und FAMILY wurde gezeigt, wie tiefergehende Funktionalitäten und Convinience-Funktionen zur einfachen Handhabung für den Benutzer der Bibliothek implementiert werden können. Die Datenstrukturen DATASET und STRUCTURE stellen die grundlegenden Datenstrukturen für die Verwaltung von eingelesene Gedcom7 Dateien in der Bibliothek *gedcom7.js* dar. Die Klassen implementieren die grundlegenden Methoden zur Administration von Gedcom7 Dateien und können in zukünftigen Arbeiten einfach erweitert werden, da alle grundlegenden Strukturen bereits implementiert sind. Außerdem wurden

Klassen für Records und spezielle Datentypen bereitgestellt. Anhand der Klassen GEDCOM DATE und FAMILY wurde gezeigt, wie tiefergehende Funktionalitäten und Convenience-Funktionen zur einfachen Handhabung für den Benutzer der Bibliothek implementiert werden können.

Vergleicht man die Bibliothek *gedcom7.js* mit den verwandten Arbeiten, die in Kapitel 3 vorgestellt wurden, wird deutlich, dass *gedcom7.js* wesentlich ausführlicher und umfangreicher ist. Besonders mit der vollständigen Syntaxüberprüfung und die einfache Erweiterbarkeit hebt die Implementierung von anderen Arbeiten ab. Vergleicht man die Bibliothek *gedcom7.js* mit den verwandten Arbeiten, die in Kapitel 3 vorgestellt wurden, wird deutlich, dass *gedcom7.js* in allen Belangen überlegen ist. Besonders die vollständige Syntaxüberprüfung und die einfache Erweiterbarkeit heben die Implementierung von anderen Arbeiten ab.

Abschließend lässt sich festhalten, dass die Bibliothek gedcom7.js eine robuste und spezifikationskonforme Möglichkeit bietet, Gedcom7 Dateien in JavaScript zu lesen, zu verarbeiten und zu schreiben. Die Syntaxprüfung und die automatisierte Grammatikerstellung durch den Grammar Generator sind besonders nützliche Features, die die Erstellung und Manipulation von Gedcom7-Dateien erleichtern und eine solide Grundlage für die Weiterentwicklung des Projekts darstellen.

Literaturverzeichnis

Ahn. Ahnenforschung: *Genealogie*. Abgerufen am 02.02.2023 von `https://www.ahnenforschung.de/themen/genealogie/`.

Chaa. Chandra, Kartik: *Documentation Glossary: nearley.js*. Abgerufen am 10.02.2023 von `https://nearley.js.org/docs/glossary`.

Chab. Chandra, Kartik: *Documentation: nearley.js*. Abgerufen am 10.02.2023 von `https://nearley.js.org/`.

Ear70. Earley, Jay: *An efficient context-free parsing algorithm*. Communications of the ACM, 1970.

Fam. FamilySearch: *About FamilySearch*. Abgerufen am 02.02.2023 von `https://www.familysearch.org/en/about/`.

Fam22. Family History Department: *The FamilySearch GEDCOM Specification 7.0.11*. The Church of Jesus Christ of Latter-day Saints, 15 East South Temple StreetSalt Lake City, UT 84150 US, 7.0.11 Auflage, November 2022.

fDA07. Demoskopie Allensbach, Institut für: *Ahnen- und Familienforschung*. Allensbacher Berichte, 13(2), 2007.

Fey16. Feynman, Richard: *Ebnf: A notation to describe syntax*. 2016.

Gen. Genealogists, Society Of: *Genealogy or Family History*. Abgerufen am 02.02.2023 von `https://www.sog.org.uk/learn/hints-tips/genealogy-or-family-history`.

Mal. Malteser: *Familienforschung: Den Stammbaum weitergeben*. Abgerufen am 25.02.2023 von `https://www.malteser.de/dabei/familie-freundschaft/familienforschung-woher-stamme-ich-eigentlich.html`.

Moc. Mocha Package Maintainer: *Mocha Documentation*. Abgerufen am 25.02.2023 von `https://mochajs.org//`.

Rad. Radvan, Tim: *Documentation: moo.js*. Abgerufen am 10.02.2023 von `https://github.com/no-context/moo`.