Tucup

Implementierung einer Turing-vollständigen Markup-Language.

Lasse Becker

Bachelorarbeit
September 2021

Programmiersprachen und Übersetzerkonstruktion
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Betreut durch
Prof. Dr. Michael Hanus

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Zusammenfassung

Subjekt dieser Arbeit ist der Entwurf der Markup-Language Tucup (Turing complete markup) und die Implementierung des zugehörigen Interpreters. Der Tucup-Interpreter übersetzt Textdokumente zu PDF-Dateien. Ebenfalls werden verschiedene Anforderungen an eine benutzerfreundliche Software zum Erzeugen von Textdokumenten vorgestellt. Inwiefern andere Software diesen Anforderungen genügt, wird für eine Auswahl von Programmen geprüft. Es sei vorweggenommen, dass Tucup nur wenige der Forderungen zu diesem Zeitpunkt einhält. Aber es wird perspektivisch erläutert, wie spätere Versionen der Sprache aussehen könnten und worauf unter Berücksichtigung der Ansprüche beim Entwurf bereits geachtet wurde. Zu guter Letzt ist dies ein Proof-of-Concept, da diese Arbeit bereits mit Tucup verfasst wurde.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **3**
2.1 Textsatz . . . 3
2.2 Turing-Vollständigkeit . . . 4
2.3 Markup-Languages . . . 4

3 Motivation und Anforderungen an Tucup **7**
3.1 Trennung von Form und Inhalt . . . 7
3.2 Konsistenz der Semantik in Raum und Zeit . . . 7
3.3 Erweiterbarkeit von Layouts . . . 8
3.4 Modulare Layouts . . . 8
3.5 Spezifikation der Strukturen . . . 8
3.6 Konfigurierbare Layouts . . . 9
3.7 Restriktive Layouts . . . 9
3.8 Statische Prüfung von Quelltext . . . 9
3.9 Dokumentation von Strukturen . . . 9
3.10 Ungestörter Schreibfluss . . . 10
3.11 Praktikables Programmieren . . . 10

4 Abgleich vorhandener Software mit den gestellten Anforderungen **13**
4.1 TEX-Familie . . . 13
4.2 Simple Markup-Languages . . . 15
4.3 HTML/CSS/JS . . . 16
4.4 Module und Packages zum PDF-Erzeugen . . . 17
4.5 WYSIWYG-Editoren . . . 19

5 Entwurf von Tucup **21**
5.1 Programmierkonzepte . . . 21
5.2 Syntax . . . 22
5.3 Code Beispiele . . . 23
5.4 Builtins . . . 26

6 Implementierung **29**
6.1 Architektur . . . 29
6.2 Speicher . . . 29
6.3 Objekte . . . 30
6.4 Parsen . . . 30
6.5 Auswerten . . . 30
6.6 Builtin-Objekte . . . 31
6.6.1 Set . . . 31
6.6.2 Funktionskonstruktor . . . 31

6.6.3 Throw und Error 31
6.7 Transpilieren und Zwischenformat Übersetzen 32

7 Fazit und Ausblick **33**

A TypeScript pdfmake Beispiel **35**

B Tucup-Code für Zufällige Additionsaufgaben **37**

C Zufällige Additionsaufgaben erstellt mit Tucup **39**

Bibliografie **41**

Kapitel 1

Einleitung

Textdokumente finden sich in fast allen Bereichen des Lebens. Dabei gewinnen digitale Formate immer mehr an Bedeutung. Etwa die automatisierte Erstellung von Rechnungen, Mahnungen, Einladungen etc. ist längst keine Besonderheit mehr und oft notwendig, um die Geschäftsprozesse in Unternehmen überhaupt handhabbar zu machen.

Die Überschneidung von digitalen zu analogen Dokumenten sind hierbei druckbare Formate. Die meisten digitalen Dokumente lassen sich einfach in ein druckbares Format bringen, doch sind die Resultate nicht immer hochwertig. Spezialisierte Programme bieten die Möglichkeit, die Maße des zu bedruckenden Mediums zu berücksichtigen, um für das Drucken zugeschnittene Dateien zu erhalten (vorweg sei hier Microsoft Office Word genannt). Hierbei ist dem Autor die Formatierung sowie das Layouting überlassen. Dabei entstehen nicht selten Inkonsistenzen oder Schriftfehler in der Typographie. Insbesondere in den Naturwissenschaften hat sich seit seiner Entstehung LaTeX durchgesetzt, welches einige Inkonsistenzen und Satzfehler verhindern kann, indem eine Trennung von Form und Inhalt angestrebt wird. Dabei ist die Verfügbarkeit von vielen Templates mitverantwortlich für den Erfolg von LaTeX. Jedoch ist der Entwurf eines eigenen Layouts sehr aufwändig und benötigt ein tiefgreifendes Verständnis des TeX Systems. Anpassungen von Layouts sind in gleichem Maße an Fachwissen gebunden. Daher ist es wenig verwunderlich, dass LaTeX für eine ausgesprochen flache Lernkurve bekannt ist und damit unattraktiv für technisch wenig versierte Nutzer.

Doch die Möglichkeiten, LaTeX zu nutzen beschränken sich nicht auf das Ausgeben von statischen Inhalten, denn viele Projekte zeigen, dass LaTeX auch genutzt werden kann, um Berechnungen durchzuführen. Dadurch ist die Möglichkeit gegeben, Dokumente dynamisch zu erzeugen. Doch mangels eines strukturierten Programmierparadigmas sind solche Berechnungen auch für erfahrene Programmierer eine Herausforderung. Durch diese Hürden ist LaTeX wenig attraktiv, auch wenn es um das automatische Erzeugen von Dokumenten geht.

Aufgrund dieser Unzulänglichkeiten von LaTeX bedarf es einer Alternative, die zusätzlichen Anforderungen genügt. Daher befasst sich diese Arbeit mit dem Implementieren einer Markup-Language, die eine steile Lernkurve sowie strukturiertes Programmieren ermöglichen soll. Diese Sprache wird Tucup genannt. Diese Arbeit beschäftigt sich mit der automatisierten Erzeugung von Textdokumenten. Daher ist händische Formatieren und Drucken von Dokumenten nicht Gegenstand dieser Arbeit und wird außer Acht gelassen.

Im folgenden Kapitel werden die grundlegenden Begriffe zum Verständnis dieser Arbeit erklärt. Die Ziele, welche bei einer Implementierung einer neuen Markup-Language verfolgt werden, sind im Kapitel 'Motivation und Anforderungen an Tucup' aufgezählt. Um die Notwendigkeit einer neuen Sprache zu rechtfertigen, werden im Kapitel 'Abgleich vorhandener Software mit den gestellten Anforderungen' bestehende Systeme an den zuvor definierten Anforderungen gemessen. Dabei wird deutlich, dass keine untersuchte Software allen Anforderungen zur Gänze genügt. Anschließend wird der Entwurf von Tucup diskutiert. Dabei werden diejenigen Maßnahmen erläutert, welche ergriffen wurden, um die definierten Anforderungen zu erfüllen. Weiterhin wird ein Einblick in die Handhabung von Tucup gewährt. Die Umsetzung des Entwurfs ist Thema des Kapitels 'Implementierung'. Die Verwendete Technologien und Programmiertechniken werden erläutert.

Kapitel 2

Grundlagen

2.1. Textsatz

Der Satz eines Textdokumentes ist dessen druckfertige Formatierung. Der Prozess des Formatierens nennt sich Setzen. Auf einige der Aspekte, die bei diesem Vorgang berücksichtigt werden sollten, wird in diesem Absatz eingegangen. Der Hauptteil eines Textdokuments ist zumeist Fließtext, und das Vorbereiten von Text zum Druck beginnt bei dem Platzieren von einzelnen Buchstaben. Dabei gilt es für je zwei Buchstaben, den richtigen Abstand zueinander zu finden. Beispielsweise werden die Buchstaben 'A' und 'V' etwas dichter aneinander gerückt. Dies ist das sogenannte Kerning (zu Deutsch: Unterschneidung). Einige Buchstabenkombinationen haben gar ein eigenes Erscheinungsbild, welches sich nicht durch einfaches Unterschneiden erreichen lässt. Dies sind sogenannte Ligaturen.

Abgesehen von Zeilenumbrüchen, für welche auch stilistische Regeln[1] existieren, gilt es, Witwen und Waisen bei Seitenumbrüchen zu vermeiden. Witwen sind Zeilen am Ende eines Absatzes, welche am Anfang einer Seite stehen. Demgegenüber sind Waisen Zeilen am Anfang eines Absatzes, welche am Ende einer Seite stehen. Ein guter Satz geht mit einer konsistenten und diversen Interpunktion einher. Der Kontext und die konkrete Verwendung der Zeichen zur Interpunktion variieren oft stark in Abhängigkeit von der geschriebenen Sprache. Aber auch innerhalb einer geschrieben Sprache ist Spielraum für die verwendeten Zeichen zur Interpunktion. Dabei sind diese oft diverser als vermutet. Als Beispiele seien hier horizontale Striche sowie Anführungszeichen aufgeführt. Einige der in Unicode zur Verfügung stehenden horizontalen Linien sind folgende:

- MINUS findet Verwendung im mathematischen Kontext (Beispieldarstellung: −, Deutsch: Minus).
- HYPHEN wird beispielsweise als Trennstrich oder Bindestrich verwendet (Beispieldarstellung: -, Deutsch: Viertelgeviertstrich).
- EN-DASH trennt oft Von-Bis Angaben (Beispieldarstellung: –, Deutsch: Halbgeviertstrich).
- EM-DASH kann als Gedankenstrich verwendet werden (Beispieldarstellung: —, Deutsch: Geviertstrich).

Für Anführungszeichen gibt es ebenfalls zahlreiche Varianten, etwa englische, deutsche oder französische Anführungszeichen. Insbesondere wird eine konsistente Verwendung von Anführungszeichen notwendig, wenn geschachtelte wörtliche Rede im Text vorkommt. [Knuth (1984)] Konkrete Vorschläge zur Verwendung der diversen Interpunktion finden sich etwa in der DIN 5008.

[1] Das Textsatzprogramm TEX implementiert einen Algorithmus zum Erzeugen von Zeilenumbrüchen. Für nähere Informationen sei das Buch 'The TEXBook' von Donald E. Knuth empfohlen. [Knuth (1984)]

2.2. Turing-Vollständigkeit

In diesem Abschnitt soll eine Beschreibung für Turing-Vollständigkeit gegeben werden. Da keinerlei Beweise zur Turing-Vollständigkeit einer Sprache in dieser Arbeit geführt werden, wird auch keine streng fromale Definition gegeben.[2] Dennoch soll dem Leser ein Gefühl dafür vermittelt werden, was Turing-Vollständigkeit bedeutet und was nicht.

Eine formale Sprache heißt Turing-vollständig, wenn es möglich ist, eine universelle Turingmaschine semantisch darzustellen. Dabei wird angenommen, dass bei Interpretation der Sprache unbegrenzter Speicher zur Verfügung steht. Ein System, welches Turing-vollständig ist, kann also eine universelle Turingmaschine simulieren. Das bedeutet unter der Annahme der Church-Turing-These, dass sämtliche sinnhafte Berechnungen mit einem solchen System vollzogen werden können. Über den Speicherbedarf und die Laufzeit einer Simulation einer universellen Turingmaschine wird keine Aussage getroffen. Es handelt sich also um eine theoretische Aussage ohne eine zwingend praktikable Umsetzung.

2.3. Markup-Languages

Markup-Languages (Kurz: ML, Deutsch: Auszeichnungssprachen) sind formale Sprachen, welche zum Gliedern von unterschiedlichen Daten genutzt werden. Dabei sind die Gliederungselemente und die Daten selbst in einer für Menschen verständlichen Form. Die Gliederungselemente werden auch Markups genannt. Dokumente, die in einer Markup-Language verfasst sind, können praktisch immer mit einem simplen Texteditor erstellt, geöffnet und gelesen werden. Viele Markup-Languages sind nicht entworfen, um Textdokumente zu gliedern. Allerdings bezeichnet in dieser Arbeit eine Markup-Language eine Sprache, die entworfen wurde, um Textdokumente zu erstellen. In einer solchen Sprache lässt sich etwa ausdrücken, ob ein Wort fett gedruckt sein soll oder ob ein Abschnitt ein Kapitel ist. Um einen mit Markups angereicherten Text entsprechend darstellen zu können, werden Interpreter genutzt. Diese verleihen einer ML erst Semantik und erzeugen beispielsweise eine PDF-Datei aus dem Quelltext. Dabei sind Markups in zwei Arten aufzuteilen:

- Darstellende Markups markieren Text in der Erwartung, dass dieser auf eine spezifische Art dargestellt wird.

- Beschreibende (oder auch semantische) Markups teilen einem Abschnitt eine Rolle zu, ohne das Erscheinungsbild zu spezifizieren.

Ein Markup für fette Schrift ist darstellend. Ein Markup, um Kapitel auszuzeichnen, ist semantischer Art. Bei beschreibenden Auszeichnungen können die Rollen in unterschiedlichen Darstellungen münden, müssen jedoch keinerlei Auswirkung auf das Aussehen des Textes haben, etwa wenn die Markierung zum maschinellen Verarbeiten des Textes gedacht ist. Einige bekannte Markup-Languages sollen hier kurz vorgestellt werden:

- TEX ist ein Textsatzprogramm, welches von Donald E. Knuth entwickelt wurde. Ziel dieser Sprache ist es, qualitativ hochwertig gesetzte Textdokumenten zu erzeugen. Gerade in den akademischen Kreisen der Naturwissenschaften und Technik erfreuen sich Variationen und Erweiterungen der Sprache großer Beliebtheit. Dies liegt mitunter an der Möglichkeit, schnell und hochwertig Formeln setzen zu können.

[2] Die Turing-Vollständigkeit von Tucup wird nicht weiter diskutiert, da einige Funktionalitäten wie Schleifen und Arithmetik noch nicht implementiert sind. Sobald diese Funktionen eingebaut sind wird eine Beweisführung erheblich erleichtert.

- DocBook ist vorrangig darauf ausgelegt, Bücher und Dokumentationen zu großen Softwareprojekten zu erstellen. Um diesem Ziel gerecht zu werden, ist DocBook ausdrucksstark und bietet eine Bandbreite an Markups. [DocBook]

- HTML wurde entworfen, um Webseiten zu gestalten. Die Interpretation dieser Sprache bleibt den Browsern überlassen. Abgesehen von der optischen Gestaltung von Webseiten ist HTML darauf ausgelegt, Funktionalität des interpretierten Quelltextes zu ermöglichen. Beispielsweise kann in HTML spezifiziert werden, dass beim Drücken eines Buttons eine Anfrage ausgeführt wird. Jedoch wird HTML kaum noch alleinstehend genutzt, sondern in Kombination mit CSS, um die Darstellung zu spezifizieren, und JavaScript, um weiter Funktionalität zu implementieren.

Die bisher vorgestellten ML's sind ausdrucksstark, bieten also eine Vielzahl von Funktionalitäten. TEX ist sogar allgemein als Turing-vollständig anerkannt. Diese vielen Funktionen sind nötig, um die Aufgaben, für welche die Sprachen entworfen wurden, zu erfüllen. Mit TEX lassen sich sehr detailliert druckfertige Dokumente erstellen. Sehr große Dokumentationen lassen sich mit DocBook einheitlich über Firmenstrukturen hinweg spezifizieren, und HTML bietet in Zusammenarbeit mit CSS und JavaScript die Möglichkeit, dynamische und funktionale Webseiten zu gestalten.

All diese Funktionalität benötigt eine entsprechend komplexe Syntax und birgt damit einen Mehraufwand, wenn das Ziel ist, einen simplen Text zu verfassen. Daher gibt es auch eine Bandbreite von sogenannten simplen Markup-Languages, welche die Anzahl der zur Verfügung stehenden Markups stark einschränkt. Dadurch reduziert sich die Mehrarbeit bei der Abfassung eines Dokumentes. Bekannte Vertreter sind Markdown und ReStructuredText. Diese Sprachen kommen oft bei der Dokumentation von Softwarecode zum Einsatz.

Kapitel 3

Motivation und Anforderungen an Tucup

Die Motivation, diese Arbeit zu schreiben, begründet sich darin, dass es kein Programm zu geben scheint, welches allen Anforderungen genügt, welche in diesem Kapitel spezifiziert werden. Die hier gestellten Forderungen sind von anderen Programmiersprachen inspiriert, und werden — für sich genommen — nochmals motiviert.

3.1. Trennung von Form und Inhalt

Die Trennung von Form und Inhalt ist ein Prinzip, nach dem die Erscheinungsform eines Dokumentes getrennt von dessen (zumeist textuellem) Inhalt behandelt wird. Beispielsweise wird der Inhalt einer Webseite mit Hilfe von HTML-Tags strukturiert, während die Darstellung aus der Interpretation von HTML/CSS/JS-Quellcode durch den Browser festgelegt wird. Um Missverständnissen vorzubeugen, wird im Folgenden die Form eines Dokumentes auch Layout genannt. Der Verfasser eines Dokumentes nennt sich Autor, daher wird die Domäne des Inhaltes als Autoren-Seite bezeichnet. Der Entwickler von Layouts nennt sich Designer, und entsprechend wird die Domäne des Layoutings als Designer-Seite bezeichnet.

Um Quelltext — also den Inhalt eines Dokumentes — in formatierter Form darzustellen, werden zusätzliche Mittel den Autoren bereitgestellt. Um eine Trennung von Form und Inhalt zu ermöglichen, sind diese Mittel in der Praxis häufig in Form von Markups implementiert. Eine Erweiterung der Quelltextes um Markups nennt sich Struktur. Wir definieren die Struktur des Inhaltes als die Gesamtheit der im Quelltext tatsächlich genutzten Markups, und die Struktur der Form als die Gesamtheit der zur Verfügung stehenden Markups im Textdokument.

Eine Anmerkung zu darstellenden Markups: Darstellende Markups geben dem Autor die Möglichkeit, in das Layout einzugreifen. Daher weichen diese die Trennung von Form und Inhalt auf und sollten deshalb vermieden werden, um eine möglichst strikte Trennung zu gewährleisten.

3.2. Konsistenz der Semantik in Raum und Zeit

Quelltext zusammen mit einem Layout sollte stets das gleiche druckfertige Dokument als Ergebnis einer Interpretation aufweisen. Das fertige Dokument ist hierbei als die Bedeutung/Semantik des Quelltextes zusammen mit einem Layout zu betrachten. Dies ist die Forderung nach Konsistenz. Der Ort, also die Maschine, auf der die Interpretation läuft, sollte keinerlei Einfluss auf das Ergebnis haben (Portabilität/räumliche Konsistenz). Aber auch künftige Interpretationen, welche möglicherweise erst in Jahrzehnten stattfinden, sollten das gleiche Dokument liefern (zeitliche Konsistenz). Ein inkonsistentes Verhalten der Software in Raum oder Zeit erschwert das Replizieren von Dokumenten. Daher ist zu möglichst strenger Konsistenz zu raten. Zum Erreichen von Konsistenz kann etwa eine Standardisierung zur Hilfe gezogen werden, welche das Verhalten von grundlegenden darstellenden

Markups spezifiziert. Solch eine Spezifizierung ermöglicht auch die Implementierung verschiedener Interpreter, welche sich nicht in den gelieferten Ergebnissen unterscheiden.[1]

3.3. Erweiterbarkeit von Layouts

Komplexe Systeme, welche mit zunehmender Konkretisierung der Anforderungen in sehr detailreiche und technische Bereiche ufern, sind ohne Bootstrapping nur sehr schwer zu händeln. Betriebssysteme etwa müssen auf technischer Ebene mit Prozessorstrukturen und Festplattenzugriffen arbeiten. Bootstrapping erlaubt, diese Ebene zu abstrahieren und zu isolieren. Ebenso ist das Setzen von Text ein aufwändiger Prozess, welcher mit der Digitalisierung desselben nicht an Komplexität verliert. Daher sollte auch hier Bootstrapping ermöglicht werden. Ein Format sollte dazu Funktionalitäten eines anderen importieren können. Dadurch ist etwa Abstraktion und Entkopplung möglich. Des Weiteren lässt sich so theoretisch dem Nutzer die volle Anpassbarkeit der Software anbieten — vom Hinzufügen eines 'first-level' Markups zu einem End-Layout (siehe unten) bis hin zum Eingriff in den Algorithmus zum Ermitteln passender Zeilenumbrüche. Inwieweit ein Designer in die Software eingreifen können sollte, ist in der Praxis zu ermitteln. Layouts sollten also aufeinander aufbauen. Typischerweise sollten Layouts mit beschreibender Struktur auf Layouts mit darstellender Struktur aufbauen. Ein Teil der Formate sollte nicht dem Autor zur Verfügung steht. Stattdessen sollten grundlegende Funktionalitäten in Form von darstellenden Markups, der Designer-Seite zur Verfügung stehen, um eben den Designern die Arbeit zu erleichtern. Diese Formen, welche nicht der Autoren-Seite zur Verfügung gestellt werden, nennen wir Technische-Layouts. Layouts, die Autoren zur Verfügung stehen, also jene, deren Markups im Quelltext verwendet werden können, sollen End-Layouts genannt werden. Wenn hier ein Layout nicht näher spezifiziert wird, ist mit Layout End-Layout gemeint.

3.4. Modulare Layouts

Eine Trennung von Form und Inhalt ermöglicht, es ein Layout mit unterschiedlichen Inhalten zu füllen. Doch wollen wir fordern, dass nicht nur der Inhalt für eine fixe Form leicht ausgetauscht werden kann, sondern auch die Form für einen fixen Inhalt austauschbar sein soll. So können aus dem selben Inhalt unterschiedliche Dokumente erstellt werden, etwa eine Präsentation und ein Handout dazu. Auch die Formatierung eines Artikels für verschiedene Journale wird durch einfache Austauschbarkeit von Layouts erleichtert. Insbesondere bedeutet dies, dass ein Layout, welches rein äußerlich (also von der Struktur her) alle Markups eines anderen Layouts anbietet, dieses ersetzen kann. Außerdem sollten möglichst viele Layouts austauschbar, also modular sein.

3.5. Spezifikation der Strukturen

Die Forderung nach modularen Layouts ist nicht bedingungslos. Nicht jedes Layout muss zwingend zu einem beliebigen Quelltext passen. Denn dies würde bedeuten, dass die Struktur aller Layouts vereinheitlicht wird, und somit auch, dass jedes Layout alle Markups zur Verfügung stellt, die es gibt. Dies würde die Erweiterbarkeit von Layouts absolut unpraktikabel machen, da das Erweitern eines Layouts die Erweiterung aller anderen nach sich zöge. Auch wenn eine bedingungslose Modularität nicht gewünscht ist, sollte eine Kompatibilitätsprüfung von Inhalt und Form einfach machbar sein,

[1] Eine alternative zu einer Spezifikation wäre es ein Plattform unabhängiger Master Interpreter, welcher keine neuen Versionen ausspielt. Dies bedarf jedoch einer sehr stabilen Implementierung. TEX etwa ist ein Beispiel für diese Vorgehensweise.

um das Tauschen von Layouts zu erleichtern. Daher sollte für ein Layout klar spezifiziert sein, welche Markups von diesem zur Verfügung gestellt werden. Die Struktur eines Layouts ist also analog zu Interfaces in Programmiersprachen, wie etwa Java, zu betrachten.

3.6. Konfigurierbare Layouts

Die programmtechnische Implementierung von Layouts ist kompliziert, da Kenntnisse in Bereichen Programmierung und Textsatz von Nöten sind. Einige Attribute von Layouts hingegen sind einfach verständlich, wie etwa die verwendeten Schriftarten und Schriftgrößen, der Zeilenabstand und Einrückungen von Überschriften. Der Satz von Schriftzeichen, das automatisierte Verhindern oder Erzwingen [2] von Witwen und Waisen, die Berechnung von 'hübschen' Zeilenumbrüchen und vieles mehr werden voraussichtlich immer eine Aufgabe sein, die Fachwissen erfordert. Letzteres sollte sich in der Implementierung von Technischen-Layouts niederschlagen. Einfach verständlich Eigenschaften jedoch lassen sich zumeist in Zahlen oder Zeichenketten ausdrücken. Daher können diese leicht in Konfigurationsdateien ausgelagert werden. Solche Konfigurationen bieten auch einem nicht technisch versierten Nutzer die Möglichkeit, Einfluss auf das Layout zu nehmen und somit auf der Designer-Seite tätig zu sein.

3.7. Restriktive Layouts

Ein Merkmal für qualitativ hochwertige Software ist, dass Fehlbedienungen in einem frühen Stadium unterbunden werden. Insbesondere ist dies auch in dem Entwurf von Programmiersprachen zu beobachten. So behauptet auch Robert C. Martin, dass erfolgreiche Programmierparadigmen allesamt Restriktionen für den Nutzer forcieren. Strukturelles Programmieren etwa verbietet die Nutzung von GoTo-Ausdrücken oder funktionale Programmierung (unter anderem) die Neuzuweisung von Variablen.[Martin] Daher sollte ein Layout in der Lage sein, den Autor insofern einzuschränken, dass unabsichtlicher Missbrauch und die qualitätsmindernde Verwendung von Markups nicht zugelassen werden.

3.8. Statische Prüfung von Quelltext

Heutzutage sind statische Analyse-Tools beim Programmieren kaum noch wegzudenken. Dabei ist bereits eine Syntaxprüfung als statische Analyse zu sehen. Die Möglichkeit zu statischen Analysen wird zumeist vom Compiler gestellt. Um solche Analysen bereits vor der Kompilierung zur Verfügung zu haben, gibt es oft Plugins für IDEs oder einen Language-Server. Mit solchen Mitteln sind häufig Funktionen wie 'go-to-definition' oder Refactoring von Variablennamen möglich. Da die Designer-Seite de facto Funktionalität programmieren muss, ist ein Language-Server mit umfangreichen statischen Analyse-Tools sehr zu empfehlen sowie eine umfangreiche Analysefunktionalität des Compilers.

3.9. Dokumentation von Strukturen

In Programmiersprachen sind aus Kommentaren automatisch erzeugte Dokumentationen nicht unüblich. So gibt es etwa Tools wie Javadoc, Haddock für Haskell oder Docstrings in Python. Gerade dann,

[2] Die Erzwingung von Witwen und Waisen kann nötig sein, um Negativbeispiele zu generieren.

wenn ein Nutzer sich mit der API eines fremden Moduls auseinandersetzt, können zusammengefasste Dokumentationen oder Tooltips in einer Entwicklungsumgebung helfen. Betrachtet man eine ML als Programmiersprache, so ist die Struktur eines Layouts schlicht eine API. Zudem ist die Dokumentation von semantischen Markups sehr wichtig, da zur richtigen Nutzung die Bedeutung des Markups verstanden werden muss. Eine Konvention zur Dokumentation sowie ein Werkzeug zum Bereitstellen dieser Dokumentation ist somit ein großer Schritt in Richtung einer benutzerfreundlichen Software.

3.10. Ungestörter Schreibfluss

Die Syntax sollte den Schreibfluss eines Autors so wenig wie möglich unterbrechen. Daher sollten die Sonderzeichen der Syntax mit Bedacht gewählt werden. Diese sollten nicht zu häufig in Fließtexten vorkommen, da jedes Mal ein Escaping stattfinden muss. Außerdem sollte der Autor keine allzu komplizierten Bewegungen mit den Fingern ausführen müssen, um der Syntax zu genügen. Sonderzeichen, welche auf herkömmlichen Tastaturen also nur schwer zu erreichen sind, sollten gemieden werden. Da sich unterschiedliche Tastaturen und Tastaturlayouts[3] in verschiedenen Teilen der Welt etabliert haben, sollte auch eine Anpassbarkeit der Syntax in Betracht gezogen werden. Die Vor- und Nachteile sind jedoch noch abzuwägen. In dieser Arbeit wird sich auf QWERTZ-Tastaturen beschränkt.

3.11. Praktikables Programmieren

Über die Trennung von Form und Inhalt hinaus geht der Wunsch, inhaltliche Daten vor dem Erzeugen eines druckfertigen Dokumentes zu berechnen. Dabei ist diese Forderung sehr naheliegend. Berechnungen von Daten sind bereits für simple Anforderungen notwendig, etwa für das automatische Erzeugen eines Datumsstempels. Weitere Beispiele wären das Einlesen einer CSV Datei oder Code-Reflection, wobei der Autor etwa einen Teil des Quelltextes ad hoc, also ohne Duplikation von Code, in seinem Dokument abbilden möchte. Auch das Erzeugen von Zufallsdaten könnte etwa zum automatischen Erstellen von Schülertests genutzt werden. Um die Möglichkeit zu sämtlichen Berechnungen einzuräumen, sollte Turing-Vollständigkeit gegeben sein.

Da der Inhalt in einem solchen System nicht mehr allein von einem Menschen stammt, wird das Programm selbst zu einem maschinellen Co-Autor. Da dieser Co-Autor Turing-vollständig, aber nicht eigenständig denkend ist, gilt es, Vorsicht bei der Nutzung walten zu lassen. Möglicherweise ist es vernünftig, Berechnungen nicht in der Quelldatei des Dokumentes zu spezifizieren. Das verwässert einzig die Trennung von Form und Inhalt, wobei das Ergebnis der Berechnungen Inhalt ist. Aber die Berechnungsvorschrift an sich ist eben kein Inhalt. Andererseits kann aber argumentiert werden, dass dies nicht die Erscheinungsform der unterschiedlichen Strukturelemente beeinflusst (wenn eine saubere Implementierung vorliegt).

Daher mag eine weitere Trennung von Form und kalkulierten Daten stattfinden. Nennen wir dieses Prinzip der Einfachheit halber die Trennung von Form und Daten. Dabei mag dies bereits nach einer sinnigen Forderung klingen, da sich ein Designer auf das Design und nicht auf das Berechnen von Inhalten fokussieren sollte. Möglicherweise ist aber eine Trennung nicht sinnvoll. Denn wenn bereits das automatische Einfügen von einem Datum als Co-Autor Aktion des Interpreters gewertet wird, gilt dies dann auch für das Ergänzen von Anführungszeichen bei einem wörtlichen Zitat? In solchen Fällen überschneiden sich Form und Daten. Wenn sich diese Schnittmenge als zu groß herausstellt, ist eine Trennung von Form und Daten möglicherweise nicht praktikabel. Ob eine solche Trennung sinnvoll ist, wird sich wohl erst in der Praxis zeigen. Eventuell stellt sich heraus, dass die Berechnung von Inhalt

[3] Tastaturlayout ist nicht im Sinne von Form gemeint.

ebenso im Bereich des Layouts aufgehoben sein sollte, wie es auch klassische Markup-Definitionen sind. Daher wird hier in dieser Hinsicht nicht unbedingt auf eine Trennung von Form und Daten gedrängt.

Wenn wir fordern, dass inhaltliche Daten kalkulieren zu können, dann können wir dies als die Forderung nach Turing-Vollständigkeit bezeichnen. Doch der Titel dieses Unterkapitels lautet bewusst nicht 'Turing-Vollständigkeit' sondern 'Praktikables Programmieren'. Denn nur, weil sich Daten kalkulieren lassen, ist das Niederschreiben der Berechnungsvorschriften nicht notwendigerweise ergonomisch. Daher wird nebst der Turing-Vollständigkeit auch die erkennbare Ähnlichkeit zu gängigen Programmiersprachen und die Implementierung von modernen Programmierparadigmen gefordert. Speziell ist damit das Paradigma der strukturellen Programmierung sowie Objekt-Orientierung gemeint.

Um an einigen Stellen dieser Arbeit wird, gezeigt, dass wie die Anforderung nach praktikablen Programmieren erfüllt wird. soll stets ein kleines, aber in der Anforderung gleichbleibendes Beispiel angeführt werden. In diesem Beispiel geht es um das Erzeugen eines Dokuments, welches zehn zufällig erzeugte Additionsaufgaben enthält. Die Aufgaben sollen untereinander geschrieben stehen und jeweils eine Addition von zwei Zahlen sein, welche zufällig zwischen 0 und 1000 gewählt werden. Die Aufgaben sollen nummerierte Überschriften haben. Eine Lösung zu dieser Problemstellung, implementiert in Tucup, findet sich in Anhang B, und das zugehörige Ergebnis in Anhang C.

Kapitel 4

Abgleich vorhandener Software mit den gestellten Anforderungen

Je nachdem, in welchem Bereich Textdokumente erstellt werden, sind die Anforderungen unterschiedlich. Beispielhafte Anforderungen:

- Im Privaten ist oft eine kurze Einarbeitungszeit in das Erstellen von Dokumenten gewünscht.
- Für naturwissenschaftliche Veröffentlichungen sind das Einhalten einer bestimmten Form sowie gut leserliche Formeln wichtige Kriterien.
- Unternehmen, welche Rechnungen ausstellen, müssen beim Erstellen, Zugriff auf etwa die Kundendaten haben und benötigen eine Automatisierung des Prozesses.
- Webauftritte benötigen ein angepasstes Erscheinungsbild des Inhaltes, um auf Bildschirmen verschiedener Größe gut lesbar zu sein.

Für solche speziellen Anforderungen wurden mit der Zeit viele Lösungen in Form von Software erarbeitet. Einige dieser Systeme sollen nun mit den Anforderungen aus dem vorhergehenden Kapitel abgeglichen werden.

4.1. TEX-Familie

In den 70er und 80er Jahren entwickelte Donath E. Knuth das Textsatzprogramm TEX. Dieses System versetzt den Nutzer in die Lage hervorragend gesetzte, Dokumente auf digitalem Wege zu erstellen. Speziell wurde bei dem Entwurf auf das Setzen von Formeln geachtet, was unter anderem der Grund ist, dass TEX in technischen und naturwissenschaftlichen Kreisen noch heute eine große Anhängerschaft hat. Das vorrangige Ziel von TEX war es das Setzen von Dokumenten zu ermöglichen. Zu diesem Zwecke wird Gebrauch von darstellenden Markups gemacht, sogenannten Makros. Das Augenmerk liegt also nicht auf der Trennung von Form und Inhalt. Von TEX gibt es zahlreiche Varianten. Donald E. Knuth hat durchgesetzt, dass keine dieser Varianten TEX heißen darf. Diese Restriktion ist die einzige, welche die Nutzung von TEX und dessen Quellcode betrifft. Außerdem ist TEX portabel und erzeugt auf allen Geräten, auf denen es läuft die gleichen Ergebnisse. [Knuth (1984)] Somit wird die Forderung nach Konsistenz in Raum und Zeit mit beispielloser Striktheit erfüllt. Mehrfach wurde gezeigt, dass sich auch komplexe Programme mit TEX realisieren lassen. [Raichle (1995)] [Murrish] Daher scheint es allgemein anerkannt, dass TEX Turing-vollständig ist. Donald E. Knuth achtete beim Entwurf auf die Erweiterbarkeit der Software, und so gibt es die Möglichkeit, neue Makros zu definieren. Dennoch wurde TEX im Wesentlichen zum Setzen von Dokumenten entworfen und nicht als Programmiersprache. Typisierung gibt es nicht, Kontrollstrukturen stehen nur als Erweiterungen zur Verfügung, und Zuweisungen funktionieren nur durch die Definition von Makros. Eine moderne und komfortable Programmierung ist unter diesen Umständen nur mit großer Disziplin möglich.

So ließen sich etwa Kontrollstrukturen, Zuweisungen und Typisierung programmieren, da Turing-Vollständigkeit (dem Konsens nach) gegeben ist. Doch lässt sich kein Standard zum Programmieren in TEX ausmachen. Daraus folgt, dass jeder Nutzer seine eigene Praxis zum Programmieren mit TEX erarbeiten muss. Dies soll in dieser Arbeit nicht als praktikables Programmieren gelten. Des Weiteren stellen die bekanntesten Editoren und Werkzeuge [1] zum Schreiben von TEX keine Funktionalität wie etwa 'go-to-definition' zur Verfügung. Diese Funktionalität ist für herkömmliche Sprachen Standard.

Die Erweiterbarkeit von TEX hatte zur Folge, dass verschiedene TEX Interessierte ihre eigenen Erweiterungen schreiben konnten. Um eine Möglichkeit zu schaffen, sogenannte TEX-Sammlungen allen Nutzern zur Verfugung zu stellen, wurde CTAN (Comprehensive TEX Archive Network) zu Beginn der 90er Jahren aufgebaut. CTAN ist eine Art Package Management System. Dieses läuft über ein Netzwerk von Servern, welche alle verfügbaren TEX-Sammlungen speichern und an Nutzer ausliefern können. Ein Kern-Server regelt die Einspeisung und das Updaten von TEX Erweiterungen. Durch CTAN sind mittlerweile über 6000 Pakete weltweit verfügbar. [CTAN-Team] Dadurch sind viele Vorlagen für diverse Dokumentarten nutzbar, ohne diese selbst schreiben zu müssen. Damit trägt CTAN sicherlich einen großen Beitrag zur Popularität von TEX bei.

Leslie Lamport machte in den 80er Jahren von der Möglichkeit, TEX zu erweitern Gebrauch. Er schrieb ein Paket von Makros, welches unter dem Namen LATEX bekannt ist. Dieses Paket bietet eine Reihe semantischer Markups, um zum Beispiel Kapitelüberschriften auszuzeichnen. Auch algorithmische Funktionalität ist in LATEX integriert, um beispielsweise ein Inhaltsverzeichnis automatisch zu erzeugen. Somit ist die Trennung von Form und Inhalt zumindest in der Idee von LATEX vertreten. Die Layouts haben keine Restriktionen, welche die Autoren an Regeln binden, die über die Syntax von TEX hinaus gehen. Zusätzlich hat LATEX eine eher unspezifische Konvention für die Syntax von Makros entwickelt. Dies erleichtert die Handhabung auf einem höheren Abstraktionslevel. Allerdings baut diese Syntax auf den primitiven Makros von TEX auf und daher zu umgehen. Zudem bieten viele LATEX-Packages Makros an, welche in ihrer Bedienung von dieser Syntax abweichen. Alles in allem ist die Möglichkeit, Berechnungen durchzuführen, auch mit LATEX nicht sehr praktikabel.

ConTEXt ist ebenso wie LATEX eine Erweiterung von TEX um eine Menge von Makros. Ein Vorteil gegenüber LATEX ist die Verfügbarkeit von wesentlich deutlicheren Konventionen. So gibt es eine konsistentere Nutzung von Klammern zur Übergabe von Parametern und Optionen. Auch eine klare Ordnerstruktur wird zur Nutzung von ConTEXt Projekten vorgeschlagen. Berechnungen lassen sich ebenfalls in ConTEXt durchführen, und zwar ohne den Zwang, TEX Makros dafür nutzen zu müssen, denn ConTEXt baut auf einer Variante namens LuaTEX auf. Diese bietet die Möglichkeit, Lua-Code in ein Dokument einzubetten. Zwar ist die Handhabung etwas eingeschränkt; so muss etwa auf Escaping für einige Zeichen im eingebetteten Code geachtet werden, doch praktikables Programmieren ist mit ConTEXt möglich. Im Vergleich zu anderen Systemen ist der nötige Code, um unser Demonstrationsdokument zu erzeugen, recht schlank gehalten. Dazu muss einzig folgender Lua-Code in ein ConTEXt Dokument eingebettet werden [2].

```
function newRandomAdditionString(limit)
        a = math.random(0,limit)
        b = math.random(0,limit)
        return "" .. a .. " + " .. b .. " = "
end
```

[1] Konkret wurden Overleave, TeXmaker, MiKTeX, TeXstudio sowie die VSCode Erweiterung 'LaTeX language support' betrachtet.

[2] Der geneigte Leser mag sich fragen wie genau Lua-Code eingebettet wird. Dazu finden sich nähere Informationen in der Dokumentation zu finden auf der ConTEXt Homepage https://wiki.contextgarden.net/

```
    function printExercise(exerciseNumber)
            exercise = newRandomAdditionString        (1000)
            context("Exercise " .. exerciseNumber .. "\n\n")
            context.startformula()
            context(exercise)
            context.stopformula()
            context("\n\n\n\n")
    end

for i = 1, 10, 1 do
            math.randomseed(os.time() + i)
            printExercise(i)
end
```

Damit ist ConTEXt wahrscheinlich die Software, welche die meisten der hier vorgeschlagenen Anforderungen erfüllt. Trotz der guten Reglementierung der Syntax durch Konventionen bleibt ConTEXt eine Erweiterung von (Lua)TEX, und die Konventionen bleiben somit unverbindlich (in einem technischen Sinne).

4.2. Simple Markup-Languages

Systeme wie TEX, LATEX und ConTEXt, aber auch TEX fremde ML's wie DocBook oder HTML, sind in ihrer Benutzung oft komplex, bieten dafür aber umfangreiche Funktionalität. Um Dokumente ohne komplexe strukturelle Ansprüche zu erstellen, ist eine Menge dieser Funktionalität oft nicht nötig. Außerdem sind die Lernkurven für solch komplexe Systeme oft flach. Diese Problematik wird von Simple Markup-Languages (auch Lightweight Markup-Language, kurz: SMLs) angesprochen. Diese bieten nur eine geringe Menge von Markups und reduzieren damit die Ausdrucksstärke. Dadurch wird das Design schlanker und die Handhabung erleichtert. Zudem ist auch eine Zielsetzung von SMLs, den Quelltext ansehnlich und lesbar zu gestalten. Auf Grund der nicht selten drastischen Einschränkungen von Markups werden leicht verständliche und damit fast ausschließlich darstellende Markups angeboten. Die Verwendung von semantischen Markups würde nur zu einer Verkomplizierung der Sprache führen. Häufig anzufindende Markups sind zum Markieren von fetten, kursiven oder unterstrichenen von Texten gedacht. Ebenfalls gibt es oft die Möglichkeit, Überschriften und Auflistungen sowie Aufzählungen auszuzeichnen. Die Reduktion von Markups hat allerdings zur Folge, dass SMLs zum praktikablen Programmieren nicht in Frage kommen. Aber dies ist auch nicht die Zielsetzung von SMLs. Beispiele für simple Markup-Languages sind etwa reStructuredText oder Markdown. Auf Letztere Sprache wollen wir etwas weiter eingehen. Markdown kompiliert zu XHTML[3] und bietet somit die Möglichkeit, CSS auf den Output anzuwenden. Ohne näher darauf einzugehen, ist damit die Anforderung für austauschbare Layouts gegeben sowie auch eine schwache Trennung von Form und Inhalt. Die Kompilation zu XHTML zeigt desweitern, dass die erzeugten Dokumente vornehmlich zur Darstellung auf Bildschirmen gedacht sind und nicht darauf ausgerichtet sind, druckreife Dokumente zu erstellen. Die Portabilität der Dateien ist bereits durch das reine Textformat gegeben. Interpreter gibt es nicht nur für diverse Betriebssysteme, sondern auch

[3] XHTML (Extensible Hypertext Markup Language) unterscheidet sich von HTML vor allem in der Spezifikation. Während HTML in SGML (Standard Generalized Markup Language) spezifiziert ist, wird XHTML mit Hilfe von XML (Extensible Markup Language) definiert.

in zahlreichen Implementierungen in verschiedenen Programmiersprachen. Die Spezifikation der leztendlichen Darstellung obliegt allerdings dem Programm, welches zum Interpretieren des erzeugten HTML-Codes genutzt wird. Restriktiv ist Markdown nur insofern, als dass die Struktur eingeschränkt ist. Da Markdown jedoch das Einbetten von HTML erlaubt, ist auch diese Einschränkung aufgeweicht.

Um die Vorstellung von Markdown abzuschließen, wird an dieser Stelle ein kleines Codebeispiel gegeben, welches Aufgabenzettel mit drei (nicht dynamisch erzeugten) Rechenaufgaben entspricht. Dabei bedeuten die führenden Doppelkreuze, dass nachfolgend eine Überschrift der Ebene vier folgt.

```
#### Exercise 1
1 + 2 =

#### Exercise 2
3 + 4 =

#### Exercise 3
5 + 6 =
```

4.3. HTML/CSS/JS

HTML steht für Hypertext Markup Language und nutzt sogenannte Tags als Markups. Entworfen wurde HTML, um Webseiten darzustellen. Damit ist die Zielsetzung gegenüber TucUP bereits signifikant anders. Die Trennung von Form und Inhalt wird normalerweise mit dem Verzicht auf die Nutzung vom 'style' Attribut und dem Einbinden von CSS (Cascading Style Sheet) umgesetzt. Ziel von HTML/CSS ist die Darstellung von Inhalt auf Bildschirmen. Denn die Maße von Bildschirmen sind sehr unterschiedlich, und damit müssen Browser beim Anzeigen von Webseiten umgehen. HTML und CSS sind also darauf ausgelegt, dynamisch in den Maßen des dargestellten Dokumentes zu sein. Zusätzlich ist es für eine Webseite vorrangig wichtig, dass diese sinnvoll angezeigt wird und nicht, dass diese auf allen Geräten bis ins Detail identisch ist, auch wenn eine identische Darstellung in machen Situationen gewünscht sein mag. Weiterhin bietet HTML auch Funktionalitäten zum Versenden von HTTP-Requests, etwa zum Senden von Form-Daten. Mit der Entstehung des Web2.0 haben sich die Anforderungen an HTML noch weiter in die Richtung von Funktionalität entwickelt, und das Einbinden von JavaScript in HTML ist heutzutage nahezu selbstverständlich. Interaktive, dynamische Dokumente (als welche sich Webseiten auch betrachten lassen) zu erzeugen, ist Hauptaufgabe von HTML, CSS und JavaScript. Durch das Einbinden von JavaScript lässt sich in einem solchen System schon nahezu selbstverständlich praktikabel Programmieren. Es ist auch kein Escaping wie bei ConTEXt von Nöten, einzig die Verwendung von einfachen oder doppelten Anführungszeichen braucht — etwa beim inline Coding — ein gewisses Augenmerk. Auch hier soll ein Beispiel in HTML mit eingebettetem JavaScript code gegeben werden, welches ein Arbeitsblatt mit zufälligen Additionsaufgaben erzeugt.

```
<html>
<body>

<p id="exercises"></p>

<script>
```

```
function getRandomIntegerUpTo(limit){
        return Math.floor(Math.random() * (limit+1));
}

function makeRandomExerciseTextTag(){
        const exerciseText = document.createElement("p");
        const a = getRandomIntegerUpTo(1000).toString();
        const b = getRandomIntegerUpTo(1000).toString();
        exerciseText.innerHTML = a.concat(" + ", b, " =");
        return exerciseText;
}

function makeExerciseHeadingTag(exerciseNumber){
        const exerciseHeading = document.createElement("h3");
        exerciseHeading.innerHTML = "Exercise ".concat(exerciseNumber.toString());
        return exerciseHeading;
}

function makeExerciseTag(exerciseNumber){
        const exercise = document.createElement("div");
        exercise.appendChild(makeExerciseHeadingTag(exerciseNumber));
        exercise.appendChild(makeRandomExerciseTextTag());
        return exercise;
}

for (let i = 1; i <= 10; i++) {
        const exercise = makeExerciseTag(i);
        document.getElementById("exercises").appendChild(exercise);
}
</script>

</body>
</html>
```

Gerade auch durch die weite Verbreitung von HTML und JavaScript gibt es eine Reihe von Tools zur Unterstützung, welche auch eine Reihe statischer Validierungen durchführen. Wird JavaScript durch eine alternative Programmiersprache ersetzt, welche zu JavaScript kompiliert (etwa TypeScript), lässt sich sogar Typsicherheit beim Programmieren erreichen.

4.4. Module und Packages zum PDF-Erzeugen

Nicht nur im Web-Kontext ist es wichtig, aus vorliegenden Daten automatisch PDF-Dateien zu erzeugen. Zahlreiche Programme sind in der Lage, PDF-Dateien automatisch zu generieren. Dafür gibt es zahlreiche Module und Packages unterschiedlicher Programmiersprachen. Stellvertretend wird hier das JavaScript Modul pdfmake untersucht. Eine Version dieses Moduls steht auch mit Typenannotationen für TypeScript zur Verfügung. Wenn TypeScript mit entsprechenden Annotationen genutzt wird, stehen also auch Typen bei der Programmierung zur Verfügung. Und auch das automatisierte

Einspielen von Daten in ein zu erzeugendes Dokument ist leicht gemacht. Praktikables Programmieren ist also schon allein daher möglich, da die Sprache, in welcher das Dokument verfasst wird, eine etablierte Programmiersprache ist.

Zur Demonstration ist hier ein TypeScript Programm gegeben, welches unser Beispiel für einen Arbeitsbogen als PDF-Datei erzeugt. Hierbei wird nur der für den Inhalt relevante Code angegeben, der komplette Code ist in Anhang A zu finden.

```
function getRandomIntegerUpTo(limit: number): number{
    return Math.floor(Math.random() * (limit+1));
}

function makeExerciseText(): DocumentContent{
    const a = getRandomIntegerUpTo(1000).toString();
    const b = getRandomIntegerUpTo(1000).toString();
    return a.concat(" + ", b, " =");
}

function makeExerciseHeading(exerciseNumber: number): DocumentContent{
    return {
        text: "Exercise ".concat(exerciseNumber.toString()),
        style : {
          fontsize: 16,
          bold: true,
        },
    };
}

function makeExercise(exerciseNumber: number): DocumentContent{
    return [
        makeExerciseHeading(exerciseNumber),
        makeExerciseText(),
        "\n\n",
    ];
}


function makeDocDefinition(): DocumentDefinition {
    const content = [];
    for (let i = 1; i <= 10; i++) {
        const exercise = makeExercise(i);
        content.push(exercise);
    }
    return {
        content: content,
    }
}
```

Wie das Beispiel veranschaulicht, ist die Handhabung sehr technisch und nicht darauf ausgelegt, von einem Nutzer ohne Programmiererfahrung genutzt zu werden. Die Trennung von Form und Inhalt ist daher auch nicht vorgegeben und bleibt dem Designer/Autor selbst überlassen. Insofern ist auch die Möglichkeit, konfigurierbare, erweiterbare, modulare und restriktive Layouts mit spezifischen Strukturen zu schaffen, gegeben. Es gibt aber keinen Konsens für eine konkrete Umsetzung solcher Anforderungen. Die Portabilität ist dadurch gegeben, dass die JavaScript-Laufzeitumgebung Node für alle gängigen Betriebssystemen zur Verfügung steht. Allerdings ist die Konsistenz der erzeugten Dateien über die Zeit eher fraglich, da gewartete Module häufig mehrmals im Jahr neue Versionen ausspielen. Im Jahr 2020 etwa wurden sechs Versionen von pdfmake veröffentlicht.[Pampuch] Rückwärtskompatibilität ist beim Ausspielen neuer Versionen natürlich ein oft gefordertes Kriterium, doch wächst mit häufigen Updates die Wahrscheinlichkeit, dass Inkonsistenzen zwischen den Versionen auftreten.

4.5. WYSIWYG-Editoren

Eine Word-Processor-Software wie etwa MicrosoftWord bietet einen rapiden Einstig in das Erstellen von druckfertigen Textdokumenten. Dieser wird durch das WYSIWYG (What you see is what you get) Prinzip ermöglicht. Dabei wird dem Nutzer während der Bearbeitung des Dokumentes angezeigt, wie es gedruckt aussehen würde. Nutzer arbeiten sich schneller in die Handhabung solcher Systeme ein, als dies etwa bei LaTeX der Fall ist. Für die meisten Arten von Textdokumenten ist auch die Produktivität höher. Das heißt, dass die gleiche Anzahl an Worten vom Nutzer schneller niedergeschrieben wird. Einzig für formellastige Texte hat sich LaTeX als die effizientere Software erwiesen. [Knauff (2014)] WYSIWYG-Editoren bieten häufig die Möglichkeit den Inhalt auf Rechtschreib- und Tippfehler zu kontrollieren, da entsprechende Werkzeuge mittlerweile zum Standard gehören. Eine Prüfung des Textes auf Konformität mit einem speziellen Layout ist nicht primäre Zielsetzung von solchen Systemen. Und per se ist es auch nicht möglich, ein Layout ohne Inhalt zu definieren.[4] Einzelne Teile sind zwar auch ohne Inhalt konfigurierbar, wie etwa die Breite der Seitenränder. Aber andere Teile wie die Form und Ausrichtung von Überschriften ist nur bedingt festzulegen. An sich hat der Autor zu jedem Zeitpunkt die Möglichkeit, in das Design einzugreifen. Das heißt: eine Trennung von Form und Inhalt ist nicht gegeben. Auch die Darstellung des Inhaltes kann variieren, wenn unterschiedliche Software zum Öffnen und Darstellen von Dateien genutzt wird. Auch die zeitliche Konsistenz ist nicht gegeben, da neuere Versionen oft Dateien früherer Versionen anders darstellen. Auch wenn Word etwa die Möglichkeit, gibt Makros zu programmieren, wird Visual Basic for Applications (VBA) zum Programmieren vorgegeben. VBA steht dabei nach einer Umfrage von Stackoverflow an der Spitze der unbeliebtesten Programmiersprachen 2020, was gegen eine gute Programmiererfahrung mit dieser Sprache spricht. [Stackoverflow]

[4] Die Funktionalität von bekannten WYSIWYG-Editoren wächst zunehmend. Daher soll hier betont werden, dass es Editoren geben mag, welche eine Trennung von Form und Inhalt zulassen. Dennoch wird diese nicht forciert.

Kapitel 5

Entwurf von Tucup

5.1. Programmierkonzepte

Um dem Ziel der Ähnlichkeit zu modernen Programmiersprachen zu genügen, sind einige Konzepte aus diesen in den Entwurf von Tucup eingeflossen.

Statische Typisierung ist ein wichtiges Konzept, um präventiv Fehler beim Programmieren zu vermeiden, ehe das Programm ausgeführt wird. Dabei werden extra Notationen im Code verlangt, welche also Teil der Syntax sind und welche die Typen von Variablen festlegen. Dadurch ist es möglich, statische Analysen auf dem Code durchzuführen, um Typenfehler zu finden. Allerdings implementiert der Interpreter keine statische Typisierung in diesem Stadium, auch wenn diese im Entwurf vorgesehen war.

Ein weiteres Mittel um präventiv Zugriffe auf nicht definierte oder instantiierte Variablen zu vermeiden sind statische/lexikalische Scopes. Auch eine solche Prüfung wird nicht vom Prototypen durchgeführt. Sämtliche Variablenzugriffe werden zur Laufzeit ausgeführt und stoppen den Prozess bei einem ungültigen Zugriff.

Aus der objektorientierten Programmierung wurde gänzlich die Idee von Objekten und Attributen übernommen. Tucup arbeitet fast ausschließlich mit Objekten, wobei diese möglichst simpel gestrickt sind. Ein Objekt kann behandelt werden wie ein assoziatives Array. Ein Objekt enthält Werte, welche einem bestimmten Attribut zugeordnet werden. Allerdings sind nicht alle Objekte veränderlich, teilweise können auch nur bestimmte Attribute gesetzt werden. Zusätzlich kann ein Objekt aufrufbar sein, solche Objekte können auch als Funktionen bezeichnet werden.

Mit dieser Definition von Funktionen ist auch das mittlerweile in den meisten Programmiersprachen vertretene Konzept 'Functions as first class citizens' implementiert. In diesem Zuge werden die lokalen Variablen einer Funktion in eben dem assoziativen Array gespeichert, welches jedem Objekt inne wohnt. Dabei hat die Funktion zur Laufzeit auch Zugriff auf Variablen, welche dem Programm angehören, das die Funktion aufgerufen hat. Gibt es Variablen mit dem gleichen Namen sowohl im aufrufenden Programm als auch unter den lokalen Variablen der aufgerufenen Funktion, überschatten die lokalen Variablen die des Programms.

Ein Programm ist ebenfalls ein Objekt im Speicher des Tucup-Interpreters. Dieses hält seine eigenen lokalen Variablen im objektinternen Speicher. Ein Aufruf eines Programm-Objektes entspricht dem Starten seiner Ausführung. Programm-Objekte nehmen keine Parameter beim Aufruf entgegen, dahingehend sind Programme zwar Funktionen, aber nicht jede Funktion ist ein Programm. Ein Programm speichert intern eine Liste von Ausdrücken, die bei einem Aufruf der Reihe nach evaluiert werden. Intern gespeicherte Daten sind nicht für den Nutzer als Attribut eines Objektes zugänglich, werden also nicht im Objektspeicher gehalten.

Ausdrücke sind Objekte, welche stets mit einem vom Nutzer getippten Code-Abschnitt korrespondieren. Der Aufruf solcher Objekte entspricht der Auswertung des Ausdrucks. Momentan implementierte Ausdrücke sind Aufruf-Ausdruck-Objekte (standard, zuweisend und ausschließlich zuweisend), Text-Ausdruck-Objekte und (Variablen)Zugriff-Ausdruck-Objekte.

Die vorhergehenden Abschnitte verdeutlichen, dass das Prinzip 'everything is an object' in Tucup nicht nur auf primitive Daten und Funktionen, sondern auch auf den geschriebenen Code selbst ausgeweitet wird. Dies bietet auch die Möglichkeit weitreichender Code-Reflection.

Seit den 60er Jahren ist Structured-Programming Stand der Technik und hat weitestgehend das GoTo-Programmieren abgelöst. Seither sind Kontrollstrukturen unabdingbar für eine moderne Highlevel-Programmierung. Somit ist auch Ziel von Tucup, Schleifen und Verzweigungen anzubieten. Die Umsetzung von Kontrollstrukturen erfolgt als Funktionen, welche eine interne Implementierung für ihren Aufruf bieten. Dabei nimmt beispielsweise die If-Funktion ein Boolean-Objekt, ein Programm-Objekt und ein weiteres optionales Programm-Objekt entgegen. Erst durch diese Art des Umgangs mit Kontrollstrukturen ist die Realisierung von Programmen und Ausdrücken als Objekte überhaupt erst sinnvoll geworden.

Die Entscheidung Kontrollstrukturen als Funktionen umzusetzen begründet sich in in einem möglichst ungestörten Schreibfluss für Tucup. Die Entscheidung eine einzige Sprache für alle Parteien, zu wählen, führt dazu, dass diese Sprache einen ungehinderten Schreibfluss für die Autoren-Seite bieten sollte. Dafür sind wenige Sonderzeichen notwendig. Zudem sollten Schlüsselworte, wenn vorhanden, mit Sonderzeichen markiert werden. An dieser Stelle angekommen, wäre eine Sonderbehandlung von Kontrollstrukturen gegenüber Funktionen eine Verkomplizierung, zumindest für den Nutzer.

5.2. Syntax

Ehe die Auswahl der Sonderzeichen diskutiert wird, werden die bisher implementierte, formale Syntax und einige Codebeispiele vorgestellt. Die Spezifikation der Syntax ist hier in erweiterter Backus-Naur-Form beschrieben und schrittweise besprochen.

Wie bereits zu erahnen, ist alles, was aus einem Tucup Dokument geparst, wird ein Programm-Objekt, welches dann vom Interpreter aufgerufen, also ausgeführt wird. Dies findet sich auch in dem ersten Teil der Syntax-Spezifikation wieder:

```
S ::= ExpressionLst

Program ::= ExpressionLst

Expression ::= Call
           |   Text
           |   Access
```

Die unterschiedlichen Ausdrücke (Expressions) werden im nächsten Teil der Spezifikation dargestellt:

```
Call ::= SimpleCall
     |  AssigningCall
     |  AssigningOnlyCall

SimpleCall ::= "<" VarNameChain ArgLst ">"

AssigningCall ::= "[" VarNameChain "]" "<" VarNameChain ArgLst ">"
```

```
AssigningOnlyCall ::= "{" VarNameChain "}" "<" VarNameChain ArgLst ">"

Text ::= text

Access ::= "*" VarNameChain "*"
```

Die Syntax spezifiziert auch Listungen von Ausdrücken, Variablennamen und Argumenten. Dabei ist eine Liste von Ausdrücken (ExpressionLst) dazu da, um eine Programmsequenz abzubilden. Die Trennung der Ausdrücke bedarf keiner Trennzeichen. Eine Liste von Variablennamen (VarNameChain) bildet den Zugriff auf Variablen und Attribute ab, wobei einzelne Variablennamen durch Punkte getrennt werden. Die Argumente, welche einem Aufruf übergeben werden, sind Programme. Dies ist deshalb so umgesetzt, um etwa Kontrollstrukturen oder einen Funktionskonstruktor als Funktion implementieren zu können. Argumente werden durch Bindestriche getrennt.

```
ExpressionLst ::= <empty>
              |   Expression
              |   Expression ExpressionLst


VarNameChain ::= varName
             |   varName "." VarNameChain

ArgLst ::= <empty>
       |  "-"
       |  "-" Program
       |  "-" Program ArgLst
```

Abgesehen von den Sonderzeichen kennt diese Syntax nur zwei Terminale, nämlich 'text' und 'varName'. Das 'text' Terminal steht dabei für eine Zeichenkette, welche keine (nicht escapeten) Sonderzeichen enthält. Das 'varName' Terminal ist eine Zeichenkette, welche keine Sonderzeichen oder Punkte enthält. Für diese Syntax sind nun einige Code-Beispiele aufgelistet, um informal die Semantik von Tucup zu erläutern. Auch wird die Auswahl der Zeichen begründet.

5.3. Code Beispiele

Das erste Beispiel zeigt, wie ein einfacher Textausdruck aussehen könnte. Dabei wird simpler Text ohne Sonderzeichen geschrieben. Die Interpretation des Programms liefert ein PDF-Dokument, welches nur den getippten Text enthält. Somit ist ein schneller Einstieg in die Syntax für den Nutzer möglich.

```
Hallo Welt!
```

Der Zugriff auf eine Variable erfolgt über das Schreiben des Variablennamens zwischen zwei * Symbole. Die Auswahl des Asterisk als Sonderzeichen erfolgte, da dieses auf der Haupttastatur einer QWERTZ-Tastatur zu finden ist. Dieses Programm liefert die Interpretation des Variablenwertes, übertragen in PDF.

```
*var*
```

Der Zugriff auf Attribute erfolgt über die gängige Punktnotation. Der Punkt wurde als Trennzeichen gewählt, da dieser als Zugriffsoperator in vielen anderen bekannten Programmiersprachen üblich ist. Des Weiteren ist der Punkt auf herkömmlichen QWERTZ-Tastaturen leicht zu erreichen. Allerdings hat sich als Konsequenz aus dieser Wahl ergeben, dass Variablenzugriffe nicht nur das Öffnen einer Umgebung erfordern, sondern auch das Schließen, da der Punkt ein häufiges Satzzeichen ist.

```
*var.attr*
```

Variablennamen dürfen zwar Leerzeichen beinhalten, jedoch werden führende und folgende Leerzeichen ignoriert. Folglich wird der folgende Code ebenso interpretiert, wie der vorhergehende:

```
* var . attr *
```

Aufrufe von Funktionen sind umgeben von spitzen Klammern. Spitze Klammern sind einfach zu erreichen auf einer Standardtastatur (QWERTZ), kommen selten in Fließtext vor, und die Syntax erinnert an die Tags von HTML. Zwischen den Klammern befindet sich ein Variablenname, welcher zu einem aufrufbaren Objekt evaluiert werden muss. Die Rückgabewerte von den Aufrufen werden in einem sogenannten Dokument-Objekt festgehalten.

```
<func 0>
```

Dem Funktionsnamen nachfolgend, können Argumente mit Bindestrichen getrennt übergeben werden, ehe der Aufruf mit einer schließenden Klammer vom Rest des Programmes getrennt wird. Der Bindestrich ist auf herkömmlichen Tastaturen gut zu erreichen. Allerdings kommt dieser recht häufig in Fließtexten vor. Bekannte Programmiersprachen nutzen oft Kommata oder Leerzeichen als Trenner, diese wurden jedoch auf Grund ihrer Häufigkeit in Fließtexten für Tucup als Trennzeichen ausgeschlossen. [1]Des Weiteren wird in der Zeichensetzung zwischen mindestens vier horizontalen Linien zur Interpunktion unterschieden (siehe 'Grundlagen'). Für ein Textdokument von herausragender Qualität muss der Nutzer die Unterschiede kennen und zu nutzen wissen. Die Behandlung des Bindestriches als Sonderzeichen kann also (muss aber nicht) dazu genutzt werden den Nutzer explizit auf diese Unterschiede zu stoßen.

```
<func 1- arg0- arg1 ...>
```

An dieser Stelle sei die Begründung für die konsequente Verwendung von Umgebungen statt Präfixen gegeben. Diese Entscheidung wurde in Hinblick auf die Handhabung von Leerzeichen getroffen. Um die Vorteile zu verdeutlichen, wird im Folgenden eine Variante mit Präfixen betrachtet. TEX beispielsweise verwendet einen Backslash als Präfix für Makros. Um das TEX-Logo darzustellen, wird das Makro `\TeX` verwendet. Um einen weiteren Buchstaben anzuhängen (beispielsweise ein 't'), kann nicht `\TeXt` verwendet werden, da in diesem Fall nach einem Makro mit Namen 'TeXt' gesucht

[1]Die Häufigkeit der Schriftzeichen ist vom Autor geschätzt und nicht empirisch geprüft!

wird. Tatsächlich werden Leerzeichen nach einem Makro ignoriert, also ist die Lösung `\TeX t` zu schreiben. Soll ein Leerzeichen nach einem Makro jedoch nicht ignoriert werden, muss dies Explizit angegeben werden. Auf das TEX Beispiel bezogen heißt dies: `\tex\ t`. Die Notwendigkeit, solche Unterscheidungen treffen zu müssen, lässt sich verhindern, indem Umgebungen statt Präfixen genutzt werden, um auf Makros/Funktionen oder Variablen zuzugreifen.

Kehren wir nun zu den TucUP-Codebeispielen zurück. Ebenso wie beim Variablenzugriff kann ein Attribut eines Objektes als Funktion aufgerufen werden.

```
<obj.methode0 >
<obj.methode1 -arg0>
<obj.methode2 -arg0-arg1 ... >
```

Soll der Rückgabewert eines Aufrufes, zu einem späteren Zeitpunkt der Ausführung zugänglich sein, lässt sich dieser einer Variable zuordnen. Dazu wird der Variablenname oder aber auch der Pfad zu einem Attribut in eckige Klammern unmittelbar vor den Aufruf geschrieben. Beim Interpretieren wird auch für zuweisende Aufrufe der Rückgabewert in das Ausgabedokument geschrieben. Eckige Klammern sind nicht gut zu erreichen und wurden einzig in Anlehnung an das Labeling von LATEX gewählt.

```
[var]<Func -*arg*>
[var . attr]<Func -*arg*>
```

Soll der Rückgabewert einer Funktion in einer einzigen Variablen gespeichert werden, ohne an entsprechender Stelle im Ausgabe-PDF niedergeschrieben zu werden, so ist es möglich, einen ausschließlich zuweisenden Aufruf zu tätigen. Dieser ist wie ein zuweisender Aufruf aufgebaut, verwendet allerdings geschweifte statt eckigen Klammern. Die Wahl der geschweiften Klammern ist nur damit begründet, dass runde Klammern deutlich häufiger in Fließtext vorkommen.

```
{var}<Func -*arg*>
```

Entgegen dem Ziel, moderne und bekannte Programmiermuster zu übernehmen, werden, wie oben dargestellt, Variablen ausschließlich Rückgabewerten von Funktionen zugewiesen. Bei herkömmlichen Programmiersprachen werden Variablen den Werten von Ausdrücken zugewiesen. Um dieses Verhalten zu simulieren, gibt es die standardmäßig eingebaute Identitätsfunktion. Die Identitätsfunktion nimmt ein Argument entgegen und gibt dessen Wert wieder zurück. Allerdings ist es keineswegs so, dass sie das Argument selbst zurückgibt. Denn, wie bereits erläutert, nehmen Funktionen Programme als Argumente entgegen, die Identitätsfunktion jedoch wertet das Programm aus und gibt dessen Wert zurück. Trotzdem wollen wir diese Funktion als 'Identitätsfunktion' bezeichnen und stattdessen die Funktion, welche tatsächlich nur ihr Argument (also ein Programm) zurückgibt, als 'tatsächliche Identitätsfunktion' betiteln. Beide werden bereits vor dem Interpretieren eines Programmes vom Interpreter in den globalen Scope geladen. Dabei wird die Identitätsfunktion in die Variable mit Namen '=' und die tatsächliche Identitätsfunktion in die Variable '==' geladen.

Somit lässt sich nun das herkömmliche Programmiermuster einer Zuweisung simulieren. In folgendem Beispiel wird ein Text der Variable 'text' zugewiesen und die Identitätsfunktion selbst einer Variable 'id'.

```
{text}<=-einfacher Text>
{id}<=-*=*>
```

Ein etwas praktischeres Beispiel zeigt wie ein Kapitel mit variablem Titel erstellt werden kann:

```
{titel}<=-Hallo Welt>
<Chapter-*titel*-
        Wenig Inhalt für dieses Kapitel.
>
```

5.4. Builtins

Objekte, welche bereits vor dem Ausführen des eigentlichen Programms in den Speicher geladen werden, nennen sich Builtins. Neben der Identitätsfunktion und der tatsächlichen Identitätsfunktion gibt es noch weitere. Hier eine Liste von Variablennamen, welche mit Builtins initialisiert werden:

- True: enthält das Boolean-Objekt für den Wahrheitswert 'wahr'.
- False: enthält das Boolean-Objekt für den Wahrheitswert 'falsch'.
- Text: ein aufrufbares Objekt. Konstruktor für ein Text-Objekt; nimmt ein einzelnes Argument entgegen.
- Document: ein aufrufbares Objekt. Konstruktor für ein Dokument-Objekt; nimmt keine Argumente entgegen.
- List: ein aufrufbares Objekt. Konstruktor für ein Listen-Objekt; nimmt beliebig viele Argumente entgegen.
- Int: ein aufrufbares Objekt. Konstruktor für ein Integer-Objekt; nimmt ein einzelnes Argument entgegen.
- throw: ein aufrufbares Objekt. Nimmt ein Text-Objekt als Argument entgegen und wirft einen Fehler mit entsprechender Fehlermeldung.
- None: ist nicht veränderlich und steht stellvertretend für nichts. Wird beispielsweise von ausschließlich zuweisenden Aufrufen zurückgegeben.
- get: ein aufrufbares Objekt. Nimmt mindestens ein Argument entgegen, interpretiert die Argumente als Variablennamen.
- set: ein aufrufbares Objekt. Nimmt mindestens drei Argumente entgegen.
- if: ein aufrufbares Objekt. Nimmt zwei bis drei Argumente entgegen, nur das erste wird evaluiert. Wenn das erste Argument zu True evaluiert, wird das zweite Argument (als Programm) ausgeführt, sonst (wenn gegeben) das dritte Argument.
- Function: ein aufrufbares Objekt. Nimmt mindestens zwei Argumente entgegen. Initialisiert eine neue Funktion.

- __doc__: ein aufrufbares Objekt. Beinhaltet das Root-Dokument, welches, wenn es aufgerufen, wird sich selbst zu einem PDF übersetzt. Es nimmt keine Argumente entgegen.

Weitere Builtins wurden einzig zu dem Zweck integriert, diese Arbeit zu schreiben. Diese lassen sich im Gegensatz zu den obigen Objekten als Markups bezeichnen. All diese Markups sind aufrufbar und nehmen Text als Argumente entgegen.

- Chapter: nimmt Titel und Inhalt als Argumente entgegen.
- Title Page: nimmt Titel, Subtitel, Autor, Thesistyp und Datum als Argumente entgegen.
- Abstract: nimmt eine Zusammenfassung als Argument entgegen.
- Acknowledgements: nimmt eine gerade Anzahl an Argumenten entgegen, dabei sind die Argumente abwechselnd der Titel und der Inhalt eines Anhangs.
- Appendices: nimmt abwechselnd Titel und Anhänge als Argumente entgegen.
- Tucup Logo: nimmt keine Argumente entgegen.
- TeX Logo: nimmt keine Argumente entgegen.
- LaTeX Logo: nimmt keine Argumente entgegen.

Kapitel 6

Implementierung

Die Implementierung von Tucup erfolgte in Haskell. In diesem Kapitel wird auf die konkrete Implementierung der vorgestellten Konzepte aus dem vorherigen Kapitel eingegangen.

6.1. Architektur

Die Interpretation eines Tucup Dokumentes unterteilt sich in vier Schritte: Parsen, Auswerten, Transpilieren und Zwischenformat übersetzen. Der Transpilationsschritt übersetzt das Ergebnis der Auswertung in LaTeX Code. Dieser Code ist das Zwischenformat, welches im finalen Schritt in eine PDF-Datei gewandelt wird. Für den letzten Schritt wird das Programm pdflatex benötigt. Die Funktionalität für die Auswertung und Transpilation wird von einem Interpreter-Modul koordiniert, dabei ist das evaluierende Programm sowie der Pfad, an den der transpilierte LaTeX-Code geschrieben wird, parametrisiert. Die Orchestrierung vom Parser, Interpreter und pdflatex findet in der Main-Funktion des Haskell Programms statt. Die einzelnen Interpretationsschritte werden im Folgenden näher erläutert. Zuvor wird auf die Implementierung von Speichern eingegangen.

6.2. Speicher

Es wurden grundlegend zwei verschiedene Speicherarten implementiert. Zum einen ein Objektspeicher, welcher in der Lage ist, Objekte unter einer Adresse zu persistieren, zum anderen ein Speicher, welcher unter einem Variablennamen entweder eine Adresse oder ein primitives Objekt speichern kann. Die erste Art von Speicher wird nur einmalig verwendet und ist zur Ausführungszeit global verfügbar. Letztere Art von Speicher wird im Sinne von Tucup als Scope bezeichnet, da solche Speicher die Daten eines Objektes halten. Diese Speicher sind stapelbar, so lässt sich ein Scope über einen anderen legen. Dieser Scope-Stack ist ebenfalls global verfügbar. Ein Zugriff auf einen so kombinierten Scope sucht zuerst im obersten Scope und dann, in absteigender Reihenfolge, in den zugrunde liegenden(/umgebenden) Speichern. Dabei ist die maximale Tiefe, zu der bei der Suche vorgedrungen werden soll, parametrisiert. Ebenso lässt sich beim Setzen einer Variable die Tiefe einstellen, in welcher die Variable gesetzt werden soll.

Ein Scope-Stack und ein Objektspeicher repräsentieren zusammen den Zustand eines Programmes. Nur der nächste Ausdruck, welcher ausgewertet werden soll, ist nicht darin enthalten. Da Haskell eine rein funktionale Programmiersprache ist, wird Zustand als solcher durch die Verwendung von Monaden simuliert. Um Zustand zu implementieren, wurden `Control.Monad.Except`[1] und `Control.Monad.State.Lazy`[2] genutzt.

[1] Siehe: https://hackage.haskell.org/package/mtl-2.2.1/docs/Control-Monad-Except.html
[2] Siehe: https://hackage.haskell.org/package/mtl-1.1.0.2/docs/Control-Monad-State-Lazy.html

6.3. Objekte

Objekte werden bei der Implementierung in zwei Typen unterteilt: Standardobjekte und primitive Objekte. Primitive Objekte können in Scopes gehalten werden, jedoch nicht im Objektspeicher. Wird versucht, ein Attribut eines primitiven Objektes zu setzen, führt dies immer zu einem Fehler. Standardobjekte hingegen können ausschließlich im Objektspeicher gehalten werden, jedoch nicht in Scopes. Scopes können nur Referenzen auf Standardobjekte halten. Um die Handhabung von Standardobjekten, Referenzen, Variablennamen und primitiven Objekten zu vereinfachen, werden diese von einem Datentypen zusammengefasst. Dieser Datentyp fasst also alle objektartigen Typen zusammen. Jeder der zusammengefassten Typen implementiert eine Typklasse, welche Funktionalitäten zum Lesen und Setzen von Attributen sowie zum Aufrufen bedingt. Variablennamen und Referenzen delegieren diese Funktionalität an das referenzierte Objekt weiter. Primitive Objekte rufen stets einen Fehler beim Nutzen der Funktionen hervor. Weitere Funktionen der Klasse sind für interne Berechnungen notwendig. Wenn nicht explizit erwähnt, ist im Folgenden mit einem Objekt immer etwas objektartiges gemeint. Standardobjekte können individuelle Implementierungen für das Lesen und Setzen von Attributen haben. Dadurch ist es beispielsweise möglich, dass beim Zugriff auf eine Liste auch Integer als Schlüssel für einen Attributwert entgegengenommen werden. Ebenso ist die Implementierung des Aufrufs eines Objektes individuell. Um Standardobjekten zu ermöglichen, intern mit Haskell-Datentypen zu arbeiten, hat der Standardobjekt-Datentyp ein Feld für solche vorgesehen. Ebenso hat jedes Standardobjekt ein Feld für seinen Scope. Dieser wird standardmäßig für das Lesen und Setzen von Attributen verwendet und ist für den Endnutzer zugänglich.

6.4. Parsen

Der Parser wurde mit Hilfe des Parser-Generators Happy programmiert. Dabei erzeugt Happy aus einem Code mit bestimmter Form Haskell Code zum Parsen einer Syntax. Die geparste Syntax ist bereits im Kapitel 'Entwurf von Tucup' vorgestellt worden. Die Ziel-Datenstruktur, zu welcher ein gültiges Tucup Dokument geparst wird, ist ein spezieller abstrakter Syntax-Baum (AST). Knoten eines solchen AST sind dabei entweder Programme oder Ausdrücke. Der Wurzelknoten ist immer ein Programm. Ein Programm besteht dabei aus einer Sequenz von Ausdrücken, und ein Ausdruck seinerseits kann Programme oder Ausdrücke halten. Etwa, wenn der Ausdruck ein Aufruf ist, wie bereits erläutert. Die Spezialität dieses Baumes besteht darin, dass dieses Objekt im Speicher des Interpreters gehalten wird.

6.5. Auswerten

Die Auswertung eines Ausdrucks oder die Ausführung eines Programms entspricht dessen Aufruf. Bevor jedoch ein Programm vom Interpretermodul ausgewertet wird, initialisiert dieses einen Zustand, in welchem das Programm aufgerufen wird. Bei der Initialisierung werden Builtin-Objekte in den Objektspeicher geladen und entsprechende Variablen in dem grundlegenden globalen Scope gesetzt.

Ein Programm-Objekt wird ausgewertet, indem die Ausdrücke der Sequenz nacheinander ausgewertet werden. Ein Textausdruck wird schlicht zu einem primitiven String-Objekt ausgewertet. Ein Zugriffsausdruck bedient sich der eingebauten get-Funktion, um den Scope-Stack nach den angefragten Variablennamen zu durchsuchen und das zugehörige Objekt ausfindig zu machen. Weitere Zugriffe in der Kette von Variablennamen werden dann im Scope des Objektes durchgeführt. Ein Aufruf evaluiert zuerst einen Zugriff, um das aufzurufende Objekt zu finden. Daraufhin wird der Objekt-Scope auf

den Scope-Stack gelegt, und je nach Bedarf werden einzelne, keine oder alle Argumente ausgewertet oder als jene Objekte gehalten, als welche sie übergeben wurden. Die Auswertung der Argumente findet außerhalb des Scopes der Funktion statt. Die Speicher der als Argumente übergebenen Objekte werden für deren Auswertung nicht auf den Scope-Stack abgelegt. Nach diesem Schritt wird der eigentliche Aufruf mit den (eventuell) ausgewerteten Argumenten durchgeführt, der oberste Scope vom Stack wieder entfernt und das Ergebnis zurückgeliefert. Zuweisende und ausschließlich zuweisende Aufrufe arbeiten nach dem gleichen Prinzip, machen sich jedoch die eingebaute set-Funktion zu nutze, um den Rückgabewert an eine Variable zu binden. Ausschließlich zuweisende Aufrufe geben zudem nichts zurück außer das primitive None-Objekt.

6.6. Builtin-Objekte

Die Funktionsweise einiger Builtins ist nicht sofort ersichtlich und Bedarf einer Erläuterung. Auf diese soll hier eingegangen werden.

6.6.1. Set

Die set-Funktion nimmt mindestens drei Argumente. Das erste Argument ist dabei die Setztiefe. Diese spezifiziert, in welchem Scope die Funktion operieren soll. Wird diese als Null spezifiziert, so operiert die Funktion in ihrem eigenen Scope, da dieser während des Aufrufs oben auf dem Scope-Stack liegt. Entsprechend wird im Scope-Stack um die spezifizierte Anzahl tiefer gegangen, wenn das erste Argument größer als Null ist. Ist die Zahl kleiner als Null, wird auf dem globalen Scope gearbeitet. Das letzte Argument ist der Wert, welcher gesetzt werden soll. Die restlichen Argumente bilden den Pfad zur Variable, welche gesetzt werden soll. Alle Argumente werden evaluiert.

6.6.2. Funktionskonstruktor

Um in Tucup eine Funktion zu definieren, wird das Funktionskonstruktor-Objekt genutzt. Dieses nimmt mindestens zwei Argumente entgegen. Das erste Argument wird dabei als Variablenname genutzt. In entsprechender Variable wird nach dem Aufruf das neue Funktionsobjekt gehalten. Um die Variable nicht im Scope des Funktionskonstruktors zu setzen, wird die set-Funktion mit entsprechender Setztiefe genutzt. Das letzte Argument ist der Funktionskörper und wird nicht evaluiert. Dazwischen liegende Argumente werden als Variablennamen behandelt und dienen als Parameter. Das zurückgegebene Objekt lässt sich aufrufen und nimmt genauso viele Argumente entgegen wie dem Funktionskonstruktor als Parameter übergeben wurden. Beim Aufrufen des neuen Funktionsobjektes werden die Argumente evaluiert. Dann werden diese im Scope des Funktionskörpers, welcher ein Programm-Objekt ist, als Attribute gesetzt. Die Namen der Attribute korrespondieren dabei mit den Parametern, welche bei der Konstruktion übergeben wurden. Dadurch stehen die Parameter als Variablen beim Aufruf der Funktion zur Verfügung.

6.6.3. Throw und Error

Throw ist ein aufrufbares Objekt, um kontrolliert Fehler auszugeben. Dieses nimmt ein Argument entgegen, welches als Fehlermeldung genutzt wird. Dabei wird lediglich ein Error-Objekt erzeugt und dieses aufgerufen. Beim Aufruf eines Error-Objektes wird der Zustand der Except-Monade auf einen Fehlerzustand gesetzt. Neben der Fehlermeldung wird auch der Scope-Stack ausgegeben, um

zu verdeutlichen, an welcher Stelle der Error aufgerufen wurde. Die Ausgabe des Fehlers erfolgt über den Interpreter.

6.7. Transpilieren und Zwischenformat Übersetzen

Für das Zwischenformat wurde ein extra Datentyp in Haskell angelegt, welcher ein LaTeX Dokument repräsentieren soll. Dieser Datentyp wird von einem Builtin Objekt zur Laufzeit zur Verfügung gestellt. Der Datentyp bietet Funktionalitäten, um Text zu schreiben. Jedes Objekt muss Funktionalität zum Transpilieren bieten, wobei die Möglichkeit gegeben ist, dass bei der Nutzung ein Error-Objekt aufgerufen wird. Für jene Objekte, welche sich tatsächlich transpilieren lassen, ist in der Transpile-Funktion festgelegt, wie das Ausgabedokument manipuliert werden soll. Dazu wird Gebrauch von dem Dokument-Objekt gemacht.

Beim Aufruf eines Programms wird dessen Sequenz von Ausdrücken ausgewertet, und die Rückgabewerte werden in einer Liste zwischengespeichert. Die Rückgabe eines Programms ist entweder eben diese Liste oder, wenn die Liste nur ein Element hat, nur dieses Element. Nachdem das Hauptprogramm zu einer Liste ausgewertet wurde, wird diese Liste transpiliert, indem für jedes Element in der Liste die Transpile-Funktion aufgerufen wird.

Das nun gefüllte Dokument-Objekt bietet eine Methode, die seinen Inhalt als Zeichenkette zurückgibt. Diese Zeichenkette wird in eine TeX-Datei geschrieben und mit Hilfe vom Konsolen-Programm pdflatex in eine PDF-Datei übersetzt.

Kapitel 7

Fazit und Ausblick

Es gibt einige Bestrebungen, welche ähnliche Ziele wie Tucup verfolgen. Auch wenn keines der analysierten Systeme eine vollständige Übereinstimmung aufweist, ist ConTEXt ein System, welches sehr dicht an die Vision von Tucup heranreicht. Dennoch werden Weiterentwicklungen von ConTEXt die Anforderungen, welche Tucup zu erfüllen versucht, nie erreichen, da es eine Erweiterung von TEX bleiben wird.

Mittelfristig und langfristig gibt es einige Ziele, welche bei der Implementierung von Tucup weiter verfolgt werden:

- Wegen der sequenziellen Interpretation von Tucup ist eine Referenz auf ein Objekt erst ab dem Zeitpunkt zu nutzen, da das Objekt konstruiert wird. Dadurch ist eine Referenzierung auf etwa ein späteres Kapitel nicht ohne weiteres möglich. Um dieses Problem zu lösen, sollte eine Transformation von der sequenziellen Interpretierung hin zu einer Kompilation von Tucup stattfinden.
- Die Implementierung von Typen und weiteren Programmierkonstrukten.
- Die Implementierung von Layout Dateien.
- Das Konzipieren und Implementieren von Konfigurationsdateien.
- Das Konzept der Trennung von Daten und Inhalt soll in die Architektur eingeordnet und Implementiert werden.
- Da TEX durch seine räumliche und zeitliche Konsistenz eine stabile Grundlage als Zwischenformat bietet, ist TEX mittelfristig als Zwischenformat beizubehalten. Langfristig jedoch ist die Spezifikation oder Wahl eines Formates notwendig, welches ein Zwischenformat überflüssig macht, indem das Format als Ausgabeformat fungiert. Dies soll einer höheren Responsivität bei einer Interpretation dienen.
- Finales Ziel ist die Implementierung eines WYSIWYG-Editors für Tucup-Dateien.

Anhang A

TypeScript pdfmake Beispiel

```
import * as PdfPrinter from "pdfmake";
import * as fs from "fs";

interface DocumentDefinition {
    content : Array<DocumentContent>
}

type DocumentContent  = string | Array<DocumentContent> | {
    text : string
    style : {
        fontsize: number,
        bold : boolean,
    }
}

const fonts = {
    Roboto: {
        normal: 'path/to/Roboto-Regular.ttf',
        bold: 'path/to/Roboto-Medium.ttf',
        italics: 'path/to/RobotoRoboto-Italic.ttf',
        bolditalics: 'path/to/RobotoRoboto-MediumItalic.ttf'
    }
};
const printer = new PdfPrinter(fonts);

function getRandomIntegerUpTo(limit: number): number{
    return Math.floor(Math.random() * (limit+1));
}

function makeExerciseText(): DocumentContent{
    const a = getRandomIntegerUpTo(1000).toString();
    const b = getRandomIntegerUpTo(1000).toString();
    return a.concat(" + ", b, " =");
}

function makeExerciseHeading(exerciseNumber: number): DocumentContent{
    return {
        text: "Exercise ".concat(exerciseNumber.toString()),
```

```
        style : {
          fontsize: 16,
          bold: true,
        },
    };
}

function makeExercise(exerciseNumber: number): DocumentContent{
    return [
        makeExerciseHeading(exerciseNumber),
        makeExerciseText(),
        "\n\n",
    ];
}


function makeDocDefinition(): DocumentDefinition {
    const content = [];
    for (let i = 1; i <= 10; i++) {
        const exercise = makeExercise(i);
        content.push(exercise);
    }
    return {
        content: content,
    }
}

const docDefinition : DocumentDefinition = makeDocDefinition();

const pdfDoc = printer.createPdfKitDocument(docDefinition);
pdfDoc.pipe(fs.createWriteStream('pdfmakeExercises.pdf'));
pdfDoc.end();
```

Anhang B

Tucup-Code für Zufällige Additionsaufgaben

```
// Designer Seite
{0}<Int-0>
{1000}<Int-1000>
<Function-Zufällige Additionsaufgabe-titel-
        [ueberschrift]<Text-*titel*>
        {ueberschrift.bold}<=-*True*>


        <random int-*0*-*1000*> + <random int-*0*-*1000*> =

>

// Autoren Seite
<Zufällige Additionsaufgabe-Exercise 1>

<Zufällige Additionsaufgabe-Exercise 2>

<Zufällige Additionsaufgabe-Exercise 3>

<Zufällige Additionsaufgabe-Exercise 4>

<Zufällige Additionsaufgabe-Exercise 5>

<Zufällige Additionsaufgabe-Exercise 6>

<Zufällige Additionsaufgabe-Exercise 7>

<Zufällige Additionsaufgabe-Exercise 8>

<Zufällige Additionsaufgabe-Exercise 9>

<Zufällige Additionsaufgabe-Exercise 10>
```

Anhang C

Zufällige Additionsaufgaben erstellt mit Tucup

Exercise 1
1000 + 411 =
Exercise 2
43 + 110 =
Exercise 3
677 + 32 =
Exercise 4
67 + 194 =
Exercise 5
978 + 599 =
Exercise 6
79 + 592 =
Exercise 7
440 + 64 =
Exercise 8
857 + 474 =
Exercise 9
235 + 438 =
Exercise 10
450 + 53 =

Literaturverzeichnis

CTAN-Team. Was ist ctan? https://www.ctan.org/ctan. Zugriff: 12.09.2021.

DocBook. What is docbook? https://docbook.org/whatis. Zugriff: 20.09.2021.

Jelica Knauff, Markus und Nejasmic. An efficiency comparison of document preparation systems used in academic research and development. *PloS one*, 9(12), 2014.

Duane Knuth, Donald Ervin und Bibby. *The texbook*. Addison-Wesley Reading, 1984.

Robert C. Martin. The future of programming. https://www.youtube.com/watch?v=ecIWPzGEbFc.

Robert Murrish. Latex is more powerful than you think - computing the fibonacci numbers and turing completeness. https://www.overleaf.com/learn/latex/Articles/LaTeX_is_More_Powerful_than_you_Think_-_Computing_the_Fibonacci_Numbers_and_Turing_Completeness. Zugriff: 20.09.2021.

Bartek Pampuch. Github pdfmake repository. https://github.com/bpampuch/pdfmake/release. Zugriff: 12.09.2021.

Bernd Raichle. Sorting in tex's mouth. *Nederlandstalige TEX Gebruikersgroep*, page 163, 1995.

Stackoverflow. 2020 developer survey. https://insights.stackoverflow.com//survey//2020. Zugriff: 12.09.2021.