Prototyp eines S-Expression-basierten Rahmenwerks für sprachübergreifende Metaprogrammierung

Benjamin Teuber

Hamburg
27. November 2010

Diplomarbeit am
Arbeitsbereich „Theoretische Grundlagen der Informatik“,
Department Informatik,
Universität Hamburg

Erstbetreuer: Dr. Daniel Moldt
Zweitbetreuer: Prof. Dr. Leonie Dreschler-Fischer

Zusammenfassung

Template-gestützte Code-Generierung ist in der Informatik allgegenwärtig und tritt in den verschiedensten Formen auf. Entsprechend groß ist die Vielfalt verwendeter Werkzeuge – seien es der C-Präprozessor, PHP oder auch XSL-Transformationen. Die meisten bestehenden Technologien sind allerdings aufgrund der verwendeten Syntax und Datenstrukturen recht fehleranfällig und umständlich in der Benutzung. Einen weitgehend vergessenen, aber äußerst eleganten und praktischen Ansatz zur generativen Metaprogrammierung bietet dagegen die Lisp-Sprachfamilie, welche mit S-Expressions als universeller, einfacher Datenstruktur arbeitet sowie mit Makros, die als „Lightweight-Compiler“ fungieren.

Diese Diplomarbeit beschäftigt sich mit der Frage, wie der Lisp-Ansatz weiter verallgemeinert werden kann, um beliebige Zielsprachen, verschiedene Modelltypisierungsgrade sowie eine EBNF-Grammatiken ähnliche Komposition von Makros zu ermöglichen. Dafür wird mit MagicL ein prototypisches Rahmenwerk in Haskell implementiert. Mit Hilfe einer kategorientheoretisch motivierten Architektur auf der Basis von Arrows lassen sich Parser, Generatoren und andere Compiler erstellen und miteinander kombinieren, wobei neben S-Expressions auch typisierte Objekte und Zeichenketten als Ein- und Ausgaben unterstützt werden. Darauf aufbauend wird eine ebenfalls in S-Expressions notierte DSL für die Compilerdefinition konstruiert, die mit der Haskell-DSL eine funktionale Programmiersprache enthält. Für die einfache Verarbeitung von S-Expressions stellt die Compiler-DSL einen Quasiquote-Operator sowie ein gegenüber Lisp verallgemeinertes Makrosystem bereit.

Abstract

Template based code generation is ubiquitous throughout computer science. It is used in many different forms, for example the C preprocessor, PHP or XSL Transformations – just to mention a few. However, most existing tools are quite cumbersome to use and error prone due to employed syntax and data structures. An alternative, yet almost forgotten approach is taken by the Lisp family of languages. There, S-Expressions are used as a simple and universal data structure for code representation and are being processed by macros that can be regarded as “lightweight compilers”.

This thesis deals with the idea of generalizing the Lisp approach in order to support arbitrary output languages and different model types as well as composing macros in a way similar to EBNF grammars. As a proof of concept, a prototypical framework called MagicL is implemented in Haskell, using an architecture inspired by category theory. This allows to create and combine parsers, generators as well as other kinds of compilers by the means of arrows that can have S-Expressions as well as strings and typed objects as input and output types. On top of this, a DSL for defining compilers is constructed. The compiler DSL includes a Haskell DSL, which is a complete functional programming language, as well as a quasiquote operator and a generalized macro system for easy processing of S-Expressions.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ASCII American Standard Code for Information Interchange

CSV Comma Separated Values

DSL Domain Specific Language

DSSSL Document Style Semantics and Specification Language

EBNF Erweiterte Backus-Naur-Form

GUI Graphical User Interface

GHC Glasgow Haskell Compiler

HTML Hypertext Markup Language

JVM Java Virtual Machine

MDA Model Driven Architecture

PDF Portable Document Format

PHP PHP: Hypertext Preprocessor

REPL Read Eval Print Loop

SQL Structured Query Language

UML Unified Modelling Language

XML Extensible Markup Language

XML-RPC XML Remote Procedure Call

XPath XML Path Language

XSL Extensible Stylesheet Language

XSL-FO XSL - Formatting Objects

XSLT XSL Transformation

W3C World Wide Web Consortium

WWW World Wide Web

1 Einleitung

Unter Metaprogrammierung versteht man das programmatische Manipulieren von Programmen - anders gesagt: „Metaprogramming is writing code that writes code“ [37, S.16]. Man unterscheidet zwischen statischer und dynamischer Metaprogrammierung [37, S.17]. Dynamische Metaprogrammierung behandelt die Manipulation von Programmen während der Laufzeit, weshalb man auch von Laufzeit-Metaprogrammierung spricht. Dies funktioniert nur in Sprachen, die diese Möglichkeit explizit vorsehen, wie Ruby [37] oder auch Java [17].

Diese Arbeit beschäftigt sich stattdessen mit statischer Metaprogrammierung, in der es um das Erzeugen von Quelltext geht, der anschließend an beliebige Compiler oder Interpreter gereicht werden kann, weshalb auch die Bezeichnungen Kompilationszeit-Metaprogrammierung und generative (Meta-)Programmierung [32, S.17] benutzt werden. Diese Art der Metaprogrammierung ist in der Informatik allgegenwärtig, beispielsweise kann jeder Compiler als Metaprogramm gesehen werden.

Die auf Code-Generierung basierende „modellgetriebene Softwareentwicklung“ erfreut sich immer größerer Popularität: Während UML [27] bald sein 15-jähriges Bestehen[1] feiert, vermarktet IBM seit Jahren das Produkt *Rational Application Developer* [19], das, wie auch das freie *Eclipse Modelling Framework* sowie *Enterprise Architect*, Eclipse um modellbasierte Code-Generierung bereichert. Seit neuestem zeigt auch Microsoft Interesse und verspricht mit *Oslo* [18] einen „Mainstream-Ansatz für Modellierung“. Generativer Programmierung fällt auch im WWW eine immer größer werdende Rolle zu: Ob *Ruby on Rails* oder *Google Web Toolkit* – praktisch alle Web Frameworks generieren zumindest ihre HTML-Dateien oder SQL-Abfragen für den Datenbankzugriff.

Code-Generierung aus Modellen[2] bietet viele Vorteile: Zunächst gibt es dadurch einen höheren Abstraktionsgrad, so dass Modellarchitekten mit Domain Specific Languages (DSLs) arbeiten können und so die Ziel-Programmiersprache nicht kennen müssen. Auch ist es möglich, durch Austauschen des Generator-Backends die Zielsprache zu wechseln, ohne Modelle anpassen zu müssen. Alternativ sind durch Metaprogrammierung integrierte DSLs oder Spracherweiterungen möglich, die dann beliebig mit der Backendsprache vermischt werden können. Die Möglichkeit, eine Programmiersprache zu modifizieren, liegt somit nicht mehr ausschließlich beim Hersteller. Metaprogrammierung erhöht die Produktivität, da sich Algorithmen in einer speziell auf die Problemdomäne zugeschnittenen Sprache mit weniger Aufwand formulieren lassen. Sie kann ebenfalls die Qualität verbessern, da ein korrekter Generator auch aus neuen Modellen, sofern diese verifiziert

[1] Der Standarisierungsprozess von UML begann bereits 1994, als Gary Booch und Jim Rumbaugh mit der Vereinheitlichung ihrer Modellierungstechniken anfingen [27, S. 17].

[2] Auch Programmcode wird hier als Modell angesehen – mit anderen Worten: „Code is data and data is code.“

sind, immer fehlerfreien Code erzeugt. Daneben weist generierter Code eine höhere Konsistenz auf, da generierte Strukturen, Funktionsnamen etc. immer einheitlich sind. Dies erleichtert die Benutzung von generierten Schnittstellen enorm.

Mit Metaprogrammierung sind allerdings auch Nachteile verbunden: Die Komplexität des Software-Systems wird durch die zusätzliche Ebene erhöht und es entstehen neue mögliche Fehlerquellen. Auch die Suche nach Fehlern in den Modellen kann erschwert werden, wenn die Probleme sich erst im generierten Code manifestieren – hier wird oft eine manuelle Rückverfolgung benötigt um festzustellen, welcher Code aus welchem Modell erzeugt wurde. Darüber hinaus erfordern neue Modellierungssprachen immer eine Einarbeitung und müssen dokumentiert werden. Auch sollte es nicht zu viele konkurrierende DSLs für eine Domäne geben, so dass einheitliche Standards verloren gehen. Beispielsweise kommt es bei Lisp-Dialekten [41] aufgrund der Leichtigkeit, mit der die Sprache erweitert werden kann, durchaus vor, dass jeder Programmierer in seiner eigenen Sprache programmiert, so dass die Wartung von fremden Code sehr aufwändig wird. Um dies zu vermeiden, ist der rege Austausch von Sprachen und Spracherweiterungen wichtig, damit die Community schnell zu gemeinsamen Standards kommt und nicht jeder Programmierer „das Rad neu erfindet“.

Trotz dieser möglichen Probleme überwiegen in vielen Fällen die Vorteile, so dass die Bedeutung von generativer Programmierung sowie die Vielfalt an Werkzeugen immer weiter zunehmen. Diese Arbeit beschäftigt sich ganz allgemein mit der Frage, wie Metaprogrammierung besser, einfacher und einheitlicher gestaltet werden kann als bisher.

1.1 Motivation

Die Entwicklung eines Code-Generators ist gewöhnlich mit großem Aufwand verbunden. So muss zunächst eine Eingabesprache festgelegt werden, für die dann eine typisierte Modellrepräsentation sowie ein Parser zur Überführung in diese benötigt werden. Für den Parser wird eine eigene Beschreibungssprache verwendet, die vom Programmierer vorher erlernt werden muss und eventuell eine eingeschränkte Mächtigkeit besitzt, so dass eine programmatische Nachbearbeitung der Modelle nötig werden kann. Im Zuge der Verarbeitung, die in einer allgemeinen Programmiersprache geschieht, werden eventuell noch weitere Modelle benötigt. Die Quelltexterzeugung selbst erfolgt dann meist mittels String-basierter Templates, welche aufgrund der fehlenden Ausgabetypisierung oft zu Syntaxfehlern führen und so aufwändiges Debugging benötigen. Auch die Template-Sprache ist einzeln zu erlernen und eventuell ebenfalls in ihren Möglichkeiten eingeschränkt, so dass in einigen Fällen nochmals eine Vorverarbeitung der Daten erfolgen muss.

Andererseits verfolgen Lisp-Programmierer bereits seit ca. 40 Jahren [42] einen strukturellen Ansatz zur Metaprogrammierung, der den mit Spracherweiterungen verbundenen Aufwand minimiert und Lisp Bezeichnungen wie „The programmable programming language“ [15] beschert hat. Hier werden Ein- und Ausgabe-Code ausschließlich durch S-Expressions (siehe Abschnitt 2.4) in einer untypisierten Baumstruktur repräsentiert bei der Zeichenketten weder eingelesen noch erzeugt werden müssen, wodurch ein großer

Teil des Generators bereits wegfällt, denn das Verarbeiten und Erzeugen von Bäumen ist simpler und weniger fehlerbehaftet. S-Expressions fungieren auch direkt als Modelle, so dass weitere Datenstrukturen für die Verarbeitung nur noch selten benötigt werden. Lisp-Makros (siehe Abschnitt 3.8) stellen kleine Compiler-Plugins dar, die jeweils die Übersetzung eines einzelnen Sprachkonstrukts beschreiben und so einen inkrementellen, baukastenartigen Ansatz der Compiler-Entwicklung unterstützen. Da Makros selbst Lisp-Funktionen sind, ist die einzige verwendete Metasprache mit der generierten Sprache identisch, was die Einarbeitung erleichtert und darüber hinaus die Möglichkeit eröffnet, Meta-Code selbst durch Generierung zu erzeugen.

Leider ist das System ausschließlich auf die Erzeugung von Lisp-Code ausgerichtet und wird normalerweise nicht für andere Zielsprachen benutzt – eine der wenigen Ausnahmen ist Parenscript [40], ein Werkzeug zur Erzeugung von JavaScript-Code aus Common Lisp. Generell wird die Generierung anderer Sprachen von Lisp-Systemen selbst aber schlecht unterstützt, weshalb die Bereitstellung derartiger Bibliotheken mit unverhältnismäßig hohem Aufwand verbunden ist. Lisp-Dialekte werden heutzutage nur noch von einer kleinen Community benutzt, was teils an veraltetem Sprachdesign, hauptsächlich aber an der geringen Anzahl verfügbarer Bibliotheken im Vergleich zu Mainstream-Sprachen liegt. Während viele andere Lisp-Konzepte wie Garbage Collection oder der Read Eval Print Loop (REPL), welcher eine Lisp-Kommandozeile zum interaktiven Programmieren und Testen darstellt, bereits von modernen Sprachen nachgeahmt werden, ist der Ansatz einer S-Expression-basierten Metaprogrammierung mittels Makros[3] nach wie vor einzigartig. Es wäre also wünschenswert, diesen verallgemeinern und auf beliebige Backendsprachen übertragen zu können.

1.2 Fragestellung und Zielsetzung

Die grundlegende Fragestellung lautet: Ist es möglich, den Lisp-Ansatz inkrementeller Metaprogrammierung auf S-Expression-Basis derart zu verallgemeinern, dass Code beliebiger Zielsprachen komfortabel generiert werden kann? Der Rahmen dieser Arbeit soll allerdings deutlich weiter gefasst sein als die bloße Anwendung von Lisp-Makros auf die Erzeugung verschiedener Sprachen. Diese könnte eventuell bereits durch die Implementation einer geeigneten Lisp-Bibliothek erreicht werden. Vielmehr sollen sowohl Lisp als auch andere Systeme zunächst auf Stärken und Schwächen untersucht und anschließend eine eigene Architektur entworfen werden, die eine hohe Flexibilität sowie möglichst viele Vorteile bestehender Werkzeuge in sich vereinen soll. Diese soll dann in einer geeigneten Programmiersprache umgesetzt und so ein prototypisches Framework bereitgestellt werden. Anschließend soll eine möglichst praktische, selbst in S-Expressions notierte DSL für die Beschreibung von S-Expression-Compilern kreiert werden.

[3] Der C-Präprozessor bietet ebenfalls Makros an (siehe Abschnitt 3.6), welche allerdings weit weniger mächtig sind.

1.3 Aufbau dieser Arbeit

Zunächst stellt Abschnitt 1.4 mit einer DSL für Bildschirmpräsentationen ein Anwendungsbeispiel für generative Programmierung vor, das im weiteren Verlauf dieser Arbeit immer wieder aufgegriffen wird.

Kapitel 2 erläutert verschiedene Möglichkeiten der Repräsentation von Quelltext und anderen Modellen und die damit verbundenen Vor- und Nachteile. Hier werden insbesondere S-Expressions vorgestellt und mit XML und anderen Notationen verglichen.

Fast alle Code-Generierungswerkzeuge benutzen Templates für die Code-Erzeugung – wenn auch auf sehr verschiedene Arten und Weisen. Kapitel 3 beschreibt Templates zunächst anhand der Begriffe Quasiquote und Unquote, welche aus der Lisp-Welt stammen. Anschließend wird eine Auswahl Template-basierter Werkzeuge – unter anderem auch Lisp-Makros – vorgestellt und in Hinblick auf vorher erarbeitete Kriterien evaluiert.

Aus den vorangegangenen Untersuchungen über Modelle und Werkzeuge werden in Kapitel 4 Anforderungen für das universelle, im Rahmen dieser Arbeit entworfene Rahmenwerk MagicL abgeleitet. Wesentlich ist hier, dass die Definition beliebiger Compiler zwischen den Modelltypen Zeichenkette, S-Expression und Objekt – mit einem Fokus auf S-Expressions – auf eine möglichst einfache und flexible Weise unterstützt wird, weshalb neben Lisp-Makros verschiedene Aspekte der im vorhergehenden Kapitel vorgestellten Werkzeuge zusammen mit einigen neuen Ideen kombiniert werden sollen.

Um die geforderte Flexibilität erreichen zu können, basiert die gesamte Architektur von MagicL auf Konzepten der Kategorientheorie, einem sehr abstrakten Zweig der Mathematik. Neben vielfältigen anderen Verwendungsmöglichkeiten können mit Kategorien beliebige Arten von Prozessketten derart beschrieben werden, dass Verarbeitungseinheiten über Operatoren nach dem Baukastenprinzip kombinierbar sind. Kapitel 5 führt deshalb in die Kategorientheorie ein und definiert später benötigte Begriffe wie zum Beispiel Funktoren.

Die Programmiersprache Haskell stellt mit Arrows und Monaden programmatische Umsetzungen kategorientheoretischer Konzepte zur Verfügung, mit deren Hilfe es unter anderem gelingt, Nebeneffekte sauber isoliert in die ansonsten pur funktionale Sprache zu integrieren. MagicL ist in Haskell mittels Arrows implementiert. Daher erläutert Kapitel 6 zunächst die Sprachgrundlagen und widmet sich anschließend den kategorientheoretischen Typklassen und Funktionen.

Ein grundlegender Überblick über die Architektur von MagicL wird in Kapitel 7 vermittelt. Hier werden die verschiedenen modellspezifischen Bibliotheken und Hilfsmittel präsentiert, die MagicL für die Konstruktion von Compilern bereitstellt. Zudem werden die Haskell-DSL und die darauf aufbauende Compiler-DSL vorgestellt, die dem Metaprogrammierer in MagicL zur Verfügung stehen.

Kapitel 8 zeigt, wie sich durch die Kombination einfacher Funktoren eine Parser-Kategorie konstruieren lässt. Die Haskell-Umsetzung dieser Kategorie ergibt eine elegante Art der Beschreibung von Parsern, die nicht nur für Zeichenketten verwendet werden können. Darauf aufbauend wird eine Parser-Bibliothek mit nützlichen Hilfsfunktionen zur Erzeugung und Kombination von Parsern bereitgestellt.

Neben der Parser-Bibliothek enthält MagicL eine Reihe weiterer Basiskomponenten,

welche in Kapitel 9 vorgestellt werden. Eine Generator-Bibliothek erlaubt die funktionale Erzeugung von Quelltext auf eine Weise, die weniger anfällig für Syntaxfehler ist als ein direktes Templating auf String-Ebene. S-Expressions werden in MagicL ebenfalls durch Parser verarbeitet, welche als verallgemeinerte Makros gesehen werden. Eine S-Expression-Bibliothek stellt eine Reihe von Hilfsfunktionen für S-Expression-Parsing bereit und liefert darüber hinaus einen Parser und einen Generator für die Serialisierung von S-Expressions in Zeichenketten. Außerdem wird die allgemeine, ebenfalls Arrow-basierte Compiler-Schnittstelle vorgestellt, mit der verschiedene Parser sowie funktionale und imperative Compiler angesprochen werden können. Damit ist es zudem möglich, Haskell mittels Typinferenz selbstständig den passenden Compiler für eine Aufgabe auswählen zu lassen.

Mit Hilfe der beschriebenen Werkzeuge wird in Kapitel 10 die Haskell-DSL implementiert, welche eine auf S-Expressions übertragene Version von Haskell und damit eine vollständige funktionale Programmiersprache darstellt und eine rein S-Expression-basierte Generierung von Haskell-Quelltext ermöglicht.

Kapitel 11 stellt schließlich die DSL vor, in der MagicL-Benutzer eigene Compiler definieren können. Die Compiler-DSL ist eine Erweiterung der Haskell-DSL und enthält einen Quasiquote-Operator für S-Expressions sowie ein Lisp ähnliches Makrosystem. Der Compiler-Compiler, der von der Compiler-DSL in die Haskell-DSL übersetzt, wird selbst in der Compiler-DSL definiert und ist somit metazirkulär.

Kapitel 12 diskutiert das entstandene Framework in Hinblick auf die vorherigen Anforderungen und vergleicht MagicL mit anderen Systemen zur Metaprogrammierung. Schließlich wird die gesamte Arbeit in Kapitel 13 zusammengefasst und ein Ausblick auf mögliche Fortsetzungen dieser Arbeit gegeben.

1.4 Beispieldomäne

Für die einfachere Diskussion verschiedener Werkzeuge und Konzepte versucht diese Arbeit, möglichst eine einheitliche Domäne für Anwendungsbeispiele zu verwenden. Dabei geht es um eine fiktive, zu erstellende DSL für Präsentationen – genannt Slide – die Konzepte wie Folientitel, Aufzählungen, Bilder etc. in einer kurzen Notation bereitstellt. Slide-Code wird vom Modellierer – hier dem Ersteller einer Präsentation – geschrieben und vom Slide-Compiler in Latex-Code [12] mit der Beamer-Klasse [43] übersetzt (siehe Abb. 1.1), aus der anschließend eine PDF-Datei erzeugt werden kann.

Die Aufgabe eines Werkzeugs zur Code-Generierung besteht nun darin, den Metaprogrammierer bei der Entwicklung des Slide-Compilers soweit wie möglich zu unterstützen.

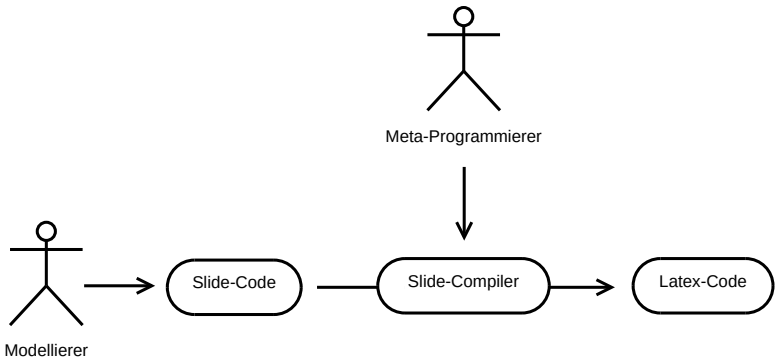


Abbildung 1.1: Der Slide-Compiler

2 Repräsentation von Modellen

In bestehenden Code-Generierungswerkzeugen werden Modelle für Ein- und Ausgaben auf eine von drei Arten repräsentiert: Als rohe Zeichenkette, als typisiertes Objekt oder durch eine ungetypte Baumrepräsentation. Beispiele für letztere sind unter anderem S-Expressions, XML sowie JSON. Dieses Kapitel erarbeitet zunächst Vergleichskriterien für Repräsentationsformen. Anschließend werden alle diese Ansätze sowie die graphische Repräsentationssprache UML vorgestellt und anhand der Kriterien bewertet.

2.1 Vergleichskriterien

Als Vergleichskriterien für die Bewertung von Modellrepräsentationen werden Typisierungsgrad, Universalität, graphische Darstellung sowie Komposition von Modellen verwendet.

2.1.1 Typisierung

Das wohl wesentliche Merkmal einer Modellrepräsentation ist seine Typisierung. Es gibt auf der einen Seite gänzlich untypisierte Repräsentationen wie beispielsweise Bit-Streams, die viel Flexibilität, aber keinerlei Typsicherheit oder strukturellen Datenzugriff mit sich bringen. Auf der anderen Seite stehen statisch getypte Objekte, die komfortablen Zugriff und Typsicherheit garantieren, im Gegenzug aber sehr unflexibel sind. Dazwischen stehen universelle Notationen wie XML, die eine Struktur, aber keine direkte Typisierung besitzen[1].

2.1.2 Universalität

Manche Code-Generatoren sind nur für eine spezielle Ausgabesprache gedacht, was sich in den festen Modellen wiederspiegelt. Andere verwenden Modelle, die bereits universell und direkt auf jede Domäne anwendbar sind. Dazwischen liegen Systeme, in denen der Metaprogrammierer neue Modelle definieren kann und daher eine Übertragung möglich, allerdings mit einem Mehraufwand verbunden ist.

[1] Es gibt zwar die Möglichkeit, XML-Dokumente gegen eine externe Spezifikation zu validieren oder automatisch in ein typisiertes Objekt zu überführen. Allerdings ist dies eher mit einer Grammatik zum Parsen von Strings vergleichbar als mit inhärent typisierten Daten.

2.1.3 Graphische Darstellung

Die Möglichkeit, ein Modell zu visualisieren oder gar mit einer graphischen Benutzeroberfläche zu bearbeiten, ist sehr praktisch für Dokumentation, Übersichtlichkeit sowie Zusammenarbeit mit Domänenexperten, die das verwendete System nicht genau verstehen müssen.

2.1.4 Komposition

Sinnvollerweise sollten Modelle schachtelbar sein, so dass komplexe Modelle aus einfacheren zusammengesetzt werden können. Dies lässt sich durch die Konzepte Unterknoten oder Attribut ausdrücken.

Modelle lassen sich oft als knotenbenannte Bäume beschreiben, in denen die Kinder eines Knotens als Untermodelle oder Komponenten aufgefasst werden. Dies entspricht dem Entwurfsmuster Komposition [29, S.182f]. Unterknoten sind nicht weiter gekennzeichnet und können somit nur durch ihre Reihenfolge oder ihren Inhalt (zu dem auch der Name gehört) unterschieden werden. Abb. 2.1 zeigt eine simple Aufzählung als Listenknoten mit zwei Unterknoten.

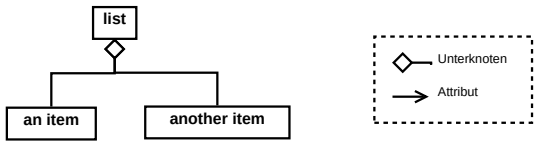


Abbildung 2.1: Eine einfache Aufzählung als Baumstruktur

Attribute hingegen ordnen auf Ebene des Oberknotens Eigenschaftsnamen Werte zu, was eher dem gängigen Bild von Objekteigenschaften entspricht. Eine Komposition durch Attribute ist nur dann möglich, wenn als Werte wiederum beliebige Modelle zugelassen sind – was beispielsweise nicht für XML-Attribute gilt. Da bereits eine Benennung auf oberer Ebene erfolgt, besitzen Unterknoten hier oft keinen Namen mehr. Manche Repräsentationen erlauben auch listenwertige Attribute. Abb. 2.2 zeigt die Repräsentation eines betitelten Bildes durch Attribute, welche als beschriftete Kanten dargestellt werden.

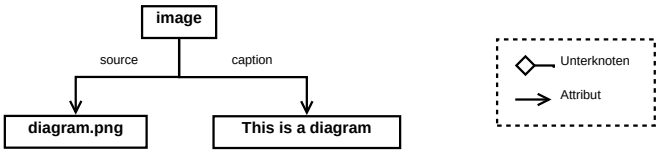


Abbildung 2.2: Eigenschaften eines Bildes als Attribute

Beide Sichtweisen lassen sich leicht ineinander überführen. Um beispielsweise Attribute durch Unterknoten zu ersetzen, wird pro Attribut ein neuer Knoten desselben Namens eingeführt (oder ein bestehender, anonymer Unterknoten benannt). So zeigt Abb. 2.3 das letzte Beispiel nochmals in Knotenrepräsentation. Aus diesem Grund reicht bereits die komplette Unterstützung eines der beiden Konzepte, um alle Ausdrucksmöglichkeiten bereitzustellen.

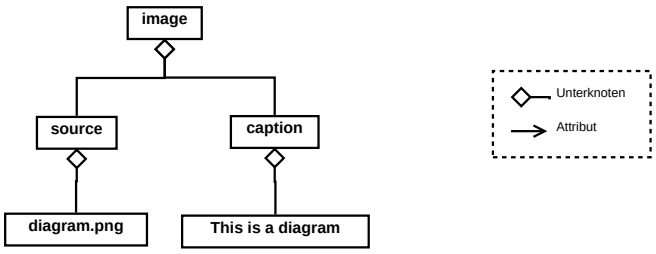


Abbildung 2.3: Eigenschaften eines Bildes als Knoten

Im nächsten Abschnitt werden die verschiedenen Modellarten im Hinblick auf diese Kriterien diskutiert.

2.2 Strings

Zeichenketten stehen auf der niedrigsten Stufe der Datenrepräsentation – schließlich sind sie lediglich Bitfolgen. Damit sind Strings immer dann erforderlich, wenn Modelle zur persistenten Speicherung oder Kommunikation serialisiert werden müssen.

Um ein Modell als String zu repräsentieren, muss zunächst eine Syntax festgelegt werden. Eine Folie, die eine Aufzählung enthält, könnte beispielsweise so codiert werden:

```
a slide titled "My Slide" with list "one", "two" and "three"
```

Da Zeichenketten per se keinerlei Struktur oder Typisierung ermöglichen, ist eine String-Repräsentation für die Erzeugung und Bearbeitung sowie komplexere Verarbeitung von Modellen eher ungeeignet. Vor der Verarbeitung sollte ein String deshalb vom Parser in ein strukturiertes Modell umgewandelt werden. Eine graphische Darstellung besitzen Strings nicht, wenn man von der Anzeige als Text absieht.

2.3 Objekte

Das Wort „Objekt“ wird in dieser Arbeit nicht ausschließlich im objektorientierten Sinn verwendet – vielmehr ist hier „Datentypinstanz einer Programmiersprache“ gemeint, so dass auch nicht objektorientierte Sprachen einbezogen werden. Objekte bieten den höchsten Typisierungsgrad und die damit verbundene Sicherheit[2]. Es muss zwischen

[2]Eine weitere Steigerung der Sicherheit wäre über Typen hinausgehende Validierung von Modellen.

Objekten in statisch und dynamisch getypten Sprachen unterschieden werden, denn in dynamischen Typsystemen werden Typfehler erst beim Zugriff, bei Code-Generierung also vermutlich erst im Übersetzungsvorgang erkannt. In den meisten Sprachen gibt es Attribute für Objekte sowie einen Listen- oder Arraydatentyp. Unbenannte Komponenten haben Objekte gewöhnlich nicht – eine Ausnahme bilden Haskell-Datentypen (siehe Abschnitt 6). Eine Visualisierung von Objekten kann als knotenbenannter Graph erfolgen. Abb. 2.4 zeigt obiges Folienbeispiel in einer statisch typisierten Programmiersprache mit listenwertigen Attributen.

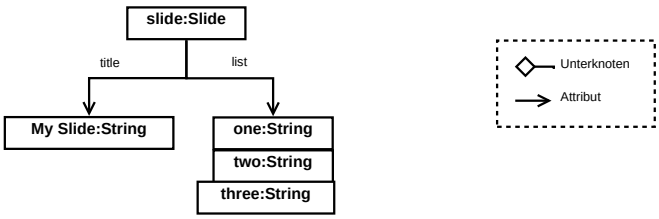


Abbildung 2.4: Folie als typisiertes Objekt

2.4 S-Expressions

S-Expressions bilden die grundlegende Syntax aller Lisp-Dialekte. Die Definition ist sehr einfach: Ein S-Expression ist entweder ein Symbol oder eine Liste von S-Expressions. Symbole sind Zeichenketten[3], die keine Leerzeichen oder andere vom jeweiligen Lisp verwendeten Sonderzeichen enthalten. Listen werden in runde Klammern gesetzt – mit Leerzeichen als Trennsymbol. Beispiele für S-Expressions sind:

- `foo`
- `(slide)`
- `(list one two three)`

S-Expressions sind zwar formal als Listen definiert, werden aber meist als Bäume mit benannten Knoten interpretiert – hierfür wird das erste Element einer Liste – typischerweise ein Symbol – als Name sowie alle weiteren als Unterknoten aufgefasst. Zwischen `(foo)` und `foo` wird dabei weiterhin unterschieden. Nicht alle gängigen Verwendungen von S-Expressions fügen sich jedoch in dieses Baumschema ein. Beispielsweise enthält Common Lisp oft Konstrukte in der Art von `(interval (1 10))`, wo in `(1 10)` die 1 keinen Namen, sondern einen Wert darstellt. Konsistenter wäre hier `(interval 1 10)`.

[3]Die meisten Lisp-Dialekte unterscheiden nicht zwischen Groß- und Kleinbuchstaben in Symbolen – das im Rahmen dieser Arbeit entwickelte Framework hingegen schon.

Darüber hinaus kommen oft der „leere S-Expression“ `()` sowie nicht-symbolische Knotennamen wie in `((lambda x (+ x 1)) 41)` vor, wo eine anonyme Funktion direkt auf einen Wert angewendet wird.

Über die bloße S-Expression-Syntax hinaus besitzen Lisp-Dialekte oft noch Syntax für Kommentare, Strings, Symbolnamen mit Leerzeichen etc. Dennoch sind S-Expressions wohl die einfachste benutzte Repräsentationsform für strukturierte Daten. Das Modell selbst ist ungetypt, da in den Blättern Strings stehen. Eine Validierung kann daher erst nach dem Einlesen erfolgen.

Attribute sind in der Grundstruktur nicht vorgesehen, können aber wie oben beschrieben durch Unterknoten simuliert werden. Beispielsweise könnte eine Folie mit Bild und Aufzählung wie folgt dargestellt werden:

```
(slide
  (title My Slide)
  (image
    (source diagram.png)
    (caption This is a diagram))
  (list one two three))
```

Dies entspricht dem Baum in Abb. 2.5. In konkreten Lisp-Dialekten würde man diese Repräsentation selten anfinden: Zum einen würde eine String-Notation zur Unterscheidung von Werten und Variablen verwendet werden, zum anderen ist es üblich, feste Parameter wie Folientitel nicht zu benennen, sondern implizit durch die Reihenfolge zuzuordnen. Dies könnte dann derart aussehen:

```
(slide "My Slide"
  (image "diagram.png" "This is a diagram")
  (list "one" "two" "three"))
```

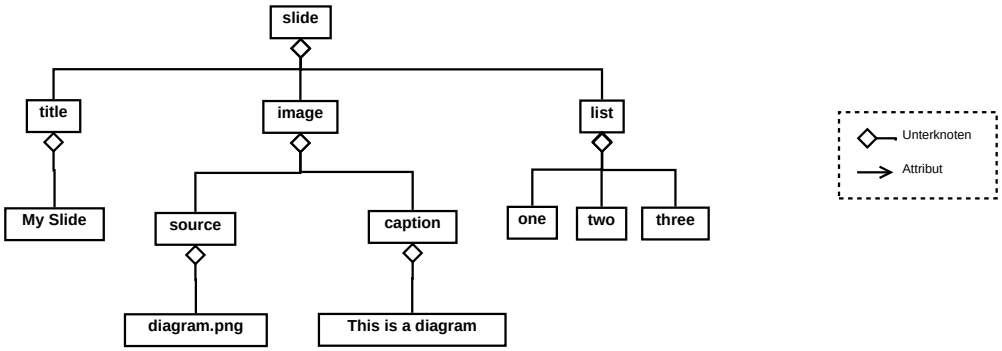


Abbildung 2.5: Baumansicht einer Folie in S-Expressions

Trotz der simplen Struktur und Syntax, die auch benötigte Werkzeuge wie Editoren, Parser usw. vereinfachen, bieten S-Expressions also eine universelle Modellierungssprache an, mit der auch Attribute und Listen indirekt repräsentiert werden können.

2.5 XML

XML ist die wohl derzeit meistverbreitete Markup-Sprache, weshalb hier auf eine detaillierte Einführung verzichtet wird. Wie bei S-Expressions handelt es sich um eine strukturierte, ungetypte Repräsentation, die einem knotenbenannten Baum entspricht. XML unterscheidet sich von S-Expressions – Syntax zunächst ausgenommen – hauptsächlich durch das Vorhandensein von Attributen, die allerdings nur Strings als Werte erlauben. Darüber hinaus gibt es Features wie Namensräume, Referenzen und CDATA-Abschnitte, welche Zeichenketten repräsentieren können, die mit XML-Syntax interferieren würden.

Das Folien-Beispiel lässt sich in XML wie folgt modellieren:

```
<slide title="My Slide">
  <image source="diagram.png">
    This is a diagram
  </image>
  <list>
    <item>one</item>
    <item>two</item>
    <item>three</item>
  </list>
</slide>
```

Titel und Bildquelle werden als Attribute modelliert, als Bildanschrift wird implizit das Innere vom Knoten `image` verwendet. Das Beispiel zeigt, dass die XML-Syntax im Vergleich zu S-Expressions eine erhöhte Redundanz aufweist. Die erzwungene Wiederholung des Knotennamens in schließenden Tags soll die Robustheit im Falle vergessener Tags erhöhen, hat aber neben dem erhöhten Schreibaufwand den Nachteil, dass XML damit im Gegensatz zu S-Expressions eine kontextsensitive Sprache und folglich aufwändiger zu parsen ist. XML-Parser, die kontextfreie Parser-Generatoren benutzen, müssen deshalb anschließend in einem zusätzlichen Durchlauf die korrekte Zuordnung der Schließ-Tags überprüfen. Die Schreibarbeit lässt sich zwar durch die Verwendung eines speziellen XML-Editors wieder verringern, allerdings könnte ein solcher auch die Zuordnung von Unterknoten – beispielsweise durch Einrückung – verdeutlichen, was die Redundanz wieder unnötig erscheinen lässt.

Ein anderer Grund weshalb das Beispiel umständlich wirkt ist, dass XML-Knoten als Unterknoten zwar beliebig viele XML-Knoten, aber im Gegensatz zu S-Expressions nur einen String besitzen können. Aus diesem Grund sowie dem Nichtvorhandensein listenwertiger Attribute sind zusätzliche `item`-Knoten erforderlich[4]. Abb. 2.6 zeigt die Baumdarstellung dieses Beispiels.

Ein häufiges Problem bei der Definition von XML-Formaten ist die Frage, ob eine spezielle Eigenschaft durch einen Unterknoten oder durch ein Attribut repräsentiert werden sollte. Attribute sind in der Schreibweise kürzer und deshalb komfortabler, auf

[4] Würde man stattdessen den `list`-Knoten weglassen, gäbe es Probleme bei mehreren Listen auf einer Folie.

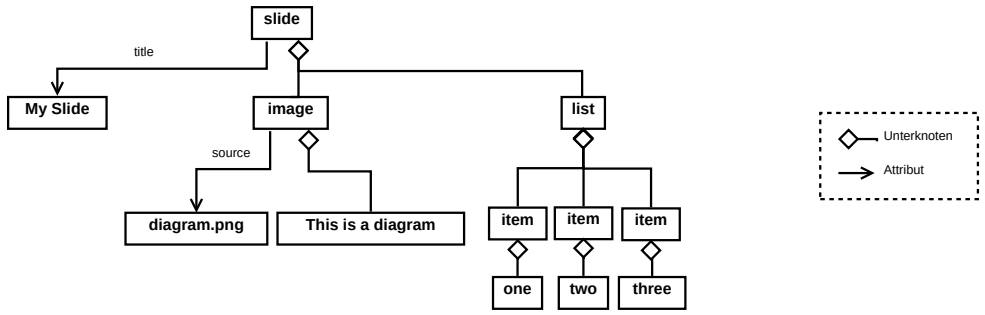


Abbildung 2.6: Baumansicht einer Folie in XML

der anderen Seite erlauben sie aber keine zusammengesetzten Werte[5]. Es kommt deshalb durchaus vor, dass eine Eigenschaft zunächst als Attribut realisiert wird, dann aber später bei wachsender Komplexität und Modellierungsgenauigkeit durch einen Unterknoten ersetzt werden muss, wodurch eine Inkompatibilität zu alten Dateien entsteht. Beispielsweise könnte es sich nachträglich als Fehler herausstellen, den Folientitel als Attribut gewählt zu haben, wenn spätere Slide-Erweiterungen es erlauben sollen, Titel mittels `color`-Tag einzufärben.

Es stellt sich also die Frage, ob XML-Attribute überhaupt notwendig sind oder vielmehr ein optionales Feature, was nur aufgrund der umständlichen Syntax für Knotendefinitionen genutzt wird. In diesem Fall sollten Attribute wohl zukünftig aus XML verschwinden, schließlich schreibt das W3C [44]: „The number of optional features in XML is to be kept to the absolute minimum, ideally zero“.

Ist also eine zweite Art Eigenschaften zu definieren für bestimmte Zwecke nützlich? Ein Beispiel wäre die Trennung von normalen und Meta-Eigenschaften, wobei letztere als Attribute repräsentiert werden könnten. Ein anderes ist die Aufteilung zwischen Pull- und Push-Verarbeitung: Attribute werden üblicherweise direkt mit einem per Query erfragten Knoten mitgeliefert (Push), während Unterknoten nur mitgeschickt werden, wenn diese explizit abgefragt wurden (Pull). Auch für die Trennung von optionalen und erforderlichen Eigenschaften könnte solch ein zweiter Kantentyp nützlich sein.

Das Problem bei all diesen Beispielen ist jedoch die Einschränkung auf String-Attribute. Sobald ein Attribut einen strukturierten Wert erhalten soll, bricht jede systematische Trennung zusammen, und es muss auf Unterknoten zugegriffen werden. In diesem Fall ist es aber vermutlich besser, von Anfang an auf Attribute zu verzichten.

2.6 JSON

JSON ist sowohl eine Markupsprache als auch eine Teilmenge von JavaScript [16], weshalb es direkt von einem JavaScript-Interpreter eingelesen werden kann. Dies macht

[5]Es ist zwar möglich, über einen Attribut-String im XML-Format innere Struktur in Attribute einzubauen – dieser würde dann aber nicht automatisch geparst werde, so dass von diesem Trick eher abzuraten ist.

JSON vor allem für Webanwendungen interessant. Mit JSON können anonyme Objekte mit Attributen – auch listenwertige – explizit repräsentiert werden. Als Blätter sind Strings und elementare Datentypen zugelassen, die beim Einlesen dynamisch typisiert werden.

Das Folien-Beispiel kann in JSON wie folgt repräsentiert werden:

```
{
  "title" : "My Slide"
  "image" : {
    "source"  : "diagram.png"
    "caption" : "This is a diagram"
  }
  "list"  : ["one", "two", "three"]
}
```

Die Benennung als `slide` muss außerhalb erfolgen, d.h. bei einem übergeordneten Objekt als Attribut oder auf der höchsten Ebene beispielsweise durch den Dateinamen.

Aufgrund der Austauschbarkeit von Attributen und benannten Unterknoten sind JSON und S-Expressions gleich ausdrucksstark, wie das Beispiel zeigt. Der einzige strukturelle Unterschied besteht im obersten Knoten, der bei JSON anonym und bei S-Expressions benannt ist.

2.7 UML

UML ist eine graphische Modellierungssprache, die speziell für die Spezifikation von Softwaresystemen entworfen wurde und insbesondere den objektorientierten, imperativen Entwurf unterstützt. Viele Werkzeuge können aus UML beispielsweise Java-Quelltext generieren. Es gibt in UML eine Reihe verschiedener Diagrammarten, die verschiedene Knoten- und Kantentypen in einem Graphen erlauben. Teilweise können auch Unterknoten in einen Knoten eingebettet werden. Abgespeichert werden UML-Diagramme meist in einem XML-Format, da UML keine eigene Serialisierung definiert.

Nach graphischen Elementen ist bereits eine Teilmenge der UML-Klassendiagramme äquivalent zu XML etc., da diese bereits knotenbenannte Bäume mit Attributen darstellen kann. Die bisherigen Abbildungen dieses Kapitels benutzen beispielsweise Aggregations- und Attributkanten um Unterknoten und Attribute zu repräsentieren. Es handelt sich hierbei jedoch nicht um semantisch korrekte UML-Diagramme – beispielsweise wurden benannte Knoten wie UML-Klassen aufgemalt. Abb. 2.7 zeigt eine Modellierung des Folienbeispiels als UML-Objektdiagramm.

Wenn man also die von UML (mehr oder weniger) klar definierte, objektorientierte Semantik aller graphischen Elemente berücksichtigt, handelt es sich nicht mehr um eine abstrakte, domänenunabhängige Sprache.

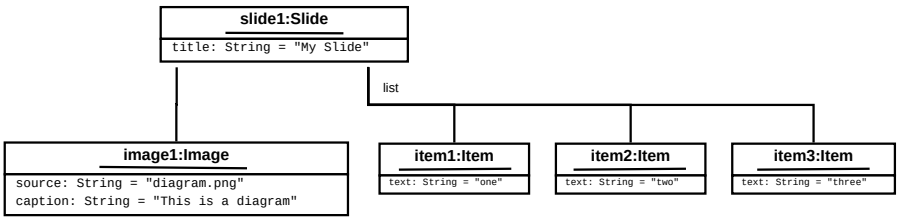


Abbildung 2.7: Folienbeispiel als UML-Objektdiagramm

2.8 Zusammenfassung

Es gibt in bestehenden Code-Geneneratoren drei grundlegende Arten, ein Modell zu repräsentieren, die alle ihre Daseinsberechtigung haben und über Parser, Generatoren und Compiler ineinander umgewandelt werden können.

Strings sind immer nötig, wenn ein Modell dauerhaft gespeichert oder verschickt werden soll. Für alle anderen Zwecke sind diese dagegen eher unpraktisch.

Für eine aufwändigere Verarbeitung, die komplexe Rechnungen oder Algorithmen erfordert, eignen sich Objekte am besten. Fehler werden vor allem bei statisch typisierten Objekten frühestmöglich erkannt. Dafür ist es allerdings nötig, spezielle Klassen bzw. Datentypen für diese Modelle zu definieren. Zudem ist eine Übertragbarkeit zwischen Programmiersprachen im Allgemeinen nicht gegeben.

Untypisierte Baumrepräsentationen eignen sich aufgrund ihrer Struktur und Flexibilität besonders für die Erzeugung und Bearbeitung von Modellen. Mit einem geeigneten universellen Editor könnten Benutzer komfortabel und ohne die Gefahr von Syntaxfehlern Modelle beliebiger Domänen erzeugen, ohne dass der Editor vorher Informationen über Syntax oder Semantik der Modelle benötigt. Auch die maschinelle Erzeugung von Modellen ist in einer Baumrepräsentation am einfachsten. Simple Verarbeitungsprozesse können mit Bäumen ebenfalls gut durchgeführt werden.

Abb. 2.8 zeigt eine tabellarische Zusammenfassung des in diesem Abschnitt erfolgten Vergleichs von Strings, Objekten, UML und den Baumrepräsentationssprachen S-Expressions, XML und JSON. Den S-Expressions fällt unter den Baumrepräsentationen eine besondere Rolle zu, da diese im Gegensatz zu beispielsweise XML minimal und redundanzfrei definiert sind, ohne an Ausdrucksmächtigkeit zu verlieren. Damit wird S-Expressions verarbeitenden Werkzeugen keinerlei unnötige Komplexität aufgezwungen. Aus diesem Grund widmet sich diese Arbeit weitgehend der S-Expression-gestützten Code-Generierung.

Modell	String	Objekt	S-Expressions	XML	JSON	UML
Typisierung	gering	sehr hoch	mittel	mittel	mittel	relativ hoch
Domänenunabhängig	ja	Klasse erforderlich	ja	ja	ja	nein
Graphische Darstellung	Text	Graph	Baum	Baum	Baum	Graph
Unterknoten	nein	selten	ja	ja	nein	ja
Attribute	nein	fast immer	nein	primitiv	ja	ja
Listenwertige Attribute	nein	meistens	nein	nein	ja	nein
Feature-Anzahl	minimal	nach Sprache	minimal	mittel	klein	groß

Abbildung 2.8: Vergleich verschiedener Modellrepräsentationen

3 Template-basierte Code-Generierung

Templates werden für die Erzeugung von Dokumentmengen eingesetzt, für die gewisse statische Bereiche in jedem Dokument identisch sind, während sich nur die dynamischen Bereiche ändern. Ein Beispiel könnte eine Webseite sein, die Daten aus einer Datenbank anzeigt. Auch bei der Erzeugung von Programmen tritt dieser Fall auf, so dass Templates eine große Rolle bei der Code-Generierung spielen. Ein Template legt die statischen Bereiche fest und lässt Platzhalter für die dynamischen. Es handelt sich dabei im Grunde um eine mathematische Funktion, deren Eingabe die dynamischen Daten und deren Ausgabe ein fertiges Dokument ist.

Gewissermaßen können also Templates bereits in jeder Programmiersprache definiert werden, die Funktionsdefinitionen ermöglicht. Solch eine Funktion müsste das Zieldokument mittels String-Operatoren zusammensetzen, was im Normalfall zu umständlich ist. Stattdessen bieten Templating-Sprachen eine „schablonenartige“ Notation, die sich gut anhand der im Lisp-Umfeld üblichen Operatoren Quasiquote[1] und Unquote beschreiben lässt. Ein Quasiquote erzeugt ein Template, und das Innere eines Quasiquotes wird als statisch interpretiert, solange darin kein Unquote auftritt. Ein Unquote, welches nur innerhalb eines Quasiquote auftauchen darf, erzeugt einen dynamischen Unterabschnitt, welcher in der vom Werkzeug bereitgestellten Metasprache beschrieben ist. Hier werden die Eingabeparameter des Templates verwendet. Der Vorteil dieser Notationsform besteht darin, dass ein Template – bis auf die dynamischen Abschnitte – genauso aussieht wie das Zieldokument und dadurch gut lesbar bleibt.

Es gibt viele verschiedene Werkzeuge, die vom Template-Prinzip Gebrauch machen. Diese unterscheiden sich oft neben der Syntax in vielen weiteren Punkten. Dieser Abschnitt stellt zunächst einige Vergleichskriterien vor, anhand derer anschließend eine Auswahl Template-basierter Werkzeuge zur Code-Generierung verglichen und bewertet wird.

3.1 Vergleichskriterien

Unterschiede zwischen verschiedenen Template-basierten Code-Generatoren bestehen in den verwendeten Modellen, der konkreten Weise der Template-Erzeugung, der Mächtigkeit der Metasprache, der Art des (eventuell rekursiven) Aufrufs von Templates sowie Unterstützung von Komponententrennung und der Möglichkeit, wiederum Templates aus Templates generieren zu können.

[1] Neben Quasiquote ist auch der Begriff Backquote üblich. Zudem gibt es in Lisp-Sprachen oft auch ein einfaches Quote, welches einen komplett statischen Bereich erzeugt, ohne darin auf Unquotes zu achten.

3.1.1 Verwendete Modelle

Template-Engines unterscheiden sich sowohl in den Eingabe-, als auch in den Ausgabemodellen. Verwendet werden für beides gewöhnlich Zeichenketten, typisierte Objekte sowie untypisierte Bäume wie S-Expressions oder XML (siehe Kapitel 2). Mit den Modellen variieren die Typisierungsgrade und damit auch das Maß und der Zeitpunkt einer möglichen Validierung der Daten.

Die generierbaren Zielsprachen unterscheiden sich ebenfalls. Manche Generatoren können jede Sprache generieren, während andere nur eine einzige oder eine Familie von Sprachen unterstützen.

3.1.2 Art der Template-Erzeugung

Auch wenn fast alle Ansätze der Template-Erzeugung sich mit Quote und Unquote beschreiben lassen, gibt es einige Unterschiede bei der genauen Verwendung.

Nicht überall sind Quasiquotes explizit – oft interpretieren Templating-Systeme auch automatisch die äußerste Ebene einer Template-Datei als quotiert. Manche Werkzeuge besitzen kein universelles Quasiquote, mit dem sich wirklich jede gewünschte Code-erzeugende Funktion ausdrücken lässt. Aus syntaktischen Gründen könnte es beispielsweise nicht möglich sein, eine variable Anzahl an Statements zu erzeugen.

Auch die Möglichkeit weiterer Quasiquotes in Unquote-Bereichen ist nicht überall gegeben. Darüber hinaus benutzen manche Systeme ein implizites Unquote, welches automatisch eingesetzt wird, wenn ein Token mit dem Namen eines Template-Parameters übereinstimmt.

Manche Werkzeuge bieten auch die Möglichkeit, Code nicht über die Templates-Notation, sondern direkt programmatisch (beispielsweise durch String-Manipulationen) zu erzeugen. Vor allem in Systemen ohne universelles Quasiquote wird diese Hintertür benötigt.

3.1.3 Metaprogrammiersprache

Die Bandbreite bei der Sprachmächtigkeit in Unquote-Bereichen ist hoch: Es gibt Werkzeuge, die überhaupt keine Berechnungen, sondern lediglich das Einfügen der Parameter zulassen. Auf der anderen Seite stehen vollständige (Meta-)Programmiersprachen, wobei Zwischenstufen ebenfalls möglich sind. Zudem gibt es Unterschiede beim mit der Erstellung eines Metaprogramms verbundenen Schreibaufwand.

3.1.4 Aufruf von Templates

Ein weiterer Unterschied zwischen Code-Generatoren liegt in der Art, ob und wie Templates selbst andere Templates aufrufen können. Manche Systeme können nur ein einzelnes Template verarbeiten, so dass alles weitere in einer externen Programmiersprache festgelegt ist. Engines mit einer vollständigen Metaprogrammiersprache bieten hingegen oft die Möglichkeit, programmatisch die Template-Engine selbst und damit indirekt ein

weiteres Template aufzurufen. Andere lassen den direkten Aufruf weiterer Templates zu. Eine Alternative zur aufrufbasierten Steuerung ist durch Makros gegeben. Makros sind Templates, die an einen Knotennamen oder ein Muster gebunden sind. Der Generator entscheidet dann anhand des Modells, welches Template für die Verarbeitung zuständig ist.

3.1.5 Komponententrennung

Die meisten Software-Systeme lassen sich durch eine Model-View-Controller-Architektur (MVC) [20, S.5ff] beschreiben. Daten und deren Verarbeitung werden vom Modell verwaltet. Ablaufsteuerung und Behandlung von Benutzerinteraktionen gehören in den Controller. Views sind für die graphische oder textuelle Formatierung der Modelle zuständig.

Code-Generatoren benötigen normalerweise sehr wenige Benutzerinteraktionen und haben eine simple Ablaufsteuerung, da diese lediglich Eingabemodelle in Ausgabemodelle übersetzen. Controller stellen deshalb meist nur einen kleinen Anteil am Generator. Als Views können Templates aufgefasst werden, weshalb diese meist den Hauptteil bilden.

Eine klare Komponententrennung zwischen diesen Bereichen verbessert die Modularität und somit Übersichtlichkeit, Wartbarkeit und Aufgabenverteilung innerhalb des Projektes. Deswegen gelten beispielsweise Views, die komplexe Berechnungen am Modell vornehmen, als schlechter Stil. Es ist daher von Vorteil, wenn Generatoren eine Trennung von Model, View und Controller unterstützen oder gar forcieren – auf der anderen Seite sollte das Werkzeug aber nicht im Wege stehen, wenn eine ungewöhnliche Aufgabe andere Mittel erfordert.

3.1.6 Generierung von Templates

Code-Generierung bietet generell die Möglichkeit, eine Programmiersprache um neue Abstraktionen zu erweitern, indem die neuen Konzepte vom Generator auf bestehende abgebildet werden. Manchmal ist es aber von Nutzen, die Metasprache selbst erweitern zu können. Dies lässt sich unter anderem erreichen, indem der Template-Code selbst generierbar ist.

Manche Engines können nur Code einer speziellen Zielsprache generieren, die nicht mit der Metasprache übereinstimmt – hier sind Templates offensichtlich nicht generierbar. Aber auch Frameworks, die prinzipiell jede Sprache generieren können, haben oft Probleme, wenn Metasprache und Zielsprache übereinstimmen. Soll beispielsweise ein Unquote in der Zielsprache generiert werden, wird es ohne weitere Maßnahmen fälschlich als Unquote der Metasprache interpretiert werden, was zu einem Fehler führt. Stattdessen muss dieses Unquote auf eine umständliche Weise, z.B. durch Escape-Sequenzen, erzeugt werden.

Im Folgenden werden nun die Template-Prozessoren – oder mit Templates verwandten Systeme – PHP, StringTemplate, C++, Template Haskell, C, XSL und Lisp vorgestellt

sowie anhand der vorangegangenen Kriterien eingeordnet und diskutiert.

3.2 PHP

Die Skriptsprache PHP ist von vornherein als Template-System konstruiert, womit Code beliebiger Sprachen generiert werden kann. Viele Web-Frameworks benutzen PHP oder eine PHP ähnelnde Syntax, um Templates für HTML-Dateien zu beschreiben. Jeder Block PHP-Code muss mit `<?` und `?>` umgeben sein, was praktisch ein Unquote darstellt. PHP ist selbst die (Turing-vollständige) Metasprache in diesem Template-System. Parameter in PHP sind dynamisch typisierte Objekte. Das Ergebnis der PHP-Blöcke wird als String in die Ausgabedatei eingefügt, die ansonsten aus den implizit quotierten Nicht-PHP-Bereichen der Template-Datei besteht. Die Unquote-Syntax ist an XML angelehnt, da PHP hauptsächlich für die Erzeugung von HTML-Dokumenten verwendet wird. Statt eines direkten Template-Aufrufs gibt es die Möglichkeit, über die Funktion `include` eine andere PHP-Datei einzubinden. Eine Trennung von Model und View wird von PHP nicht speziell unterstützt oder gefordert, ist aber möglich. Generierung von PHP-Dateien selbst ist nur schwer umzusetzen.

Als Beispiel hier ein PHP-Template, welches HTML-Code für eine Aufzählungsliste erzeugt:

```
<ul>
<? foreach ($items as $item) {
     echo("<li>" . $item . "</li>");
   }
?>
</ul>
```

Hierbei ist `$items` der Eingabeparameter, der eine Liste von Strings enthält. Aufgrund der fehlenden Möglichkeit zur Schachtelung von Templates müssen die einzelnen Aufzählungspunkte mit String-Konkatenation erzeugt werden, was bei einem größeren Beispiel unübersichtlich werden könnte.

3.3 StringTemplate

StringTemplate [36] ist eine zeichenkettenbasierte Template-Engine für Java. Auch hier ist eine Template-Datei implizit quotiert, ein Unquote erfolgt mittels `$foo$` oder `<foo>`. Im Gegensatz zu PHP und den meisten anderen Systemen – beispielsweise Velocity [7] für Java – ist die Mächtigkeit der Unquote-Bereiche in StringTemplate stark reduziert, um eine Model-View-Trennung zu erzwingen – alle Berechnungen sollen also im Modell, und nicht im Template selbst erfolgen.

Neben der Referenzierung von Java-Objekten – hier wird automatisch `toString()` aufgerufen – und deren Instanzvariablen sind Template-Aufrufe sowie `if`-Abfragen erlaubt, wobei letztere nur zwischen `true` und `false` bzw. `null` und nicht-`null` unterscheiden, aber keine Berechnungen oder Vergleichsoperatoren benutzen können. „Anonyme Templates“

(analog zu anonymen Funktionen) entsprechen einem Quasiquote – es ist allerdings auch möglich, extern definierte Templates aufzurufen. Statt der Bereitstellung eines Konstrukts wie `foreach` iteriert StringTemplate automatisch, wenn ein referenzierter Wert eine Liste ist, was auch bei Template-Aufrufen funktioniert. Hier kann zusätzlich ein Separator-String angegeben werden, der dann jeweils zwischen den Elementen steht. Es gibt auch die Möglichkeit, bei der Verarbeitung einer Liste abwechselnd zwei oder mehr Templates aufzurufen. Dies wird beispielsweise für sich abwechselnde Farben von Zeilen in HTML-Tabellen benötigt.

Der hier verfolgte Ansatz funktioniert in der Praxis wohl sehr gut und erreicht tatsächlich im Vergleich zu anderen Engines eine sauberere Aufteilung der Module sowie übersichtlichere und kürzere Template-Dateien. Der einzige Schwachpunkt tritt auf, wenn Berechnungen durchgeführt werden müssten, die eigentlich nicht ins Modell, sondern ins View gehören, wie beispielsweise ein rotes Einfärben von negativen Werten. Hierfür müsste ein zusätzliches Attribut `isRed` im Modell bereitgestellt werden, wodurch die Trennung wiederum verletzt wäre. Für solche Fälle wäre es hilfreich, wenn StringTemplate das Definieren von Helper-Klassen, deren Methoden aus den Templates aufgerufen werden können, unterstützen würde.

Die Erzeugung einer HTML-Liste kann in StringTemplate wie folgt aussehen:

```
<ul>
$items:{
  <li> $it$ </li>
}$
</ul>
```

Über die Eingabeliste `items` wird automatisch iteriert und für jedes Element ein anonymes inneres Template aufgerufen, um den Listeneintrag zu erzeugen. Da der innere Template-Parameter nicht benannt wurde, benutzt StringTemplate standardmäßig `it`.

3.4 Templates in C++

C++-Templates bieten die Möglichkeit, Klassen oder Funktionen unter anderen durch Typnamen zu parametrisieren, um so generische Konstrukte wie Collection-Klassen definieren zu können. Die bekannten Java-Generics [34] wurden Templates nachempfunden und haben eine ähnliche Syntax, sind allerdings simpler und weniger mächtig. Templates werden direkt vom C++-Compiler ausgewertet, so dass es kein sichtbares oder verfügbares Ausgabemodell gibt. Neben Typen können auch bestimmte Werte wie ganze Zahlen und andere Templates übergeben werden. Der Aufruf des Templates erfolgt durch die Übergabe des konkreten Typnamens oder Wertes in spitzen Klammern bei Benutzung der entsprechenden Klasse oder Funktion. Mit C++-Templates lässt sich sehr viel mehr konstruieren als lediglich generische Collection-Klassen – beispielsweise zeigt Abb. 3.1 die Berechnung von Fibonacci-Zahlen zur Kompilationszeit[2]. Da aber nur sehr primitive

[2]Dieses Programm wurde von [2] übernommen. Weitere kreative Verwendungsmöglichkeiten von Templates finden sich in der Bibliothek Boost [1].

```
template<int N> struct fib {
  static const int result = fib<N-1>::result + fib<N-2>::result;
};

template<> struct fib<0> {
  static const int result = 0;
};

template<> struct fib<1> {
  static const int result = 1;
};

// fib<10>::result ergibt 55
```

Abbildung 3.1: Die Fibonacci-Folge mit C++-Templates

Eingabemodelle zugelassen und komplexere Berechnungen nicht vorgesehen sind, sind andere in diesem Kapitel vorgestellte Systeme deutlich universeller als C++-Templates. Eine Model-View-Trennung ist ebenfalls nicht möglich, ohne komplexe Modelle aber auch nicht notwendig. Da Aufrufe von Templates über eine spezielle Syntax erfolgen, handelt es sich hier nicht um Makros. Templates werden direkt vom Compiler ausgewertet und nicht etwa einem eigenständigen Präprozessor. Deswegen ist es nicht möglich, Templates selbst zu generieren.

3.5 Template Haskell

Template Haskell ist eine Compiler-Erweiterung für Haskell, die die Grundidee von C++-Templates als „funktionale Kompilationszeit-Sprache“ aufgreift und erweitert, um ein sehr mächtiges System zu erhalten. Eingabemodelle können beliebige Haskell-Datentypen sein, die Ausgaben einzelner Templates sind Objekte eines speziellen Datentyps, der Haskell-Code beschreibt. Templates sind selbst Haskell-Funktionen, so dass Metasprache und generierte Sprache identisch sind und Templates somit Turing-vollständig und typsicher sind. In Quelldateien kann Meta-Code mit normalem Code gemischt werden. Zudem bietet Template Haskell Reifikation, d.h. in Templates können Metainformationen wie Typen über Code in derselben Datei abgefragt werden, was enorme Möglichkeiten eröffnet.

Da Templates Funktionen sind, können diese komplett programmatisch erzeugt werden – deswegen ist auch eine beliebige Modularisierung möglich. Darüber hinaus gibt es auch Syntax für Quasiquote und Unquote, wodurch kurzer und lesbarer Meta-Code ermöglicht wird. Es gibt verschiedene Quasiquote-Arten für Befehle, Patterns und Toplevel-Definitionen, wodurch die Typsicherheit verstärkt wird. Dies bringt allerdings eine höhere Komplexität mit sich. Quotierung und programmatische Erzeugung können beliebig kombiniert werden. Die Quelldatei kann als implizit quotiert aufgefasst werden,

da überall ein Unquote benutzt werden kann, um auf die Metaebene zu gelangen. Der quotierte Bereich wird allerdings übernommen – schließlich kann auch „normaler Code“ von Meta-Code aufgerufen werden, so dass dieser auch als Meta-Code verwendet werden kann. Wie in C++ werden die Templates direkt vom Compiler ausgewertet, so dass keine Generierung von Templates selbst möglich ist.

Ein (inneres) Quasiquote wird in Template Haskell durch `[|` und `|]` notiert, ein Unquote durch `$`. Beispielsweise zeigt Abb. 3.2 ein Template[3], das eine anonyme Funktion mit `n` Parametern erzeugt, welche diese aufsummiert. Das Beispiel zeigt auch, wie generierte Variablennamen automatisch nummeriert werden, um Namenskonflikte zu vermeiden.

```
-- Template-Definition
summ n = summ' n [| 0 |]
summ' 0 code = code
summ' n code = [| \x -> $(summ' (n-1) [|$code+x|] ) |]

-- Aufruf
$(summ 3)  -- ergibt (\x1 -> \x2 -> \x3 -> 0+x1+x2+x3)
```

Abbildung 3.2: Generierung einer anonymen Funktion mit n Parametern in Template Haskell

Das Quasiquote ist allerdings nicht umfassend, es gibt also Templates, die nur auf programmatische Art erzeugt werden können. Beispielsweise zeigt Abb. 3.3 ein Template, das eine Zahl n als Eingabe bekommt und den Haskell-Code für ein n-äres Tupel mit den Zahlen 1 bis n zurückliefert. Dieses Template kann nicht mittels Quasiquote implementiert werden, da Tupel variabler Länge syntaktisch in Haskell selbst nicht repräsentierbar sind. Zudem muss Code für Literale wie Zahlen typisiert erzeugt werden, wenn die Zahl selbst nicht vorn vornherein feststeht und so ein Quasiquote benutzt werden kann. Deswegen muss auf Konstruktoren wie TupE und LitE zurückgegriffen werden, wodurch die Ähnlichkeit von Template und generiertem Code verschwindet und die Komplexität leider deutlich erhöht wird.

```
tupleNums n = TupE [LitE (IntegerL i) | i <- [1..n]]

$(tupleNums 5) -- ergibt (1, 2, 3, 4, 5)
```

Abbildung 3.3: Generierung von Tupeln dynamischer Länge in Template Haskell

[3]Beide Beispiele in diesem Abschnitt sind übernommen von [4]

3.6 C-Präprozessor

Der C-Präprozessor, welcher um das Jahr 1973 entwickelt wurde [39], wird unter anderem verwendet, um Abhängigkeiten zwischen Quelltextdateien über bedingte Include-Statements zu verwalten. Er läuft als eigenes Programm vor dem C-Compiler und kann deshalb auch in Kombination mit anderen Sprachen benutzt werden. Mit dem Befehl `#define` können Makros auf Textersetzungsbasis sowie Compiler-Konstanten definiert werden. Abb. 3.4 zeigt beispielsweise ein Makro, mit dessen Hilfe sich zwei verschachtelte Iterationen über Zahlen komfortabler notieren lassen. Ein explizites Unquote wird hier

```
#define loop2d(xmax, ymax)   \
  for(int x=0; x<xmax; x++)  \
   for(int y=0; y<ymax; y++)

loop2d(2, 3) {
  printf("(%d-%d) ", x, y);
}
// Ergibt "(0-0) (0-1) (0-2) (1-0) (1-1) (1-2) "
```

Abbildung 3.4: Ein simples C-Makro für zwei verschachtelte Schleifen

nicht benötigt, da stattdessen nach Übereinstimmung mit Parameternamen entschieden wird. Makros auf Textersetzungsbasis ermöglichen viele zusätzliche Fehlerquellen wie Namenskonflikte. Beispielsweise funktioniert `loop2d` nicht, wenn es noch andere Variablen `x` oder `y` gibt, die im Inneren verwendet werden.

Zusätzlich bietet der Präprozessor die Möglichkeit, mittels `#if` einfache Abfragen auf Compiler-Konstanten wie das Vorhandensein sowie simple, ganzzahlige Berechnungen und Vergleiche durchzuführen und davon abhängig Zeilen einzubinden oder wegzulassen. Darüber hinaus sind keinerlei Berechnungen im Präprozessor möglich.

Da es sich um Makros handelt, werden diese Templates nicht über eine spezielle Syntax aufgerufen. Ein Makro-Aufruf gleicht stattdessen einem Funktionsaufruf und die Unterscheidung zwischen beiden erfolgt durch die Prüfung, ob ein Makro dieses Namens vorhanden ist. Dies kann man sowohl positiv als auch negativ sehen: Meist ist es für einen Programmierer angenehm, nicht wissen zu müssen, ob ein Befehl ein Makro oder eine Funktion ist – insbesondere wenn ein Makro nur aus Optimierungsgründen existiert, obwohl eine Funktion an der gleichen Stelle auch ginge. Andererseits verschleiern Makros die Semantik. Wenn ein Programmierer daher sorglos ein Makro verwendet im Glauben, es handele sich um eine Funktion, könnten unerwartete Fehler wie Namenskonflikte auftreten.

3.7 XSL

Die Extensible Stylesheet Language (XSL) [8] ist eine Sprachfamilie, mit der XML-Dokumente unter anderem in graphische Repräsentationen überführt werden können.

XSL besteht aus drei XML-basierten Sprachen:

- XSL - Formatting Objects (XSL-FO) zur Beschreibung des graphischen Layouts von Seiten
- XML Path Language (XPath) zum Adressieren von Elementen in XML-Dokumenten
- XSL Transformation (XSLT) als Transformationssprache für XML-Dokumente

XSL-FO spielt für die abstrakte Betrachtung von Code-Generierung keine Rolle und wird hier nicht näher behandelt.

Mit XPath können komfortabel Knotenmengen in XML-Bäumen referenziert werden. Beispielsweise selektiert der Ausdruck `//chapter[@title="Einleitung"]/paragraph` alle Elemente mit Namen `paragraph`, die direkte Kinder eines Elements mit Namen `chapter` und dem Attribut `title` mit Wert `Einleitung` sind, wobei der Knoten `chapter` an einer beliebigen Stelle im Baum stehen darf.

XSLT ist eine Template-basierte Transformationssprache, mit der ein XML-Dokument in ein anderes überführt werden kann – wobei reiner ASCII-Text als „einelementiger XML-Baum“ ebenfalls erzeugt werden kann. XSLT ist Turing-vollständig und relativ umfangreich – so sind unter anderem Konstrukte zur Iteration, Sortierung und bedingten Ausführung enthalten.

Zur Unterscheidung von Metasprache und generiertem Code wird der Namespace `xsl` benutzt, der somit als Unquote fungiert. Templates können sowohl direkt als Funktionen aufgerufen werden als auch als Makros, die über Pfad-Matching dem aktuell zu verarbeitenden Knoten zugeteilt werden. Ein direkter Aufruf erfolgt mit `<xsl:call-template>`, die alternative Kontrollübergabe an den Makro-Prozessor mit `<xsl:apply-templates/>`. Beides kann allerdings nicht derart kombiniert werden, dass ein funktional aufgerufenes Template dennoch über Matcher überprüft ob es auf einen Knoten wirklich anwendbar ist.

Abb. 3.5 zeigt ein Template, das eine eigene Notation für Aufzählungslisten in HTML übersetzt.

XSLT ist zwar so mächtig wie Programmiersprachen, allerdings aufgrund der XML-Syntax mit deutlich mehr Schreibaufwand verbunden, so dass für komplexe Berechnungen oft auf eine andere Sprache zurückgegriffen wird. Einige XSLT-Prozessoren bieten dafür die Möglichkeit, externe Programme aus XSLT-Templates heraus aufzurufen. Eine Komponententrennung ist ohne diese Möglichkeit ebenfalls umständlich. Aufgrund der Übereinstimmung von Meta- und Zielsprache ist eine Generierung von XSLT-Dateien prinzipiell möglich. Ein Problem ist allerdings wieder die Zuordnung der Unquotes – um beispielsweise ein `<xsl:apply-templates/>` zu generieren, muss dieses aufwändig entweder in einem CDATA-Abschnitt quotiert oder über das nachträgliche Ändern des Element-Namespaces erzeugt werden.

```
Eingabe:
<list>
  <item>foo</item>
  <item>bar</item>
  <item>baz</item>
</list>

Template:
<xsl:template match="/list">
  <ul>
    <xsl:for-each select="item">
      <li><xsl:value-of select="." /></li>
    </xsl:for-each>
  </ul>
</xsl:template>

Ausgabe:
<ul>
  <li>foo</li>
  <li>bar</li>
  <li>baz</li>
</ul>
```

Abbildung 3.5: Erzeugung einer Aufzählungsliste mit XSLT

3.8 Lisp

Lisp ist eine der ältesten Programmiersprachfamilien der Welt, die unter anderem für ihre Möglichkeiten zur Metaprogrammierung bekannt ist. Neben alten, aber noch immer benutzten Sprachen wie Emacs Lisp, Scheme oder Common Lisp entstehen auch immer wieder neue Lisps wie Clojure [22], welches auf der Java Virtual Machine (JVM) aufsetzt. Alle Lisp-Sprachen basieren auf S-Expressions und verzichten auf die meiste Syntax wie Infix-Operatoren. Zudem sind S-Expressions selbst in jedem Lisp als Datenstrukturen verfügbar, so dass jedes Programm gleichzeitig ein Lisp-Objekt ist und entsprechend manipuliert werden kann[4]. Diese Eigenschaft sowie die Einfachheit von S-Expressions selbst machen Lisp für Metaprogrammierung sehr geeignet. Mit der `eval`-Funktion unterstützt Lisp auch Laufzeit-Metaprogrammierung, auf die hier nicht näher eingegangen wird. Denn üblicher ist inzwischen die Benutzung von Makros für generative Metaprogrammierung. Im Folgenden beziehen sich alle Erklärungen auf die Sprache Common Lisp; andere Dialekte haben aber meist ähnliche Konstrukte.

```
Einfaches Quasiquote ohne Unquote - ergibt (+ 1 2 3):

 `(+ 1 2 3)


Quasiquote mit Unquote - ergibt (+ 1 7 3) falls x=2:

`(+ 1 ,(+ x 5) 3)


Unquote mit Splicing - ergibt (+ 1 2 3 4 5) falls input = (2 3 4):

`(+ 1 ,@input 5)
```

Abbildung 3.6: Drei Beispiele für Quasiquotes in Common Lisp

Makros werden mittels `(defmacro ..)` definiert und sind letztlich Funktionen, die Code als Parameter bekommen und Code (beides als S-Expressions) zurückliefern. Der Rückgabewert kann sowohl programmatisch als auch mittels Quasiquote erzeugt werden. Neben dem normalen Unquote gibt es auch ein Unquote mit „Splicing“, welches alle Elemente einer Liste einzeln einfügt. Abb. 3.6 zeigt drei kurze Verwendungsbeispiele, die alle Code zur Addition mehrerer Zahlen[5] generieren.

[4] Ein C-Programm kann zwar auch ein anderes C-Programm als String manipulieren – dies ist allerdings mit S-Expressions, die eine innerer Struktur besitzen, deutlich einfacher als die Arbeit mit rohen Strings.

[5] In Lisp ist der Operator `+` nicht binär, sondern akzeptiert beliebig viele Argumente. Dies ist ein Vorteil der Verwendung von S-Expression-Syntax anstelle von Infix-Operatoren – auch wenn dies zunächst gewöhnungsbedürftig ist.

Mit diesen Operationen können beliebige einzelne S-Expressions erzeugt werden. Da das Template selbst aus Lisp-Code besteht und auch beliebigen anderen Lisp-Code aufrufen kann, ist die Metasprache mächtig, erweiterbar sowie modularisierbar und benötigt vergleichsweise kurze Metaprogramme.

Lisp-Makros können nicht direkt aufgerufen werden, stattdessen wird ein S-Expression-Knoten ähnlich wie bei C nach seinem Knotennamen zu einem Makro zugeordnet. Es wird also bei jedem Expansionsschritt global geprüft, ob ein Makro so benannt ist wie das erste Element des zu expandierenden S-Expression. Ist dies der Fall, werden die restlichen Elemente als Parameter an das Makro übergeben, woraufhin der vom Makro zurückgegebene S-Expression erneut expandiert wird. Lässt sich kein Makro zu einem Ausdruck zuordnen, werden alle Elemente davon einzeln expandiert und der resultierende S-Expression als Funktionsaufruf interpretiert. Hier stellt sich wieder die Frage, ob die äußere Ununterscheidbarkeit von Makros und Funktionsaufrufen eher positiv oder negativ zu sehen ist. Positiv ist, dass die konkrete Implementation einer S-Expression-Sprache so versteckt wird und nachträglich verändert werden kann. Negativ sind unerwünschtes Verhalten wie Mehrfachevaluation oder Namenskonflikte – im Gegensatz zu C stellt Lisp allerdings Möglichkeiten bereit, um diese zu vermeiden. Beispielsweise kann mit dem `gensym`-Befehl ein global eindeutiger Variablenname erzeugt werden.

Als Beispiel soll ein Operator `if-not` definiert werden, der genau andersherum funktioniert als ein normales `if`. In einer Sprache wie Java wäre dies überhaupt nicht möglich – in Lisp reicht ein Zweizeiler:

```
(defmacro if-not (condition false-part &optional true-part)
  `(if ,condition ,true-part ,false-part))

; Aufruf
(if-not (= 4 5) "ungleich" "gleich")  ; ergibt "ungleich"
(if-not (= 1 1) "ungleich")           ; ergibt nil
```

Lisp-Makros können immer nur genau einen Ausdruck zurückliefern, weshalb auch das Quasiquote nur einen S-Expression erzeugen kann. Sind mehrere Rückgaben erforderlich, müssen diese in ein `progn` geschachtelt werden, welches den Lisp-Interpreter[6] anweist, mehrere Befehle nacheinander auszuführen. Beispielsweise definiert `define-arithmetic-functions` zwei Funktionen auf einmal:

```
(defmacro define-arithmetic-functions ()
  `(progn (defun add (x y) (+ x y))
          (defun sub (x y) (- x y))))


; Aufruf
(define-arithmetic-functions)
```

Makros können als kleine „Compiler-Plugins“ aufgefasst werden, mit denen die Sprache direkt erweitert oder DSLs integriert werden können. Eine der beeindruckenden

[6] Es kann sich selbstverständlich auch um einen Compiler handeln.

```
Definition eines "Meta-Makros":

(defmacro metamacro (name cmd)
  `(defmacro ,name (x)
     `(,',cmd ,x)))

Beispiel-Aufruf:

(metamacro mymacro print)

Wird kompiliert nach:

(defmacro mymacro (x)
  `(print ,x))
```

Abbildung 3.7: Ein Makro-erzeugendes Makro in Common Lisp

Anwendungen von Lisp-Makros ist ein nach Lisp kompilierendes Prolog [35, S.388ff], welches beliebig mit Lisp-Code gemischt werden kann. Auch Makro-schreibende Makros sind in Lisp möglich – hier ergibt sich allerdings wieder das Problem, ob sich ein Unquote auf das innere oder äußere Quasiquote bezieht. In Common Lisp bezieht sich ein Unquote immer auf das innerste Quasiquote. Die Aufhebung eines äußeren Quasiquote ist dagegen sehr trickreich und gewöhnungsbedürftig, wie Abb. 3.7 zeigt. Der Lösungsansatz benutzt zwischen zwei Unquotes ein Quote (Syntax ’), welches nicht mit diesen interagiert.

Da ausschließlich S-Expressions als Rückgabe von Makros möglich sind, kann mit dem Lisp-Makrosystem zunächst kein Code anderer Programmiersprachen generiert werden. Allerdings ist es möglich, mit Makros ein Lisp-Programm zu erzeugen, welches dann den Ziel-Code generiert.

3.9 Zusammenfassung und Diskussion

In diesem Kapitel wurde eine Reihe von Template-basierten Code-Generatoren miteinander verglichen. Abb. 3.8 zeigt eine tabellarische Übersicht dieses Vergleichs.

String-basierte Systeme sind vermutlich am einfachsten in der Benutzung und unterstützen direkt ohne Zusatzaufwand jede Zielsprache. Andererseits sind sie für die Vermeidung von Syntaxfehlern ungeeignet. Während PHP eine mächtige Metasprache bietet, erzwingt StringTemplate eine starke Komponententrennung durch Einschränkung eben dieser. C-Makros besitzen nur wenige Features, bieten aber als Makro-System direkte, von Funktionen äußerlich ununterscheidbare Spracherweiterungen – auch wenn diese Gefahren wie Namenskonflikte mit sich bringen.

C++-Templates sowie Template Haskell erweitern die Compiler je einer Sprache um die Möglichkeit von Template-Aufrufen über eine spezielle Syntax. Während Metasprache

Werkzeug	PHP	C-Makros	StringTemplate	Templates in C++
Eingabemodell	Objekt	String	Objekt als String	Typname/Wert/Template
Eingabevalidierung	relativ stark	schwach	stark	stark
Ausgabemodell	String	String	String	Code für Funktion/Klasse
Ausgabevalidierung	minimal	minimal	minimal	unklar, da Compiler-intern
Zielsprache	beliebig	beliebig	beliebig	C++
Quasiquote	implizit	`#define`	implizit / `{args | ..}`	`template<args> ..`
Unquote	`<?..?>`	implizit	`$ .. $` oder `< .. >`	implizit
universelles Quasiquote	ja	nein	ja	nein
Code-Objekt verfügbar	String	nein	nein	nein
Sprachmächtigkeit	maximal	sehr gering	minimal	gering
Metaprogrammlänge	relativ kurz	kurz	sehr kurz	lang
Template-Aufruf	nur über Engine-Aufruf	Makro	`wert:template(args)`	`token<args>`
Komponententrennung	möglich	nein	stark	keine
Template-Generierung	aufwändig	nein	aufwändig	nein

Werkzeug	Template Haskell	XSLT	Lisp
Eingabemodell	Objekt	XML	S-Expression
Eingabevalidierung	stark	mittel	mittel
Ausgabemodell	Code als Objekt	XML	S-Expression
Ausgabevalidierung	stark	mittel	mittel
Zielsprache	Haskell	beliebig	Lisp
Quasiquote	implizit /`[| .. |]`	`<xsl:template> ..`	`‘( .. )`
Unquote	`$( .. )`	über `xsl`-Namespace	`, ..`
vollständiges Quasiquote	nein	ja	nur genau ein Ausdruck
Code-Objekt verfügbar	ja	nein	ja
Sprachmächtigkeit	maximal	maximal	maximal
Metaprogrammlänge	relativ kurz	sehr lang	kurz
Template-Aufruf	Funktion	Funktion / Makro	Makro
Komponententrennung	möglich	umständlich	möglich
Template-Generierung	nein	aufwändig	etwas kompliziert

Abbildung 3.8: Vergleich der Verarbeitungsprozesse von Werkzeugen zur Code-Generierung

und Einsatzmöglichkeiten in C++ recht eingeschränkt sind, besitzt Template Haskell die komplette Sprachmächtigkeit von Haskell, ergänzt um die Möglichkeit, durch Reifikation Metainformationen über Code derselben Datei abzufragen. Template Haskell erzeugt aufgrund der starken Typisierung garantiert Code, der frei von Syntaxfehlern ist. Dies hat allerdings zum Preis, dass nicht jedes Template durch die verschiedenen Quasiquotes ausdrückbar ist, was die Lesbarkeit des Meta-Codes senkt.

XSLT und Lisp arbeiten auf untypisierten Baumstrukturen und bieten so einen Kompromiss zwischen Flexibilität und Typsicherheit. In beiden Systemen ist die Metasprache selbst ebenfalls auf diese Weise repräsentiert, wodurch die Möglichkeit zur Generierung und somit auch Erweiterung der Metasprache selbst gegeben ist. Die Generierung von Quasiquotes bringt allerdings das generelle Problem der Zuordnung eines inneren Unquotes mit sich. In XML kann dieses nur umständlich erreicht werden, während die Lisp-Lösung sehr kurz, aber etwas kompliziert ist. Während XSLT neben XML auch beliebige ASCII-Formate ausgeben kann, beschränkt sich der Lisp-Makroprozessor auf S-Expressions. Beide Frameworks haben eine Turing-vollständige Metasprache. Komplexe Algorithmen lassen sich in XSLT allerdings aufgrund der umständlichen Syntax nur mühsam ausdrücken, so dass für diese meist auf eine externe Programmiersprache zurückgegriffen wird. Lisp-Programme dagegen sind oft kurz und elegant. Makro-gesteuerte Template-Aufrufe sind ebenfalls in beiden Systemen möglich, wobei XSLT komplexere Patterns sowie alternativ funktionale Makro-Aufrufe unterstützt. Ein weiterer Unterschied ist der Aufwand für eine neue Spracherweiterung: In XSLT muss dafür eine eigene Template-Datei mitsamt zusätzlichem Kompilationsaufruf angelegt werden. Lisp dagegen erlaubt sogar die direkte Mischung von Makros und Code, wodurch praktisch Compiler-Plugins mit minimalem Aufwand erzeugt werden können. Dadurch lassen sich in Lisp leicht mächtige, voll integrierte DSLs konstruieren.

Allgemein wächst der Aufwand für Code-Generierung mit der Komplexität des Ausgabemodells. Soll die Generierung typsicher sein, ist das Modell desto umfangreicher, je aufwändiger die Syntax der zu generierenden Sprache ist. Obwohl beispielsweise Haskell im Vergleich zu Sprachen wie C++ bereits eine sehr einfache Syntax besitzt, erweist sich diese für Template-Haskell bereits als Problem, da nicht alles auf den Quasiquote-Operator abgebildet werden kann. In einer String-basierten Generierung von Haskell-Code wäre die Flexibilität größer, so dass hier ein entsprechendes Quasiquote möglich gewesen wäre – zum Preis von möglichen Syntaxfehlern und verschleierter Struktur.

Ein Mittelweg wäre eine S-Expression-Syntax für Haskell, die vom Framework zunächst in eine typisierte Repräsentation überführt und daraufhin zur Generierung verwendet wird. Diese Variante hätte folgende Vorteile:

- Das Ausgabemodell von Templates ist einfach, aber explizit strukturiert.
- Quasiquotes können für jedes Template benutzt werden.
- Es kann kein Code mit Syntaxfehlern erzeugt werden, da das S-Expression-Modell bei der Umwandlung in ein typisiertes Objekt validiert wird.

Wenn nun standardmäßig die S-Expression-Notation verwendet wird, ist auch weiterhin

die Metasprache mit der Zielsprache identisch. Dieser Ansatz wird in Kapitel 10 weiter verfolgt.

4 Anforderungen

Grundsätzlich soll mit MagicL ein sprachunabhängiges Framework für generative Metaprogrammierung entwickelt werden, das eine an Lisp angelehnte, modulare und inkrementelle Compiler-Entwicklung durch kurzen und prägnanten Meta-Code unterstützt. Zudem sollten allgemein eine möglichst hohe Flexibilität und Einfachheit angestrebt werden.

Diese Grundanforderungen werden nun in Hinblick auf die Kriterien aus Kapitel 3 konkretisiert.

4.1 Modelle

Um sprachunabhängig zu sein, sollten beliebige Ein- und Ausgabesprachen wie XML oder Java unterstützt werden. Da diese nicht alle von vornherein Teil von MagicL selbst sein können, muss es dem Metaprogrammierer möglich sein, selbst entsprechende Parser oder Generatoren in MagicL zu definieren. Der Modelltyp String muss also in Form von Parser- und Generator-Bibliotheken hinreichend unterstützt werden. Andererseits sollten Zeichenketten nur ganz am Anfang oder Ende der Verarbeitungskette liegen und im Idealfall – wenn sprachspezifische Backends bereits bestehen – überhaupt nicht vom Metaprogrammierer berührt werden. Stattdessen sollten möglichst viele Verarbeitungsschritte strukturierte Modelle benutzen.

Für typisierte Objekte spricht das automatische Finden von Fehlern bereits während der Kompilation des Generators, wohingegen diese bei einer Verarbeitungskette mit untypisierten Zwischenmodellen erst im nächsten Schritt sichtbar werden – oder gar noch später, falls dieser unvorsichtig implementiert wurde. Für umfangreiche Berechnungen sind typisierte Datenstrukturen generell von Nöten. Modelldefinitionen mit Typsignaturen sind außerdem bereits zu einem gewissen Grad selbstdokumentierend.

S-Expressions hingegen sind allgemein flexibler und ersparen die Phase der Modellerstellung. Dies ist besonders dann von Vorteil, wenn eine bestehende Modellsprache nur geringfügig erweitert oder angepasst werden soll, so dass ein komplett neues, fast identisches Modell einen übertriebenen Arbeitsaufwand bedeuten würde. Auch können S-Expressions ohne weiteren Aufwand in Zeichenketten serialisiert und so zwischen verschiedenen Programmen oder über das Netzwerk kommuniziert werden. Zudem lassen sich Modelle in S-Expressions oft kürzer und weniger redundant notieren als in einer typisierten Form.

Da sowohl Objekte als auch S-Expressions in bestimmten Anwendungsfällen von Vorteil sind, sollte MagicL beide Formen unterstützen. Auch die einfache Umwandlung von einer Repräsentation in die andere sollte möglich sein. DSLs werden dann üblicherweise

in S-Expressions repräsentiert, Modelltypen für die weitere Verarbeitung können variiert werden.

Der verwendete S-Expression-Datentyp sowie die konkrete Syntax sollten zunächst möglichst einfach und domänenunabhängig gehalten werden, weshalb beispielsweise auf eine automatische Typisierung beim Einlesen von Zahlen verzichtet wird. Auch spezielle Syntax für Strings, Kommentare, Quasiquotes oder Leerzeichen in Symbolnamen wird zunächst nicht für nötig befunden, da auch dies alles bereits mittels primitiver S-Expressions repräsentiert werden kann. Für den Fall, dass besondere Syntax für einzelne Anwendungen dennoch erwünscht sein sollte, könnte diese beispielsweise über einen Präprozessor in die entsprechende S-Expression-Darstellung überführt werden. Eine weitere Überlegung ist, ob der S-Expression-Datentyp eine „kanonische Form“ erzwingen soll, bei der das erste Element eines Knotens immer der Knotenname ist, indem ein gesondertes Datenfeld für den Namen angelegt wird. Eine derartige Form wäre analog zu XML, wo der Name ebenfalls einzeln behandelt wird. Zudem wird die Semantik von S-Expressions dadurch klarer und eine automatische Visualisierung als Baum (oder mittels Einrückung) sinnvoller. Wird hier allerdings festgelegt, dass es sich bei diesen Namen ausschließlich um Symbole handeln muss, werden die möglichen Modellierungssprachen unnötig eingeschränkt. Beispielsweise könnte eine funktionale Programmiersprache einen S-Expression als Funktionsaufruf in Präfixnotation interpretieren und die Möglichkeit zulassen wollen, die aufgerufene Funktion selbst durch einen Funktionsaufruf zu erzeugen. Werden aber beliebige S-Expressions als Namen zugelassen, ist der „Missbrauch“ für nichtkanonische Strukturen dennoch möglich, wenn auch etwas aufwändiger. Generell wird die Verarbeitung von S-Expressions durch das Sonderfeld allerdings komplexer, weshalb die einfache Struktur bevorzugt wird: Ein S-Expression ist entweder

- ein Symbol oder
- eine Liste von S-Expressions.

Die Konvention, das erste Element als Knotennamen anzusehen, ist dennoch sinnvoll und wird von MagicL vorausgesetzt.

4.2 Templates

Templating in MagicL soll ähnlich aussehen wie in Lisp, d.h. S-Expression-basiert mittels Makros und Quasiquotierung. Wie in Abschnitt 3.8 diskutiert wurde, kann ein Lisp-Makro aber nur mehrere Ausdrücke gleichzeitig zurückgeben, indem es diese in ein `progn` schachtelt. Dies funktioniert aber nicht für Zielsprachen ohne ein Konstrukt wie `progn`. Daher sollten MagicL-Makros und der Quasiquote-Operator direkt mehrere Rückgaben ermöglichen, welche vom Makrosystem durch Splicing eingefügt werden.

4.3 Metaprogrammiersprache

Die bereitgestellte Metaprogrammiersprache sollte einfach und mächtig sein. Da MagicL in Haskell geschrieben wird, ist eine Verwendung von Haskell auch als Metasprache naheliegend. Allerdings ist Haskell nicht S-Expression-basiert, so dass es ohne weiteres nicht in reinen S-Expression-Templates verwendet werden kann. Aus diesem Grund muss mit S-Expression-Haskell eine S-Expression-Syntax für Haskell bereitgestellt werden, die automatisch nach Haskell kompiliert wird. Daher muss MagicL selbst bereits generative Metaprogrammierung betreiben.

4.4 Aufruf von Templates

Makros in MagicL sollen wie in Lisp implizit über den Knotennamen zugeordnet werden können. Allerdings wäre es interessant, wenn nicht nur der Name, sondern der gesamte Knoten in einer Art Pattern-Matching ähnlich XPath berücksichtigt werden könnte. Auch eine mit EBNF [23, S.43f] vergleichbare Grammatik auf S-Expressions wäre wünschenswert.

Der Kontrollfluss im Lisp-Makrosystem ist relativ unflexibel, da lediglich eine globale Zuordnung von Namen zu Makros besteht[1]. In konkreten Sprachen hingegen gibt es häufig Konstrukte, die nur im Kontext bestimmter Oberkonstrukte verwendet werden können. Lisp-Makros, die solche Oberkonstrukte verarbeiten, benutzen für die Unterkonstrukte keine weiteren Makros – sonst wären die Konstrukte auch außerhalb verfügbar – sondern zerlegen die inneren S-Expressions manuell. Praktischer wäre ein Konzept von „lokalen Makros“, die nur an gewünschten Stellen aufgerufen werden. Dies ist vergleichbar mit dem Aufruf von XSLT-Templates über Namen. Im Gegensatz zu diesem sollte ein lokal aufgerufenes Makro seine Anwendbarkeit aber auch durch Pattern-Matching überprüfen und im Falle eines Scheiterns den Kontrollfluss an das nächste Lokalmakro weiterreichen.

Damit dies alles möglich wird, muss der Begriff „Makro“ gegenüber Lisp sinnvoll und theoretisch fundiert verallgemeinert werden, wofür einige Grundlagenarbeit benötigt wird.

4.5 Komponententrennung

MagicL soll einen minimalen und modularen Entwurf von Compilern unterstützen, um maximale Wiederverwendbarkeit von Komponenten und minimalen Entwicklungsaufwand für neue DSLs zu ermöglichen. Daher müssen verschiedene Parser, Compiler oder Generatoren einfach hintereinandergeschaltet werden können, wofür eine allgemeine Schnittstelle benötigt wird.

[1] Es gibt in Lisp zwar die Möglichkeit, Makros mittels `macrolet` nur für einen bestimmten lexikalischen Skope verfügbar zu machen. Innerhalb dieses Bereichs ist das Makro dann allerdings nur Teil der globalen Liste, so dass ohne Weiteres keine feinere Steuerung möglich ist.

4.6 Generierung von Templates

Das Hauptproblem bei der Generierung von Templates aus Templates ist die Uneindeutigkeit bei der Zuordnung eines inneren Unquote. In Common Lisp müssen in diesem Fall daher komplizierte Konstrukte wie `,',` verwendet werden (siehe Abschnitt 3.8).

Eine einfache Alternative hierzu ist die Bereitstellung verschieden benannter Quote-Unquote-Paare, so dass sich nur noch eine Zuordnungsmöglichkeit innerer Unquotes ergibt[2]. Dies soll MagicL realisieren.

4.7 Zusammenfassung

Werkzeug	**MagicL**	**Lisp**
Eingabemodell	S-Expression / beliebig	S-Expression
Eingabevalidierung	beliebig	mittel
Ausgabemodell	S-Expression / beliebig	S-Expression
Ausgabevalidierung	beliebig	mittel
Zielsprache	beliebig	Lisp
vollständiges Quasiquote	ja	nur genau ein Ausdruck
Code-Objekt verfügbar	ja	ja
Sprachmächtigkeit	maximal	maximal
Metaprogrammlänge	kurz	kurz
Template-Aufruf	verallgemeinertes Makro	Makro
Komponententrennung	unterstützt	möglich
Template-Generierung	einfach	etwas kompliziert

Abbildung 4.1: Anforderungen an MagicL im Vergleich zu Lisp

Das zu entwickelnde Code-Generierungsframework soll eine makrobasierte, baukastenartige Konstruktion von Compilern ähnlich Lisp ermöglichen, dabei allerdings beliebige Zielsprachen unterstützen. Alle drei erwähnten Modelltypen müssen in MagicL unterstützt werden: Interne DSLs und Templating sollten ausschließlich auf S-Expressions basieren. Externe Formate müssen dank integrierter Parser- und Generatorbibliotheken für Zeichenketten einfach eingelesen und erzeugt werden können. Für komplexe Algorithmen sollten typisierte Objekte verwendbar sein. Diese verschiedenen Komponenten müssen durch eine allgemeine Schnittstelle einfach hintereinandergeschaltet werden können.

Damit keine zusätzliche Syntax zu reinen S-Expressions erforderlich wird, muss die Syntax von Quasiquote und Unquote gegenüber Lisp leicht verändert werde. Zudem soll es möglich sein, eine beliebige Anzahl S-Expression zu quotieren, für die das Makrosystem automatisch ein Splicing vornimmt. Darüber hinaus sollten mehrere verschieden

[2]Eine ähnliche Erweiterung hat Drew McDermott für Common Lisp implementiert [33, S.21ff].

benannte Quote-Unquote-Paare bereitgestellt werden, um Template-generierende Templates zu vereinfachen. Als Metasprache soll eine S-Expression-Version von Haskell und damit eine vollständige Programmiersprache angeboten werden.

Dazu soll der Makro-Begriff gegenüber Lisp verallgemeinert werden, so dass die Zuordnung zu einem Makro nicht nur über den Knotennamen, sondern durch beliebiges Pattern-Matching oder gar eine EBNF-ähnliche Grammatik erfolgen kann. Auch soll es möglich sein, lokale Makros zu definieren, die nur im Kontext eines anderen Makros aktiv sind.

Die verallgemeinerten Makros sollten hinreichend theoretisch fundiert werden. Dies wird durch eine Einbettung in die Kategorientheorie erreicht, welche im folgenden Kapitel zunächst vorgestellt wird.

5 Kategorientheorie

Die Kategorientheorie ist ein sehr abstrakter Zweig der Mathematik, aber auch ein universeller, denn fast alle wichtigen mathematischen Strukturen bilden Kategorien. Kategorientheorie spielt dank ihrer Allgemeinheit und Anwendbarkeit für Berechnungen auch in der Informatik eine immer größer werdende Rolle, da sie unter anderem eine generische Schnittstelle für die Verschaltung beliebiger Verarbeitungsprozesse liefert. Hierfür wird der Kompositionsoperator von Funktionen auf beliebige Datentypen, welche Prozesse beschreiben, verallgemeinert.

Die Sprache Haskell benutzt kategorientheoretische Konzepte, um Nebeneffekte elegant in eine ansonsten pur funktionale Sprache zu integrieren. Die Architektur von MagicL basiert komplett auf der Haskell-Umsetzung kategorientheoretischer Konstrukte. So werden Parser als zusammengesetzte Funktoren konstruiert. Auf diese Weise wird eine hohe Flexibilität, Eleganz, Modularität und Wiederverwendbarkeit von Komponenten wie Parsern, Compilern oder Makros erreicht.

Dieser Abschnitt führt in die Kategorientheorie ein und stellt alle später verwendeten Konzepte vor. Die Definitionen sind weitgehend aus [14] übernommen.

5.1 Einführung

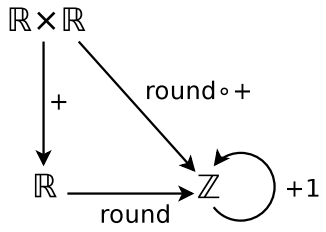


Abbildung 5.1: Ein simples Funktionensystem

Kategorientheorie lässt sich verstehen als Verallgemeinerung der Art und Weise, wie mit Funktionstypen insbesondere hinsichtlich der Komposition gerechnet wird. Abb. 5.1 zeigt ein kleines Funktionensystem zwischen den Mengen $\mathbb{R} \times \mathbb{R}$, $\mathbb{R}$ und $\mathbb{Z}$. Es kommen folgende Funktionen vor:

- $+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ addiert zwei reelle Zahlen.
- round : $\mathbb{R} \to \mathbb{Z}$ ist die Rundungsfunktion.

- $+1 : \mathbb{Z} \to \mathbb{Z}$ addiert 1 zu einer ganzen Zahl.
- $\text{round} \circ + : \mathbb{R} \times \mathbb{R} \to \mathbb{Z}$ addiert zwei reelle Zahlen und rundet anschließend.

Die Komposition round $\circ$ + (gesprochen „round *nach* +“) lässt sich bilden, weil der Definitionsbereich von round mit dem Zielbereich von + übereinstimmt (nämlich $\mathbb{R}$). Es lassen sich also zwei im Diagramm aufeinander folgende Pfeile zu einem direkten zusammenfassen. Die Kategorientheorie „vergisst“ nun den Bezug zu Mengen und Funktionen und arbeitet stattdessen mit Objekten und Morphismen, welche vollkommen abstrakt definiert sind und mit bestimmten Operationen eine Kategorie bilden.

5.2 Kategorien

Definition 1 (Kategorie) *Eine Kategorie* $\mathbf{C} = (\text{Ob}^{\mathbf{C}}, \text{Mor}^{\mathbf{C}}, \circ^{\mathbf{C}}, \text{id}^{\mathbf{C}})$ *ist gegeben durch*

- *eine Klasse* $\text{Ob}^{\mathbf{C}}$ *von Objekten*[1].
- *eine Menge von Morphismen* $\text{Mor}^{\mathbf{C}}_{A,B}$ *für alle* $A, B \in \text{Ob}^{\mathbf{C}}$*, wobei* A *und* B *Domäne und Codomäne genannt werden,*
- *einen Kompositionsoperator* $\circ^{\mathbf{C}}_{A,B,C}$ *für alle* $A, B, C \in \text{Ob}^{\mathbf{C}}$ *mit* $\circ^{\mathbf{C}}_{A,B,C} : \text{Mor}^{\mathbf{C}}_{B,C} \times \text{Mor}^{\mathbf{C}}_{A,B} \to \text{Mor}^{\mathbf{C}}_{A,C}$*,*
- *eine Identität* $\text{id}^{\mathbf{C}}_A \in \text{Mor}^{\mathbf{C}}_{A,A}$ *für alle* $A \in \text{Ob}^{\mathbf{C}}$*,*

wobei folgende Axiome erfüllt sein müssen:

- *Neutralität der Identität:*

$$f \circ_{A,A,B} \text{id}_A = \text{id}_B \circ_{A,B,B} f = f$$

- *Assoziativität:*

$$f \circ_{A,C,D} (g \circ_{A,B,C} h) = (f \circ_{B,C,D} g) \circ_{A,B,D} h$$

Indizes und Kategorien werden der Einfachheit halber weggelassen, wenn sie aus dem Kontext hervorgehen. Wie von Funktionen gewohnt, lässt sich $f \in \text{Mor}_{A,B}$ auch schreiben als $f : A \to B$. Sind A und B identisch, bezeichnet man f auch als Endomorphismus.

Wie das Einführungsbeispiel erwarten lässt, bilden Mengen und darauf definierte Funktionen Kategorien. Die Kategorie, die alle denkbar möglichen Mengen und Funktionen enthält, heißt **Set**. Das Einführungsbeispiel skizziert somit unvollständig eine Unterkategorie von **Set**. Es gibt allerdings durchaus auch andere Kategorien – beispielsweise können alle Mengen einer Potenzmenge als Objekte und die Teilmengenbeziehungen als Morphismen benutzt werden. Abb. 5.2 zeigt solch eine Potenzmengenkategorie über $\{1, 2, 3\}$, wobei die Kompositionen nicht mit eingezeichnet sind. Im Folgenden wird die konkrete Bedeutung einer Kategorie nicht näher betrachtet.

[1] Da Objekte selbst Mengen sein können, wäre eine Definition von $\text{Ob}^{\mathbf{C}}$ als Menge problematisch.

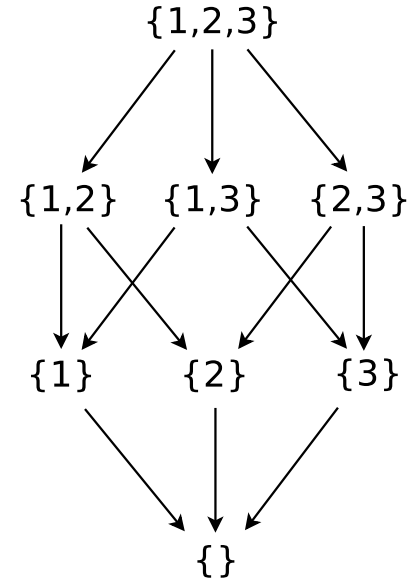


Abbildung 5.2: Eine von einer Potenzmenge gebildete Kategorie

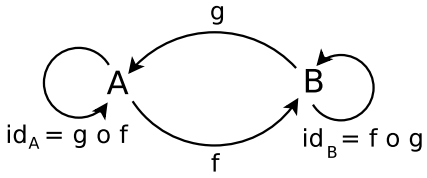


Abbildung 5.3: Eine einfache, vollständige Kategorie

Abb. 5.3 zeigt eine simple, vollständig eingezeichnete Beispielkategorie **C**, die aus den Objekten A, B und den Morphismen $f, g, \mathrm{id}_A, \mathrm{id}_B$ besteht. Alle Kompositionen davon sind wieder einer dieser vier Morphismen, ansonsten wäre die Kategorie nicht bezüglich Komposition abgeschlossen (und damit keine Kategorie).

Wie in diesen Beispielen lassen sich Kategorien in Diagrammen visualisieren, indem sie wie Multigraphen, also Graphen mit mehr als einer möglichen Kante zwischen zwei Knoten, gezeichnet werden. Formal sind allerdings nicht alle Kategorien Multigraphen, da letztere eine Menge von Knoten, Kategorien jedoch eine Klasse von Objekten besitzen.

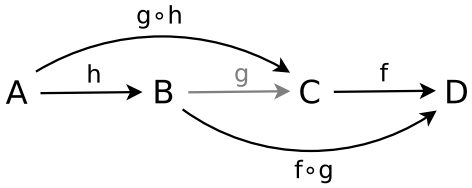


Abbildung 5.4: Das Assoziativitätsaxiom als kommutatives Diagramm

Ergeben alle in einem Diagramm möglichen Pfade zwischen je zwei Objekten denselben

Morphismus, wird das Diagramm als kommutativ bezeichnet:

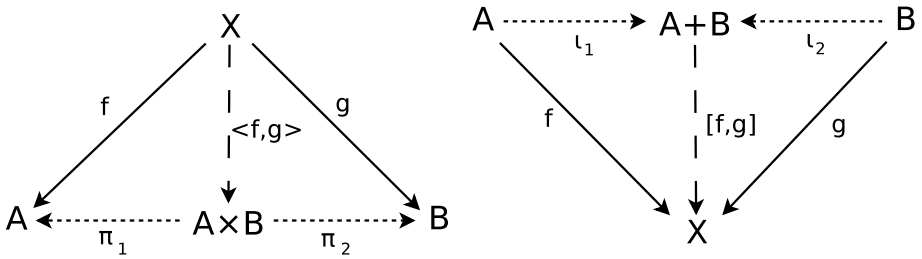
Definition 2 (Kommutatives Diagramm) *Ein Diagramm kommutiert, wenn für je zwei eingezeichnete Pfade $f = f_1 \circ \cdots \circ f_n : A \to B$ und $g = g_1 \circ \cdots \circ g_m : A \to B$ zwischen zwei Objekten A und B die Gleichung $f = g$ gilt.*

Beispielsweise zeigt Abb. 5.4 das Assoziativitätsaxiom als kommutatives Diagramm. g ist nur der Vollständigkeit halber eingezeichnet und spielt für das kommutative Diagramm selbst keine Rolle. Dieses Diagramm dient der Veranschaulichung, kann aber nicht für die Definition der Assoziativität genutzt werden – schließlich setzt die Definition kommutativer Diagramme bereits Assoziativität von $\circ$ voraus.

5.3 Produkte und Coprodukte

Die Kategorientheorie schafft es, Begriffe für Funktionseigenschaften wie Injektivität sowie Mengenoperationen wie das kartesische Produkt auf Kategorien zu verallgemeinern, indem sich die neuen Definitionen ausschließlich auf Objekte und Morphismen beziehen und nicht von Mengen oder Funktionen im speziellen Gebrauch machen. In der Kategorie **Set** stimmen die neuen Begriffe dann genau mit den alten überein. Dem kartesischen Produkt entspricht das Produkt von Objekten:

Definition 3 (Produkt) *Ein Objekt $A \times B$ mit zwei Projektionsmorphismen $\pi_1 : A \times B \to A$ und $\pi_2 : A \times B \to B$ ist ein Produkt von A und B, wenn für jedes $X \in \mathrm{Ob}$ sowie für alle $f : X \to A$ und alle $g : X \to B$ genau ein Morphismus $\langle f, g \rangle : X \to A \times B$ existiert, für den Abb. 5.5a kommutiert, d.h. $\pi_1 \circ \langle f, g \rangle = f$ und $\pi_2 \circ \langle f, g \rangle = g$ gelten.*



(a) Produkt (b) Coprodukt

Abbildung 5.5: Kommutative Diagramme zur Definition von Produkt und Coprodukt

Man kann leicht nachvollziehen, dass das kartesische Produkt genau ein Produkt für die Kategorie **Set** ist. $\langle f, g \rangle$ ist hier die Funktion, die x auf das Tupel $(f(x), g(x))$ abbildet.

Dreht man im Diagramm alle Pfeile um[2], erhält man die Definition für das Coprodukt:

[2]Formal wird durch das „Umdrehen“ aller Morphismen in $\mathbf{C}$ die duale Kategorie $\mathbf{C}^{\mathrm{op}}$ erzeugt. Entsprechend ist das Coprodukt der zum Produkt duale Operator.

Definition 4 (Coprodukt) *Ein Objekt $A+B$ mit zwei Injektionsmorphismen $\iota_1 : A \to A + B$ und $\iota_2 : B \to A + B$ ist ein Coprodukt von A und B, wenn für jedes $X \in \mathrm{Ob}$ sowie alle $f : A \to X$ und alle $g : B \to X$ genau ein Morphismus $[f, g] : A + B \to X$ existiert, für den Abb. 5.5b kommutiert, d.h. $[f, g] \circ \iota_1 = f$ und $[f, g] \circ \iota_2 = g$ gelten.*

Das Coprodukt entspricht einer disjunkten Vereinigung von Mengen, also einer Vereinigung von Mengen, die vorher explizit disjunkt gemacht werden (sofern sie es nicht bereits sind). Dies kann beispielsweise durch die Indizes L und R geschehen: $\{1, 2, 3\} + \{2, 3, 4\} = \{1_L, 2_L, 3_L, 2_R, 3_R, 4_R\}$. Der Morphismus $[f, g]$ entspricht einer Fallunterscheidung: Auf Elemente aus A wird f angewendet, auf welche aus B entsprechend g.

5.4 Funktoren

Strukturerhaltende Abbildungen zwischen Objekten und Morphismen zweier Kategorien werden Funktoren genannt:

Definition 5 (Funktor) *Ein Funktor $F = (F_{\mathrm{Ob}}, F_{\mathrm{Mor}}) : \mathbf{C} \to \mathbf{D}$ von Kategorie C nach Kategorie D*

- *bildet jedes Objekt $A \in \mathrm{Ob}^{\mathbf{C}}$ auf $F_{\mathrm{Ob}}(A) \in \mathrm{Ob}^{\mathbf{D}}$ ab,*
- *bildet jeden Morphismus $f \in \mathrm{Mor}^{\mathbf{C}}_{A,B}$ auf $F_{\mathrm{Mor}}(f) \in \mathrm{Mor}^{\mathbf{D}}_{F_{\mathrm{Ob}}(A),F_{\mathrm{Ob}}(B)}$ ab,*

wobei für alle $A, B, C \in \mathrm{Ob}^{\mathbf{C}}$ und alle $f \in \mathrm{Mor}^{\mathbf{C}}_{\mathrm{B,C}}, g \in \mathrm{Mor}^{\mathbf{C}}_{\mathrm{A,B}}$ folgende Axiome erfüllt sein müssen:

- *Erhaltung der Komposition:*

$$F_{\mathrm{Mor}}(f \circ^{\mathbf{C}} g) = F_{\mathrm{Mor}}(f) \circ^{\mathbf{D}} F_{\mathrm{Mor}}(g)$$

- *Erhaltung der Identität:*

$$F_{\mathrm{Mor}}(\mathrm{id}^{\mathbf{C}}_A) = \mathrm{id}^{\mathbf{D}}_{F_{\mathrm{Ob}}(A)}$$

Statt F_{Ob} und F_{Mor} kann einfach F geschrieben werden, wenn aus dem Kontext hervorgeht welche Abbildung gemeint ist.

Kategorien als Objekte und Funktoren als Morphismen ergeben selbst wieder eine Kategorie – die Kategorie der kleinen Kategorien. Kleine Kategorien sind Kategorien, deren Klasse von Objekten eine Menge ist. Diese Einschränkung ist nötig, da Klassen von Klassen in der Mathematik ähnlich problematisch sind wie Mengen von Mengen.

5.5 Natürliche Transformationen

Eine andere Möglichkeit aus Funktoren eine Kategorie zu bilden, besteht darin, diese als Objekte zu benutzen. Alle Funktoren $\mathbf{C} \to \mathbf{D}$ als Objekte bilden die sogenannte Funktorkategorie über $\mathbf{C}$ und $\mathbf{D}$, deren Morphismen natürliche Transformationen heißen.

Definition 6 (Natürliche Transformation) *Eine natürliche Transformation $\alpha\colon F \Rightarrow G$ ordnet jedem Objekt A aus $\mathbf{C}$ einen Morphismus $\alpha_A : F(A) \to G(A)$ aus $\mathbf{D}$ zu, wobei für alle Objekte A, B und alle Morphismen $f : A \to B$ aus $\mathbf{C}$ die Gleichung*

$$\alpha_B \circ F(f) = G(f) \circ \alpha_A$$

gelten muss, was dem kommutativen Diagramm in Abb. 5.6 entspricht.

Obwohl natürliche Transformationen in der Funktorkategorie Morphismen sind, werden sie für die bessere Unterscheidung mit $\Rightarrow$ statt $\to$ notiert.

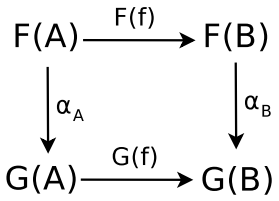


Abbildung 5.6: Kommutatives Diagramm in Kategorie $\mathbf{D}$ für die Definition von natürlichen Transformationen

5.6 Monaden

Monaden setzen sich aus einem Endofunktor und zwei natürlichen Transformationen zusammen:

Definition 7 (Monade) *Eine Monade (T, η, μ) über der Kategorie $\mathbf{C}$ besteht aus*

- *einem Endofunktor $T : \mathbf{C} \to \mathbf{C}$,*
- *einer natürlichen Transformation $\eta : \mathrm{id}_{\mathbf{C}} \Rightarrow T$,*
- *einer natürlichen Transformation $\mu : T^2 \Rightarrow T$,*

wobei für jedes Objekt A folgende Axiome erfüllt sein müssen:

- *Assoziativität:*

$$\mu_A \circ T(\mu_A) = \mu_A \circ \mu_{T(A)}$$

- *Neutrales Element:*
$$\mu_A \circ T(\eta_A) = \mu_A \circ \eta_{T(A)} = \mathrm{id}_{T(A)}$$

$\mathrm{id}_{\mathbf{C}} : \mathbf{C} \to \mathbf{C}$ bezeichnet hier den Identitätsfunktor, $\mathrm{id}_{T(A)} : T(A) \to T(A)$ dagegen den Identitätsmorphismus.

Das erste Axiom beschreibt zwei verschiedene Möglichkeiten, einen Morphismus von $T(T(T(A)))$ nach $T(A)$ zu bilden: $T(\mu_A) : T(T(T(A))) \to T(T(A))$ auf der linken Seite behält das äußere T bei und verwendet μ, um das innen stehende $T(T(A))$ in $T(A)$ zu überführen. Rechts reduziert $\mu_{T(x)} : T(T(T(A))) \to T(T(A))$ dagegen von außen und lässt das innere $T(A)$ unangerührt. Wird das Ergebnis dann in ein μ geschachtelt, sind innere und äußere Reduktion ununterscheidbar. Die Gleichung lässt sich auch kurz als $\mu \circ T\mu = \mu \circ \mu T$ schreiben.

Das zweite Axiom funktioniert ähnlich, nur beschreiben diesmal beide Seiten eine Art den Identitätsmorphismus für das Objekt $T(A)$ zu bilden. $T(\eta_A) : T(A) \to T(T(A)$ fügt das neue T innen ein, $\eta_{T(A)} : T(A) \to T(T(A))$ hingegen außen.

Zu jeder Monade lässt sich eine neue Kategorie bilden, die Kleisli-Kategorie:

Definition 8 (Kleisli-Kategorie) *Die Kleisli-Kategorie $\mathbf{C}_K$ zur Kategorie C und der Monade (T, η, μ) besteht aus*

- *den Objekten von $\mathbf{C}$,*
- *den Morphismen $f^{\mathbf{C}} \in \mathrm{Mor}^{\mathbf{C}}_{A,T(B)}$ aus C, die in $f \in \mathrm{Mor}^{\mathbf{C}_K}_{A,B}$ umbenannt werden,*
- *der Identität $\mathrm{id}_A = \eta_A$,*
- *der Komposition $f \circ^{\mathbf{C}_K}_{A,B,C} g = \mu_C \circ^{\mathbf{C}} T(f) \circ^{\mathbf{C}} g$.*

Die Erfüllung der Kategorieaxiome wird hier nicht gezeigt, folgt aber aus den Axiomen für Funktoren, natürliche Transformationen und Monaden.

Monaden sind in der Programmierung nützlich, da sie generisch das Rechnen mit „eingepackten“ Werten beschreiben. Beispielsweise könnte A für den Typ `Int` und $T(A)$ für `List of Int` stehen. η beschreibt dann die Erzeugung einer einelementigen Liste, μ reduziert eine Liste von Listen auf eine flache Liste. Über die Kleisli-Kategorie lassen sich damit beispielsweise nichtdeterministische Funktionen, die eine Liste möglicher Resultate zurückliefern, elegant verknüpfen. Haskell (siehe Kapitel 6) benutzt Monaden unter anderem, um Seiteneffekte in eine normalerweise pur funktionale Sprache einzubauen, wie Abschnitt 6.4.4 erläutert.

5.7 Zusammenfassung

Kategorientheorie ist ein abstrakter Zweig der Mathematik, der sich mit Morphismen (Pfeilen) zwischen Objekten (Knoten) und insbesondere der Komposition von Morphismen befasst. Damit können unter anderem Verarbeitungsprozesse formal und graphisch darstellbar beschrieben werden, wobei die Möglichkeiten weit über mathematische Funktionen hinausgehen. Über Produkte und Coprodukte können kartesische Produkte und

disjunkte Vereinigungen von Mengen auf Objekte in Kategorien verallgemeinert werden. Mit Funktoren lassen sich Abbildungen zwischen Kategorien beschreiben. Eine besondere Art von Funktoren sind Monaden, welche unter anderem „eingepackte“ Werte wie Listen beschreiben können. Zu jeder Monade kann wiederum eine Kategorie gebildet werden, die Kleisli-Kategorie.

Der Nutzen der Kategorientheorie für diese Arbeit besteht darin, allgemein und abstrakt die Kombination von Verarbeitungsschritten beschreiben zu können und somit ein formales Rahmenwerk für die Zusammensetzung von Compilern zu liefern. Konkret zeigt sich dies in der Umsetzung kategorientheoretischer Konzepte in Haskell.

6 Die Sprache Haskell

Haskell ist eine funktionale Programmiersprache, die generell sehr mathematisch inspiriert ist. So sind Funktionen hier grundsätzlich pur, d.h. ohne Nebeneffekte. Dank der Haskell-Umsetzung von Monaden sind Berechnungen mit Nebeneffekten dennoch möglich (siehe Abschnitte 5.6 und 6.4.4). Auch objektorientierte Konzepte gibt es hier in der Form nicht – stattdessen wird mittels algebraischer Datentypen und Typklassen modelliert. Unterstützt wird dieser Ansatz durch ein statisches Typsystem, welches praktische Features wie Typinferenz oder abhängige Typen bereitstellt. Haskell arbeitet mit Lazy Evaluation, daher werden Funktionen und Datenstrukturen erst dann ausgewertet, wenn die Daten wirklich benötigt werden. Dies ermöglicht unter anderem die Konstruktion unendlicher Listen und Bäume, solange am Ende nur endliche Teilmengen abgerufen werden.

6.1 Funktionen

Funktionsaufrufe in Haskell benötigen keine Klammern, sondern werden einfach durch Leerzeichen in der Form `funktion arg1 arg2 arg3` notiert. Definitionen von Funktionen erfolgen gewöhnlich über Gleichungen:

```
addTwo :: Int -> Int
addTwo x = x + 2
```

Die erste Zeile deklariert den Typen von addTwo, welcher mit `Int -> Int` eine Funktion ist, die einen `Int` als Eingabe bekommt und ebenfalls einen `Int` zurückgibt. Typdeklarationen sind weitestgehend optional, da die Typinferenz diese fast immer selbstständig erschließen kann. Andererseits bieten Typdeklarationen bereits eine gewisse Dokumentation für den Programmierer und werden deshalb meist verwendet.

Typen von Funktionen mit mehreren Parametern erscheinen zunächst verwirrend:

```
add :: Int -> Int -> Int
add x y = x + y
```

Der Typoperator `->` ist rechtsassoziierend, weshalb `Int -> Int -> Int` äquivalent ist zu `Int -> (Int -> Int)`. Streng genommen besitzt eine Funktion in Haskell nämlich immer nur genau einen Parameter. Eine „Funktion mit zwei Parametern“ ist deshalb in Wirklichkeit eine Funktion (mit einem Parameter), die eine zweite Funktion (mit ebenfalls einem Parameter) zurückgibt. Dieses Konzept wird als Currying bezeichnet und ermöglicht zusammen mit Funktionen höherer Ordnung sehr elegante Formulierungen von Algorithmen. Beispielsweise bekommt die Funktion `map` mit Typsignatur `(a -> b) -> [a] -> [b]` eine Funktion und wendet diese Elementweise auf eine Liste

an - `a` und `b` stehen hier für beliebige Typen. Um nun zu jedem Element einer Liste die Zahl 5 zu addieren, kann nun der Code

```
map (add 5) [1, 2, 3, 4]  -- ergibt [6, 7, 8, 9]
```

verwendet werden – `add 5` erzeugt also eine Funktion, die 5 zu ihrem Parameter addiert. Auch Funktionen können direkt auf diese Art definiert werden:

```
addFiveToList :: [Int] -> [Int]
addFiveToList = map (add 5)
```

Anonyme Funktionen können mit `\` erzeugt werden, was an ein Lambda erinnern soll. Ohne Currying und `add`-Funktion könnte daher

```
addFiveToList = \ list -> map (\ x -> x + 5) list
```

als äquivalente Funktionsdefinition benutzt werden.

Infixoperatoren wie `+` können ebenfalls vom Programmierer selbst definiert werden. Auch Links- oder Rechtsassoziativität sowie Operatorpräzedenz können angepasst werden, wobei Infixoperatoren grundsätzlich eine niedrigere Präzedenz als Präfixfunktionen haben. Mittels Klammerung kann ein Infixoperator in eine Präfixfunktion umgewandelt werden, so dass z.B. `(+) 2 3` geschrieben werden kann. Auch Infixoperatoren unterstützen direkt Currying, daher erzeugt `(-2)` eine Funktion, die vom Argument 2 subtrahiert. Für die obige, sogenannte punktfreie[1] Art der Funktionsdefinition ist auch der Kompositionsoperator `.` nützlich:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(g . f) x = g (f x)

addFiveThenMultiplyByThree :: Int -> Int
addFiveThenMultiplyByThree = (*3) . (+5)
```

Lokale Funktionen können über das Schlüsselwort `where` definiert werden:

```
sumSquare x y = square (x + y)
  where
    square x = x * x
```

Funktionsaufrufe werden grundsätzlich linksassoziativ interpretiert, d.h. `foo a b c d` und `((((foo a) b) c) d)` sind äquivalent. Manchmal gibt es aber verschachtelte Funktionsaufrufe, für die auf diese Weise sehr viele Klammern benötigt würden, z.B.

```
a (b c (d (e f)))
```

In solchen Fällen kann der `$`-Operator benutzt werden, wobei `a $ b c d` dasselbe ist wie `a (b c d)`. Das Beispiel wird daher zu

```
a $ b c $ d $ e f
```

und damit etwas einfacher zu lesen.

[1] Punktfrei bedeutet, dass für Zwischenergebnisse keine benannten Variablen oder Parameter benötigt werden, sondern diese wie beim Currying implizit bleiben.

6.2 Datentypen

Es gibt in Haskell vier Arten von Typen: Zunächst gibt es primitive Typen wie `Char`, `Int` oder `Float` sowie zusammengesetzte Typen wie Listen (Schreibweise `[a, b, c]`) und Tupel (Schreibweise `(a, b)`). Eine Sonderrolle hat hier das leere Tupel `()` (genannt „Unit“), welches dem Typ `void` in C entspricht und in eine der beiden obigen Kategorien einzuordnen ist.

Daneben gibt es Typaliase - beispielsweise ist der Typ `String` lediglich ein Alias für Listen von Zeichen:

```
type String = [Char]
```

Die letzte Kategorie sind algebraische Datentypen, welche mit dem `data`-Schlüsselwort definiert werden:

```
data Shape = Point
           | Circle Float
           | Rectangle Float Float
```

Hier sind `Point`, `Circle` und `Rectangle` verschiedene Konstruktoren, mit denen Instanzen vom Typ Shape erzeugt werden können. Konstruktoren in Haskell können sowohl als Funktionen (wie `Rectangle :: Float -> Float -> Shape`) als auch zum Pattern Matching verwendet werden:

```
shapeArea :: Shape -> Float

-- Pattern Matching in Gleichungen
shapeArea Point           = 0
shapeArea (Circle r)      = pi * r * r
shapeArea (Rectangle x y) = x * y

-- Alternative mit case-Syntax
shapeArea s = case s of
                Point         -> 0
                Circle r      -> pi * r * r
                Rectangle x y -> x * y
```

Datentypen können auch durch Typvariablen parametrisiert sein, so beschreibt der eingebaute Typ `Maybe` Werte, die nicht immer vorhanden sind:

```
data Maybe a = Nothing | Just a

tryFindRecord :: Database -> ID -> Maybe Record
tryFindRecord db id = if hasRecord db id
                      then
                        Just (fetchRecord db id)
                      else
                        Nothing
```

`Nothing` entspricht also in etwa `null` in anderen Sprachen – mit dem Unterschied, dass in Haskell ganz klar definiert ist, wo dies vorkommen kann und wo nicht.

Für den Fall, dass nur ein Konstruktor mit nur einem Argument benutzt wird, kann statt `data` auch `newtype` verwendet werden, welches vom Compiler effizienter behandelt wird, da es sich lediglich um eine Verpackung für einen bestehenden Typen handelt. Dies wird vor allem zusammen mit Typklassen benutzt.

6.3 Typklassen

Oben wurden bereits Funktionen und Datentypen präsentiert, die auf beliebige Typen anwendbar sind. Oft ist es allerdings erforderlich, die Menge der möglichen Typen einzuschränken, wie auf Typen mit Plus-Operator oder solche, die in Strings umgewandelt werden können. Dies kann mit Hilfe von Typklassen erreicht werden, welche vergleichbar mit Interfaces in objektorientierten Sprachen sowie polymorphen Funktionen sind. Beispielsweise ist das Haskell-Äquivalent zu Javas `toString()` durch die Typklasse `Show` gegeben:

```
class Show a where
  show :: a -> String
```

Algebraische Datentypen (nicht aber Typaliase) können Typklassen mittels `instance` implementieren:

```
instance Show Shape where
  show Point           = "point"
  show (Circle r)      = "circle with radius " ++ show r
  show (Rectangle x y) = "rectangle
```

`++` ist der Haskell-Operator für Konkatenation von Listen. Arithmetische Operationen sind ebenfalls durch Typklassen definiert, so dass diese auch für andere Typen wie komplexe Zahlen überladen werden können.

Typvariablen können nun mittels `=>` auf bestimmte Typklassen eingeschränkt werden:

```
formatList :: (Show a) => [a] -> String
formatList xs = "{" ++ concat listWithSeparators ++ "}"
  where listWithSeparators :: [String]
        listWithSeparators = intersperse " - " stringList
        stringList :: [String]
        stringList = map show xs

showList [1, 2, 3] -- ergibt "{1 - 2 - 3}"
```

Derartige Typabhängigkeiten werden in dieser Arbeit in Beispielprogrammen der Einfachheit halber oft ausgelassen und stattdessen `(...) =>` geschrieben.

Darüber hinaus besitzt Haskell „Multi Type Classes“, die durch eine Kombination von Typen implementiert werden. Auch können die Typvariablen selbst parametrisiert sein. Letzteres ist unter anderem für einige kategorientheoretische Begriffe nützlich, die in Haskell durch Typklassen ausgedrückt werden.

6.4 Kategorien in Haskell

Die Haskell-Bibliotheken bieten viele kategorientheoretische Begriffe an, die allerdings alle bestimmten Einschränkungen unterliegen. Die Objekte einer Kategorie sind hier immer Haskell-Typen, daher ist die Typklasse `Category` wie folgt definiert:

```
class Category cat where
  id   :: cat a a
  (.)  :: cat b c -> cat a b -> cat a c
```

Ein Typ `cat`, der selbst zwei Typparameter benötigt, ist also eine Instanz von `Category`, wenn es generische Identitäts- und Kompositionsoperatoren gibt, die für beliebige Typen a, b, c benutzt werden können. Genau genommen bestimmt `Category` also keine Kategorien, sondern vielmehr die Morphismen bestimmter Kategorien. Auch die Axiome werden hier nicht gefordert, daher liegt es am Programmierer, dies für eine „vernünftige“ Programmsemantik selbst zu verifizieren. Man sieht hier schon, dass die Haskell-Begriffe nur sehr vage mit den mathematischen übereinstimmen, was auch bei den weiteren Definitionen so bleiben wird. Die einfachste Instanz von `Category` ist `(->)`, also die Kategorie der Haskell-Funktionen, bezeichnet als **Hask**. Oft wird statt `.` der `>>>`-Operator (genannt „vor“) benutzt mit `f >>> g = g . f`.

6.4.1 Arrows

Die Typklasse `Arrow` beschreibt (die Morphismen von) Kategorien, für die ein Funktor aus der Kategorie **Hask** existiert, so dass sich jeder Haskell-Funktion vom Typ `a -> b` ein Morphismus vom Typ `cat a b` zuordnen lässt. Dies ist deshalb sinnvoll, da viele Kategorien in Haskell „mehr“ können als normale Funktionen - formal eine zu **Hask** isomorphe Unterkategorie besitzen. So benutzt MagicL „Funktionen, die fehlschlagen können“ oder „Funktionen mit Nebeneffekten“ als Kategorien, die jeweils auch normale Funktionen enthalten. Zusätzlich wird eine Operation auf Tupeln gefordert, aus der sich ein Produkt zusammensetzen lässt, während ein Coprodukt zunächst nicht gefordert wird:

```
class (Category ar) => Arrow ar where
  arr   :: (a -> b) -> ar a b
  first :: ar a b   -> ar (a, c) (b, c)
```

`arr` ist der Funktor, der jede Funktion auf einen Arrow abbildet[2]. `first` ist eine Funktion, die aus einem Arrow einen Arrow auf Tupeln macht, der nur auf dem ersten Element arbeitet, das zweite dagegen unverändert durchschleift. Somit lassen sich zusätzliche Werte weiterreichen, außerdem lassen sich aus `first` sinnvolle Operationen ableiten:

```
  second :: ar a b -> ar (c, a) (c, b)
  second = arr swap >>> first f >>> arr swap
    where swap (x, y) = (y, x)
```

[2]Die Typen werden hierbei auf sich selbst abgebildet.

```
(***) :: ar a b -> ar a' b' -> ar (a, a') (b, b')
f *** g = first f >>> second g

(&&&) :: ar a b -> ar a b' -> ar a (b, b')
f &&& g = arr diag >>> (f *** g)
  where diag x = (x,x)
```

- `second` ist analog zu `first`, reicht allerdings das erste Element unverändert weiter.
- `f *** g` wendet `f` auf das erste Element, danach `g` auf das zweite Element eines Tupels an.
- `f &&& g` ist nun die Haskell-Entsprechung von $\langle f, g \rangle$ in Def. 3. `f` und `g` werden also beide auf die Eingabe angewendet und deren Ergebnisse zu einem Tupel zusammengesetzt.

Möchte man einen Arrow mit einer Funktion verknüpfen, gibt es mit

```
f >>^ func = f >>> arr func
```

noch etwas syntaktischen Zucker.

6.4.2 Coprodukte: Die Klasse `ArrowChoice`

Die Haskell-Entsprechung zu einer disjunkten Vereinigung ist der `Either`-Datentyp, der folgendermaßen definiert ist:

```
data Either a b = Left a | Right b
```

Coprodukte für Arrows werden durch die Klasse `ArrowChoice` beschrieben:

```
class (Arrow ar) => ArrowChoice ar where
  left :: ar a b -> ar (Either a c) (Either b c)
```

Auch hier wird mit `left` nur eine einfache Operation gefordert, aus der sich anschließend alles weitere konstruieren lässt. Diese Funktion wandelt einen Arrow von `a` nach `b` um in einen Arrow von `Either a c` nach `Either b c`. Bei einem `Left`-Wert wird also der ursprüngliche Arrow angewendet, ein `Right`-Wert wird dagegen unverändert weitergereicht.

```
right :: ar a b -> ar (Either c a) (Either c b)
right f = arr swap >>> left f >>> arr swap
  where swap (Left x)  = Right x
        swap (Right x) = Left x

(+++) :: ar a b -> ar a' b' -> ar (Either a a') (Either b b')
f +++ g = left f >>> right g

(|||) :: ar a c -> ar b c -> ar (Either a b) c
f ||| g = (f +++ g) >>> arr dropEither
```

```
  where dropEither (Left x)  = x
        dropEither (Right x) = x
```

Die Definitionen sind weitgehend analog zu den entsprechenden Produkt-Operationen:

- `right` bearbeitet nur `Right`-Werte, während `Left`-Werte unverändert bleiben.
- `f +++ g` wendet auf `Left`-Werte `f` an, auf die anderen `g`.
- Haben `f` und `g` denselben Rückgabetyp, kann `f ||| g` verwendet werden, welches das in diesem Fall unnötige `Either` verschwinden lässt. Dies entspricht $[f, g]$ in Def. 4

6.4.3 Funktoren

Die Haskell-Bibliotheken definieren eine Klasse `Functor`, welche allerdings nur Endofunktoren über der Kategorie **Hask** repräsentieren:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Ein Datenkonstruktor `f` ist also Instanz von `Functor`, wenn die Operation `fmap` Funktionen von `a` nach `b` auf Funktionen von `f a` nach `f b` abbildet. Der mathematische Funktor besteht hier also aus `(f,fmap)`.

Diese Arbeit benutzt stattdessen eine eigene `Functor`-Klasse, die verschiedene Kategorien zulässt:

```
class Functor f ar | f -> ar where
  lift :: ar a b -> f a b
```

Die Arrows `f` und `ar` bilden eine Instanz von `Functor`, wenn eine `lift`-Operation `ar`-Arrows auf `f`-Arrows zwischen den gleichen Typen abbildet, wobei der Typ `f` den Typ `ar` bestimmt. Der mathematische Funktor hier ist also `(id,lift)`. Im Vergleich zu Haskells Standard-`Functor` ist diese Version also in den Kategorien allgemeiner, aber dafür spezieller in der Abbildung der Objekte, da hier die Identität vorgeschrieben ist. Man könnte auch dies allgemein formulieren, aber im Rahmen von MagicL reicht die spezielle Version bisher aus.

Alle hier verwendeten `Functor`-Instanzen konstruieren aus einem `Arrow`-Typ einen zweiten mit zusätzlichen Eigenschaften, z.B. erweitert der `FailFunctor` einen Arrow-Typ um mögliches Scheitern. Die Instanz-Deklaration dafür sieht im Gerüst folgendermaßen aus (die Details werden in Abschnitt 8.1 erläutert):

```
newtype FailFunctor ar a b = ...

instance (Arrow ar) => Functor (FailFunctor ar) ar where
  lift f = ...
```

6.4.4 Monaden

Die Typklasse `Monad` aus den Haskell-Bibliotheken beschreibt Monaden über der Kategorie **Hask**. η aus Abschnitt 5.6 heißt hier `return`, statt μ wird der Operator `>>=` mit anderer Signator gefordert:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

Es gibt auch eine genaue Entsprechung von μ, die hier `join` heißt:

```
join :: (Monad m) => m (m a) -> m a
join x = x >>= id
```

Zu jeder Haskell-Monade `m` lässt sich wieder die Kleisli-Kategorie bilden, die die Datentypen als Objekte sowie die Funktionen `a -> m b` als Morphismen enthält. Die Haskell-Bibliotheken stellen hierfür den Datentyp `Kleisli` bereit:

```
newtype Kleisli m a b = Kleisli (a -> m b)

instance (Monad m) => Category (Kleisli m)
  where id = Kleisli return
        Kleisli f . Kleisli g = Kleisli composed
          where composed x = g x >>= f

instance (Monad m) => Arrow (Kleisli m)
  where arr fun = Kleisli (return . fun)
        first (Kleisli f) = Kleisli tupleF
          where tupleF (x, z) = f x >>= (\ y -> return (y, z))
```

Arrows und Monaden werden beide verwendet, um abstrakt Verknüpfungen von speziellen Operationen zu beschreiben. Arrows sind allgemeiner, denn jede Monade lässt sich mittels `Kleisli` auf einen entsprechenden Arrow abbilden. Auf der anderen Seite entspricht aber nicht jedem Arrow eine Monade, denn aus den Arrow-Operationen allein lässt sich `>>=` nicht konstruieren. Hierfür wird zusätzlich die Operation `app` benötigt, die von der Typklasse `ArrowApply` definiert wird (für Details siehe [25, S. 18f]):

```
class (Arrow ar) => ArrowApply ar where
  app :: ar (ar a b, a) b
```

Mit `app` wird also ein Arrow bereitgestellt, der als Parameter einen weiteren Arrow sowie eine Eingabe bekommt, um dann den übergebenen Arrow auf die Eingabe anzuwenden. Damit ist es möglich Arrows einzusetzen, die erst im Verarbeitungsprozess und in Abhängigkeit der Eingabedaten erzeugt werden – eine Eigenschaft, die Monaden generell besitzen und die unter anderem für die Konstruktion kontextsensitiven Parser benötigt wird (siehe Kapitel 8).

Instanzen der Klasse `ArrowChoice` sind vollständig äquivalent zu Monaden. Dies wirft die Frage auf, weshalb Haskell überhaupt Monaden benutzt und nicht alles über Arrows realisiert. Zum einen gibt es dafür historische Gründe: Monaden waren bereits 1993 in

Haskell präsent [24, S.23ff], während Arrows erst 1998 von John Hughes vorgeschlagen wurden, da sich spezielle Parser mit statischen Komponenten nicht durch Monaden ausdrücken lassen [25]. Zum anderen bieten Monaden aber auch Vorzüge: Die Definition einer Monade ist etwas kürzer, da keine `first`-Funktion für Produkte bereitgestellt werden muss. Wie obiger Code zeigt, lässt sich dies bereits mittels `>>=` ausdrücken. Zudem gibt es für Monaden in Haskell die praktische `do`-Notation:

```
do x <- foo
   y <- bar x
   return (x, y)
```

ist syntaktischer Zucker für

```
foo >>= (\ x ->
  bar x >>= (\ y ->
    return y))
```

und ermöglicht eine Schreibweise, die imperativen Programmen ähnelt. Dies ist kein Zufall, denn Nebeneffekte werden in Haskell ebenfalls mit einer Monade beschrieben (siehe Abschnitt 6.4.4), so dass auch dort die `do`-Notation benutzt werden kann. Diese Verwendung erklärt auch nachträglich den Namen `return`, wobei es sich noch immer um die η-Transformation handelt und nicht etwa ein syntaktisches `return`-Statement wie in imperativen Sprachen üblich.

Für Arrows gibt es mit `proc` auch eine Notation, mit der Zwischenergebnisse benannt werden können. Diese ist aber weniger simpel und elegant als das Monadenäquivalent. Andererseits eignen sich Arrows besser als Monaden für eine punktfreie Schreibweise, d.h. einer Schreibweise ohne Zwischenvariablen, bei der ausschließlich auf Komposition zurückgegriffen wird. Punktfreie Notation wird auch bei Definitionen von Funktionen oft verwendet und ist für viele Haskell-Programmierer natürlicher und eleganter.

Beispiele für Monaden

`Maybe` Funktionen, die fehlschlagen können, lassen sich in der Form `a -> Maybe b` darstellen. Möchte man mehrerer solcher Funktionen verketten, werden normalerweise viele Fallunterscheidungen benötigt, da bei jeder Funktion die Rückgabe geprüft und nur im Erfolgsfall die nächste aufgerufen werden muss. Haskell vereinfacht dies, indem `Maybe` zur Monade erklärt wird:

```
instance Monad Maybe
  where return = Just
        (Nothing >>= _) = Nothing
        (Just x  >>= f) = f x
```

Obiges Code-Beispiel, mit einer Typanschrift auf die Maybe-Monade festgelegt

```
(do x <- foo
    y <- bar x
    return (x, y)) :: Maybe (Integer, Integer)
```

wird wie folgt interpretiert: Wenn `foo` erfolgreich ist, wird das Ergebnis (lokal) in `x` gespeichert. Ist daraufhin `bar x` ebenfalls erfolgreich, wird dieses in `y` gespeichert und schließlich der Wert `Just (x, y)` zurückgegeben. Schlägt eine der Funktionen fehl, ist das Ergebnis `Nothing`.

`IO` Jede Programmiersprache benötigt für die reale Welt Möglichkeiten, Operationen mit Nebeneffekten wie das Schreiben von Dateien oder solche, die von externen Nebeneffekten abhängen, wie das Lesen von Dateien, auszuführen. Dies lässt sich zunächst schwer in eine pure Sprache wie Haskell integrieren. Die `IO`-Monade bietet hier eine Lösung: Man stelle sich einen fiktiven Datentyp `World` vor, der den gesamten Zustand der externen Welt enthält. `IO a` ist nun eine Funktion, die die Welt liest und eine andere Welt sowie ein a zurückgibt:

```
newtype IO a = World -> (World, a)
```

Zum Verketten zweier `IO`-Operationen wird die von der ersten zurückgegebene Welt an die zweite übergeben. Damit dies nicht von Hand geschehen muss, wird `IO` wieder als Monade deklariert. Nun gibt es aber natürlich keinen echten `World`-Typen. In Wirklichkeit wird also der Haskell-Compiler alles was mit `IO` verpackt ist in imperativen Code übersetzen. Dennoch gelingt dadurch die saubere Trennung von purem Code und solchem, der Nebeneffekte enthalten kann. Da es nicht möglich ist, eine „Auspack-Funktion“ `IO a -> a` zu definieren[3], das Gegenteil aber mit `return` möglich ist, kann imperativer Code immer funktionalen, funktionaler Code aber niemals imperativen enthalten. Die äußerste Funktion jedes Haskell-Programms, `main`, wird deshalb immer in der `IO`-Monade ausgeführt.

Dank der Kleisli-Kategorie lassen sich mittels `IO` auch Arrows bereitstellen, die Nebeneffekte enthalten:

```
type IOArrow = Kleisli IO
```

6.5 Zusammenfassung

Haskell ist eine pur funktionale, statisch typisierte Programmiersprache mit Typinferenz. Konzepte wie Currying und Funktionen höherer Ordnung ermöglichen einen eleganten, modularen und punktfreien Programmierstil. Für die Definition eigener Datenstrukturen stehen algebraische Datentypen bereit, die in etwa die Rolle einer Klasse in objektorientierten Sprachen einnehmen. Es können auch Typvariablen verwendet werden, die deutlich mächtiger und präziser sind als beispielsweise Java-Generics [34]. Die Rolle von Interfaces nehmen in Haskell Typklassen ein.

In Haskell werden mit Arrows und Monaden auch einige kategorientheoretische Konzepte umgesetzt und in Form von Typklassen bereitgestellt. Während Arrows allgemeiner, konzeptionell einfacher sowie optimal für eine punktfreie Formulierung von Algorithmen sind, können Monaden teilweise einfacher implementiert werden und ermöglichen

[3] Es gibt eine solche Funktion in Haskell, die aber normalerweise nicht verwendet werden sollte.

mittels do-Notation eine punktierte Schreibweise mit benannten Zwischenvariablen. Alle Monaden lassen sich über die Kleisli-Kategorie in Arrows umwandeln und viele Arrows in Monaden.

Haskell selbst nutzt Monaden, um Funktionen mit Seiteneffekten sauber in die ansonsten pure Sprache integrieren zu können. Die Monade `IO` lässt sich als verstecktes Weiterreichen eines `World`-Zustandes von Funktion zu Funktion veranschaulichen.

MagicL legt den Schwerpunkt auf Arrows und beschreibt so allgemein Compiler. Parser werden hier als Spezialform eines Compilers gesehen.

7 Architektur von MagicL

Dieses Kapitel gibt einen grundlegenden Überblick über die Architektur und Verwendung von MagicL. Zunächst wird der prinzipielle Aufbau von Compilern diskutiert und eine Übersicht über Hilfsmittel zur Compilerdefinition in Abhängigkeit von Ein- und Ausgabemodellen gegeben. Anschließend wird auf die von MagicL angebotene DSL für Compilerdefinitionen eingegangen, die mit der Haskell-DSL eine funktionale Programmiersprache enthält. Zuletzt wird eine Übersicht über die Programmkomponenten gegeben, aus denen MagicL selbst besteht.

7.1 Aufbau von Compilern

Compiler sind in MagicL allgemein Übersetzungsprozesse (und damit Morphismen). Es kann sich dabei um elementare Teilschritte handeln wie ein Makro, das lediglich eine Aufzählungsliste verarbeitet, als auch um große, komplexe Übersetzer wie den kompletten Compiler von Slide-DSL-Quelltext nach Latex. Compiler können baukastenartig mit Kombinatoren wie dem Kompositionsoperator zusammengesetzt werden.

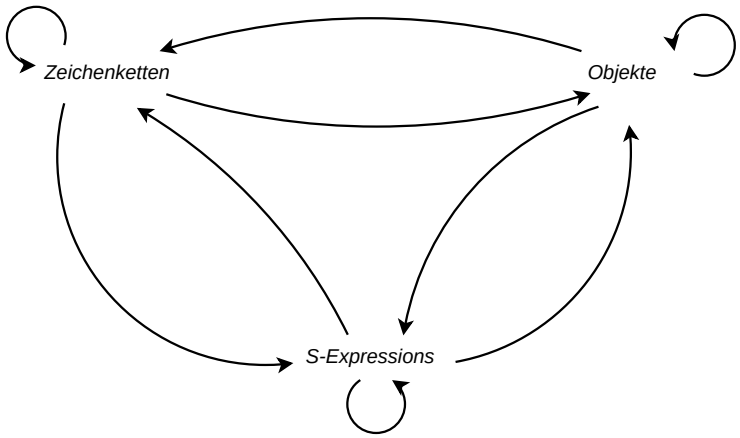


Abbildung 7.1: Mögliche Ein- und Ausgabemodelle von Compilern

Ein- und Ausgaben von Compilern können jeweils die drei Modelltypen Zeichenketten, S-Expressions und Objekte sein, so dass sich insgesamt neun mögliche Compilertypen ergeben, welche in Abb. 7.1 als Pfeile eingezeichnet sind. MagcicL bietet in Abhängigkeit der Modelltypen verschiedene Hilfsmittel für die Erstellung von Compilern an:

Zeichenketten

Für das Einlesen von Zeichenketten steht eine Parser-Bibliothek zur Verfügung. Parser sind wie Compiler spezielle Morphismen und lassen sich ebenfalls durch Komposition und andere Kombinatoren aus kleinen, atomaren Bausteinen zusammensetzen. Die Erzeugung von Quelltext geschieht mit Hilfe einer funktionalen Generator-Bibliothek.

S-Expressions

S-Expressions werden in MagicL ebenfalls durch Parser verarbeitet, da diese genau wie Strings als Listen von Token gesehen werden können – auch wenn die Token hier nicht immer atomar sind. Eine S-Expression-Parser-Bibliothek erweitert daher die allgemeine Parser-Bibliothek um S-Expression-spezifische Funktionalität. Die Anforderung „verallgemeinerter Makros“ erfüllt MagicL genau durch das Konzept der S-Expression-Parser. S-Expression-Parser, die selbst S-Expressions generieren, werden als Lisp-Makros[1] bezeichnet und werden durch einige Hilfsfunktionen und eine spezielle Syntax besonders unterstützt. Das Erzeugen von S-Expressions erfolgt durch einen Quasiquote-Operator.

Objekte

Die Verarbeitung von Objekten erfolgt durch gewöhnliche Haskell-Funktionen, so dass MagicL hier keine eigene Funktionalität bereitstellen muss.

Abb. 7.2 zeigt, welche Hilfsmittel für die Erstellung der neun verschiedenen Compiler-Typen verwendet werden können.

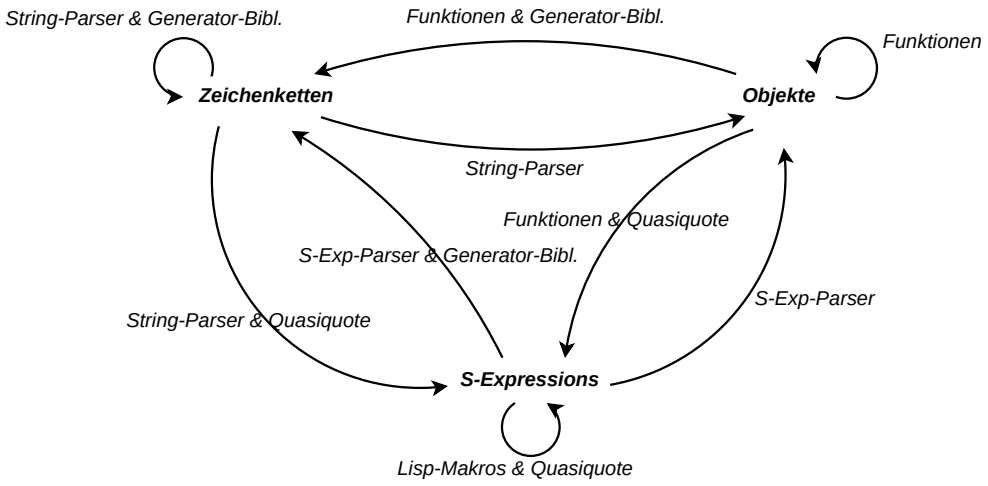


Abbildung 7.2: Hilfsmittel für Compilerdefinitionen nach Ein- und Ausgabemodellen

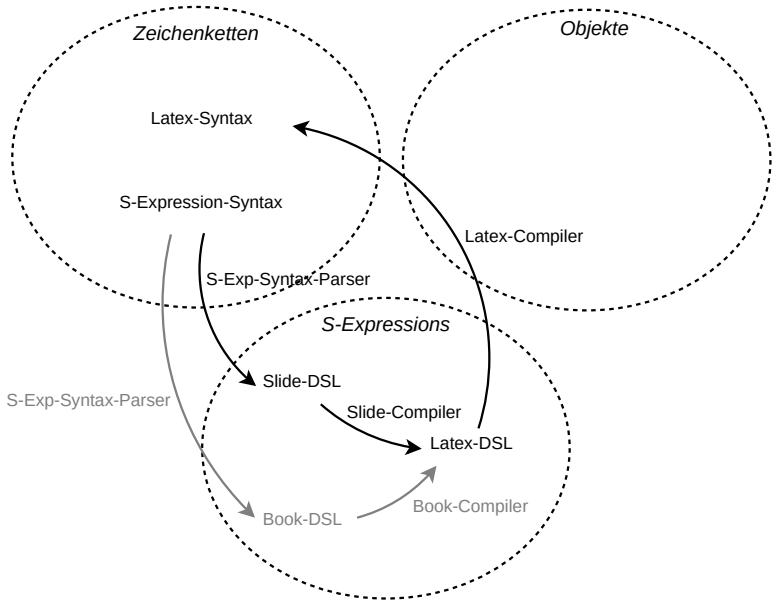


Abbildung 7.3: Mögliche Übersetzung der Slide-DSL

Beispiel: Übersetzung der Slide-DSL

Ein Compiler von der Slide-DSL nach Latex wird in Hinblick auf Wiederverwendbarkeit sinnvollerweise aus zwei Komponenten zusammengesetzt: Eine eigene Latex-DSL – praktisch eine „geklammerte Version“ von Latex – muss nach Latex-Quelltext übersetzt werden. Dieser Compiler wird mittels S-Expression-Parser und Generator-Bibliothek implementiert. Der eigentliche Slide-Compiler übersetzt die Slide-DSL in die Latex-DSL und wird mit Lisp-Makros und Quasiquote-Notation realisiert, wodurch ein relativ kurzes und lesbares Programm zu erwarten ist. Aufgrund der Modularisierung können weitere Latex-basierte DSLs – beispielsweise eine Book-DSL für das Schreiben von Büchern – mit sehr geringem Aufwand eingefügt werden (siehe Abb. 7.3). Hierfür ist lediglich ein weiterer, mit Lisp-Makros und Quasiquote realisierter Compiler von Book-DSL nach Latex-DSL nötig. Da alle DSLs auf S-Expressions basieren, wird kein eigener String-Parser benötigt. Stattdessen wird der Standardparser für S-Expression-Syntax wiederverwendet. Weiterhin ist es auch möglich, die Slide-DSL mit einer anderen Backend-Sprache wie HTML zu verbinden, so dass derselbe Folien-Quelltext für mehrere Zwecke benutzt werden kann. Typisierte Objekte wurden in diesem Beispiel nicht verwendet. Es wäre aber auch möglich, ein typisiertes Latex-Modell zu definieren, wodurch einerseits die Suche nach möglichen Fehlern vereinfacht wird, andererseits aber auch der Entwicklungsaufwand für den Latex-Compiler steigt. Dieser Ansatz wird in der Implementierung einer Haskell-DSL verwendet.

[1] Es handelt sich hierbei nicht exakt um eine Nachbildung von Makros in Lisp – sie sind aber bereits deutlich näher an Lisp-Makros als allgemeine S-Expression-Parser.

7.2 Bereitgestellte DSLs

MagicL-Nutzer definieren neue Compiler ausschließlich in einer bereitgestellten Compiler-DSL. Diese enthält den Quasiquote-Operator und Konstrukte zur einfachen Makro-Definition und integriert mit der Haskell-DSL eine komplette funktionale Programmiersprache, die eine S-Expression-Version von Haskell darstellt. Dieser Abschnitt stellt zunächst die Haskell-DSL und anschließend die Compiler-DSL genauer vor.

7.2.1 Haskell-DSL

Da die Compiler-DSL von MagicL letztlich nach Haskell übersetzt werden soll, ist es wie im Latex-Beispiel sinnvoll, eine eigene Haskell-DSL als Zwischenstufe anzubieten, so dass Nutzer weitere DSLs direkt mit Lisp-Makros und Quasiquote nach Haskell übersetzen können. Die Haskell-DSL ist im Wesentlichen eine auf S-Expression-Syntax angepasste Version von Haskell. Beispielsweise würde die Haskell-Funktionsdefinition

```
sumOfSquares x y = (x2 + y2)
  where
    x2 = (x * x)
    y2 = (y * y)
```

in der Haskell-DSL als

```
(= (sumOfSquares x y)
   (+ x2 y2)
  (where
    (= x2 (* x x))
    (= y2 (* y y))))
```

geschrieben werden. Intern wird die Haskell-DSL zunächst in ein getyptes Modell und anschließend nach Haskell übersetzt, wie Abb. 7.4 zeigt. In Kapitel 10 wird genauer auf Spezifikation und Umsetzung der Haskell-DSL eingegangen und die Vor- und Nachteile des Zwischenschritts über ein getyptes Modell diskutiert.

7.2.2 Compiler-DSL

Obwohl es im Prinzip möglich ist, eigene Compiler direkt in Haskell zu implementieren, ist für Anwendungscode die Compiler-DSL angedacht. Diese beinhaltet zunächst die Haskell-DSL als Subsprache, so dass dem Metaprogrammierer die volle Haskell-Funktionalität zur Verfügung steht. Darüber hinaus enthält die Compiler-DSL einen Quasiquote-Operator und Operatoren für die gegenüber Haskell leicht vereinfachte Definition von Makros und zusammengesetzten Compilern.

Die Syntax von Quasiquote und Unquote unterscheidet sich etwas von Lisp: Statt `‘(1 2 ,x 3)` wird beispielsweise `(’ (1 2 (, x) 3))` verwendet. Auf diese Art kann die normale S-Expression-Syntax verwendet werden, während Lisp-Parser dies als Sonderfall berücksichtigen müssen. Zudem passt diese Syntax gut zur Anforderung einer beliebigen

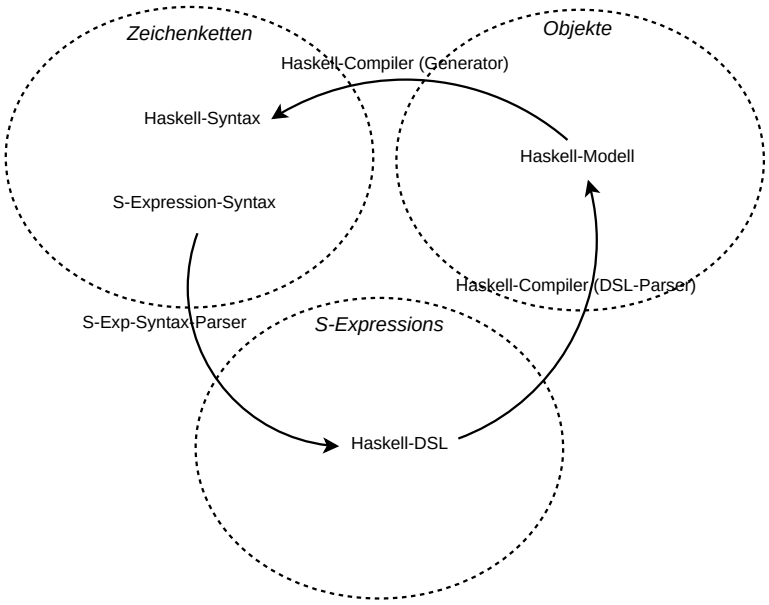


Abbildung 7.4: Übersetzung der Haskell-DSL

Anzahl von zurückgegebenen Ausdrücken – beispielsweise können mit `(')` kein Ausdruck oder mit `(' a b c d)` vier Ausdrücke erzeugt werden. Die Compiler-DSL von von einem Compiler-Compiler in die Haskell-DSL übersetzt und anschließend vom Haskell-Backend weiterverarbeitet, wie in Abb. 7.5 dargestellt. Definition und Implementation der Compiler-DSL finden sich in Kapitel 11.

7.3 Komponenten des Frameworks

Intern besteht MagicL aus einer Reihe von Komponenten, die alle in Haskell implementiert sind. Grundlegend ist eine Arrow-Bibliothek, auf der unter anderem die Parser-Bibliothek aufsetzt. Für das Erzeugen von Quelltext steht eine Generator-Bibliothek bereit. Darauf aufbauend stellt eine S-Expression-Bibliothek spezielle Parser und Hilfsfunktionen für den Umgang mit S-Expressions zur Verfügung. Zudem enthält MagicL ein minimales Test-Framework für Compiler sowie ein eigenes Build-Tool.

7.3.1 Arrow-Bibliothek

Arrows sind die grundlegende Abstraktion, auf der Compiler und Parser in MagicL basieren. Eine Arrow-Bibliothek definiert daher abstrakt Arrow-Kombinatoren wie z.B. `many`, welches eine Wiederholung entsprechend dem Operator `*` in regulären Ausdrücken enthält, welche in Abschnitt 8.4 gemeinsam mit der Parser-Bibliothek detaillierter vorgestellt werden. Darüber hinaus wird mit den Typklassen `Executable` und `Compilable` sehr allgemeine Compiler-Schnittstellen definiert, die es beispielsweise ermöglichen, Standardcompiler fest mit bestimmten Ein- und Ausgabetypen zu verdrahten, so dass diese nicht

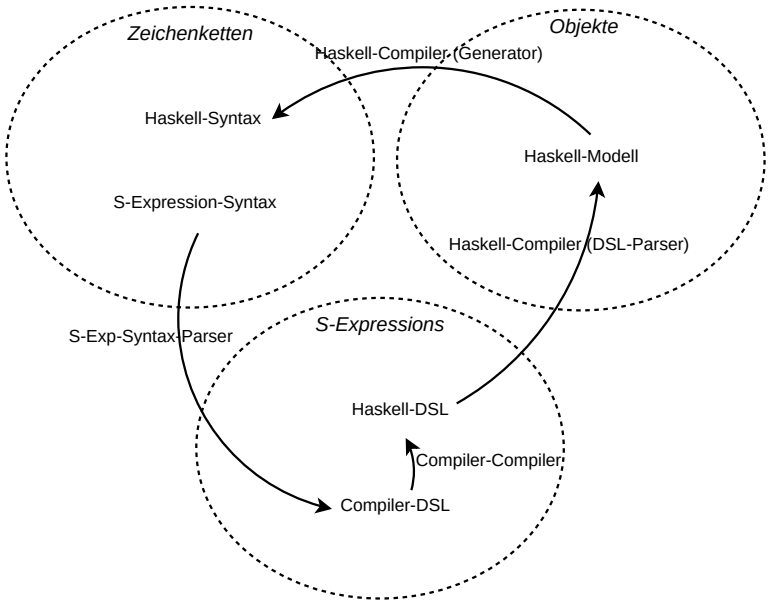


Abbildung 7.5: Übersetzung der Compiler-DSL

explizit aufgerufen sondern automatisch durch Haskells Typinferenz gefunden werden können. Diese Schnittstelle wird in Abschnitt 9.3 erläutert.

7.3.2 Parser-Bibliothek

Parser werden in Kapitel 8 als Arrows einer speziellen Kategorie definiert, die das Weiterreichen eines Eingabestreams sowie die Möglichkeit eines Scheiterns beinhaltet. Diese Kategorie realisiert der Typ `ParseFunctor` (siehe Abschnitt 8.3). Zudem bietet die Parser-Bibliothek einige Funktionen, mit denen elementare Parser definiert werden können. Beispielsweise erzeugt `member "aeiou"` einen Parser, der genau einen Vokal einliest.

7.3.3 Generator-Bibliothek

Die Generatorbibliothek definiert eine Reihe von Funktionen, um die Erzeugung von formatiertem Quelltext zu vereinfachen. Hierfür wird zunächst durch verschachtelte Aufrufe von Konstruktor-Funktionen wie `lines`, welche eine Liste von Texten mit Zeilenumbrüchen verbindet, ein Qellcode-Modell aufgebaut, das anschließend mit `layout` für eine bestimmte Zeilenlänge[2] formatiert wird. So lässt sich mittels

```
layout 80 (braces
           (indent2
            (lines [text "foo",
                    words [text "hello", text "world"],
                    commaSep [text "1", text "2", text "3"]])))
```

[2]Die Zeilenlänge ist für Gruppierungen relevant, welche in diesem Beispiel nicht verwendet werden.

der Quelltext

```
{foo
  hello world
  1, 2, 3}
```

erzeugen. Dies erscheint auf den ersten Blick deutlich aufwändiger als eine direkte Erzeugung mit einer Templating-Bibliothek wie StringTemplate. Der funktionale Ansatz bietet allerdings mehrere Vorteile: Zum einen kann die Quelltext-Erzeugung auf diese Weise auf viele Komponenten aufgeteilt und so unter anderem direkt in (Sub-)Parsern benutzt werden. Zum anderen ist es möglich, eigene sprachspezifische Layout-Funktionen wie für `for`-Schleifen in Java zu definieren. Zudem passieren Syntaxfehler im generierten Code auf diese Weise nicht so schnell, schließlich können mit Funktionen wie `braces` niemals Fehler wie unbalancierte Klammern enstehen. Die Generator-Bibliothek wird in Abschnitt 9.1 besprochen.

7.3.4 S-Expression-Bibliothek

Eine Bibliothek enthält eine Reihe von Hilfsmitteln für die Verarbeitung von S-Expressions. Hierzu gehören ein String-Parser sowie ein Pretty-Printer für S-Expression-Syntax. Darüber hinaus werden einige spezielle Kombinatoren für das Verarbeiten von S-Expressions durch Parser bereitgestellt. Beispielsweise kann mit `macro` ein S-Expression-Parser definiert werden, der wie ein Lisp-Makro nur aktiv wird, wenn das erste Element eines Ausdrucks mit dem Knotennamen übereinstimmt. Die S-Expression-Bibliothek wird in Abschnitt 9.2 erläutert.

7.3.5 Test-Framework

MagicL enthält ein primitives Unit-Test-Framework für die Überprüfung einzelner Compiler (und damit auch Parser und Makros). Zunächst werden dafür die Eingaben und erwarteten Ausgaben als Strings in Listen notiert – zum Beispiel ist die Übersetzung einiger Haskell-DSL-Ausdrücke nach Haskell wie folgt spezifiziert:

```
exprCases = [ ( "(Str 42)"                    -- Haskell-DSL
              , "\"42\"" )                    -- Haskell

            , ( "(Str a  b c)"                -- Haskell-DSL
              , "\"a b c\"" )                 -- Haskell

            , ( "(List a b (Str c))"          -- Haskell-DSL
              , "[a, b, \"c\"]" )             -- Haskell

            , ( "(foo a (bar b c) d)"         -- Haskell-DSL
              , "(foo a (bar b c) d)" )       -- Haskell

            , ( "(map + myList)"              -- Haskell-DSL
              , "(map (+) myList)" ) ]        -- Haskell
```

Ein Makro, was diese Ausdrücke übersetzt, kann dann mit dem Aufruf

```
testMacro "Expressions" exprMacro exprCases
```

getestet werden. Der erste Parameter dient der Dokumentation und wird für die Formatierung etwaiger Fehlermeldungen benutzt. Da es sich hierbei um keine essentielle Komponente handelt, wird auf das Test-Framework in dieser Arbeit nicht weiter eingegangen.

7.3.6 Build-Tool

Das Kompilieren des MagicL-Projektes ist relativ kompliziert, da für die Übersetzung des Compiler-Compilers bereits Code-Generierung benötigt wird. Aus diesem Grund wurde ein kleines Build-Tool in Haskell geschrieben, was Standardfunktionen wie bedingten Aufruf eines Compilers, falls die Quelldatei neuer ist als die Zieldatei, beinhaltet und damit die Übersetzung und das Testen von MagicL per Kommandozeile steuert. Das Build-Tool wird ebenfalls in dieser Arbeit nicht weiter behandelt.

7.4 Zusammenfassung

Benutzerdefinierte Compiler werden in einer bereitgestellten DSL geschrieben, die unter anderem eine S-Expression-Version von Haskell beinhaltet. Damit können Compiler zwischen beliebigen Modelltypen konstruiert werden, wobei je nach Modelltyp verschiedene Hilfsmittel verwendet werden. Während typisierte Objekte mit gewöhnlichen Haskell-Funktionen verarbeitet werden, stehen für Zeichenketten Parser- und Generator-Bibliotheken zur Verfügung. Parser, welche als Arrows realisiert sind, werden auch für die Verarbeitung von S-Expressions verwendet und sind MagicL's Umsetzung der Anforderung verallgemeinerter Makros. Für das Generieren von S-Expressions steht ein gegenüber Lisp syntaktisch leicht veränderter Quasiquote-Operator bereit, mit dem es unter anderem möglich ist eine beliebige Anzahl von Ausdrücken zu erzeugen.

8 Arrow-basierte Parser

Parser sind für die Umwandlung von Zeichenketten in Objekte verantwortlich und werden deshalb für Code-Generatoren benötigt, die ihre Eingaben aus Text-Dateien beziehen. In vielen Programmiersprachen ist die direkte Implementation von Parsern relativ aufwändig, weshalb der entsprechende Code häufig selbst aus einer DSL wie JavaCC generiert wird, die eine EBNF-ähnliche Notation mit bestimmten Abschnitten kombiniert, die Code in der Zielsprache enthalten.

Die Haskell-Bibliothek Parsec [30] hingegen beschreitet einen anderen Weg: Hier werden Parser als monadische Funktionen konstruiert, so dass diese direkt in Haskell und ohne spezielle Syntax beschrieben werden können. Parser-Kombinatoren wie `<|>` oder `many` entsprechen den EBNF-Operationen. Darüber hinaus können sogar eigene Kombinatoren definiert werden. So könnte `sepBy` Listen mit Trennzeichen einlesen, wenn ein Parser für die Elemente und einer für die Trennzeichen übergeben werden. Aus diesem Grund können komplexe Parser in Parsec oft deutlich eleganter konstruiert werden als mit anderen Parser-Generatoren – außerdem können hier auch kontextsensitive Sprachen wie XML ausnahmslos verarbeitet werden (siehe [30, S. 3]), was mit anderen Bibliotheken nicht möglich ist.

Es wäre möglich gewesen, Parser in MagicL mittels Parsec zu konstruieren. Dennoch wurde für diese Arbeit entschieden, ein eigenes, allgemeineres Framework für die Parser-Konstruktion zu entwickeln. Dadurch wird neben der freien Auswahl des Token-Typs[1] auch eine freie Auswahl der Kategorie ermöglicht, in der Parser ausgeführt werden. Zum Beispiel kann `IOArrow` benutzt werden, um Debug-Ausgaben oder andere Nebeneffekte in Parser einzubauen. Dies ist in Parsec nicht möglich.

MagicL benutzt Arrows und Funktoren, um möglichst allgemein zu sein. Alles, was im Rahmen dieser Arbeit implementiert wurde, hätte zwar auch wie in Parsec über Monaden und Monaden-Transformatoren konstruiert werden können. Dennoch besteht die Möglichkeit, dass später Ergänzungen vorgenommen werden sollen, die sich nicht mittels Monaden ausdrücken lassen – beispielsweise ein effizienterer Parsing-Algorithmus, der sowohl statische als auch dynamische Komponenten enthält, wie ihn Swierstra und Duponcheel entworfen haben (siehe [25, S. 8ff]). Zudem sind Arrows als programmiersprachliche Umsetzung von Morphismen anschaulicher als Monaden und besitzen eine praktische graphische Repräsentation.

Wenn allerdings kontextsensitive Parser wie in Parsec konstruiert werden sollen, müssen die zugrundeliegenden Arrows äquivalent zu Monaden sein und daher die Typklasse `ArrowApply` implementieren.

[1] Auch Parsec lässt es allerdings zu, dass statt Zeichenketten auch Listen beliebiger Token eingelesen werden können.

Dieses Kapitel entwickelt nun schrittweise einen Parser-Datentyp durch die Kombination einfacher Funktoren. Parser zeichnen sich hauptsächlich durch zwei Eigenschaften aus:

- Sie können fehlschlagen sowie Alternativmöglichkeiten im Falle des Scheiterns besitzen.
- Sie bearbeiten einen Zustand, der die Position im Eingabe-Stream beschreibt.

Diese beiden Eigenschaften werden von den Datentypen `FailFunctor` und `StateFunctor` umgesetzt.

8.1 Fehlschlagende Arrows

Berechnungen von A nach B, die fehlschlagen können, sollen zwei mögliche Resultate haben. Im Erfolgsfall wird ein normaler Rückgabewert aus B geliefert, während im Falle eines Scheiterns eine Fehlermeldung als String zurückgegeben wird – im Gegensatz zur `Maybe`-Monade, die im Fehlerfall nur ein `Nothing` liefert. Der Rückgabetyp ist deshalb das Coprodukt String $+ B$. Statt Morphismen von A nach B werden nun Morphismen von A nach String $+ B$ benutzt. Um den aufrufenden Code nicht komplizierter zu machen, empfiehlt es sich, diese Änderung in einer neuen Kategorie $\mathbf{C}_f$ zu verstecken. $f_f : A \to B$ aus der neuen Kategorie wird abgebildet auf $f : A \to \text{String} + B$. Der Arrow $\text{fail}_f : \text{String} \to a$ in C_f schlägt immer fehl und entspricht $\text{fail} : \text{String} \to \text{String} + a$ in C.

Der Operator $\bigvee : Mor_{A,B} \times Mor_{A,B} \to Mor_{A,B}$ bietet Alternativen, wodurch ein Backtracking im Fehlerfall möglich wird. Um Morphismen aus $\mathbf{C}$ in $\mathbf{C}_f$ benutzen zu können, gibt es den Funktor $\text{lift}_f : \mathbf{C} \to \mathbf{C}_f$, welcher die unveränderte, d.h. gelingende Operation für die neue Kategorie übernimmt.

Die Haskell-Entsprechung von String$+B$ ist `Either String b`, was sich mit `Failable b` abkürzen lässt, wenn man den parametrisierten Typ `Failable` einführt:

```
type Failable a = Either String a
```

Der Fehlerfall wird durch `Left String` ausgedrückt, ein Erfolg durch `Right a`. Fehlschlagende Arrows nun sind in MagicL durch den Typ `FailFunctor` implementiert:

```
newtype FailFunctor ar a b = FailF (ar a (Failable b))

instance (Arrow ar) => Functor (FailFunctor ar) ar where
    lift f = FailF (f >>> arr Right)
```

Arrows von `a` nach `b`, die fehlschlagen können, sind also Arrows von `a` nach `Failable b`, die in den zusätzlichen Konstruktor `FailF` eingebettet wurden. Um einen normalen Arrow `f` aus `ar` nach `FailFunctor ar` zu „liften“, wird der Rückgabewert in ein `Right` gebettet und der resultierende Arrow in `FailF`. Der Arrow `fail` wird in eine `Arrow` erweiternde Typklasse ausgegliedert, so dass dieser auch in anderen Kategorien bereitgestellt werden könnte:

```
class (Arrow ar) => ArrowFail ar where
  fail :: ar String a

instance (ArrowChoice ar) => ArrowFail (FailFunctor ar) where
  fail = FailF (arr Left)
```

Dem Oder-Operator $\bigvee$ entspricht in Haskell `<+>`, welcher von der Typklasse `ArrowPlus` aus den Haskell-Bibliotheken bereitgestellt wird und für `FailFunctor` wie folgt implementiert ist:

```
instance (ArrowChoice ar) => ArrowPlus (FailFunctor ar) where
  FailF f <+> FailF g = FailF ((f &&& id) >>> arr proc >>> (g ||| arr Right))
   where proc :: (Either a b, c) -> Either c b
         proc (Left  _, y) = Left y
         proc (Right x, _) = Right x
```

Es werden also zunächst `f` und die Identität parallel evaluiert. Dies ist nötig, um die Eingabe „durchzuschleifen“. Anschließend erfolgt ein Verarbeitungsschritt um das Tupel aufzulösen: Scheitert `f`, wird dessen Ausgabe verworfen, im Erfolgsfall die Eingabe. Anschließend wird im Fehlerfall `g` auf die Eingabe angewendet, bei Erfolg wird das Ergebnis wieder in ein `Right` eingepackt.

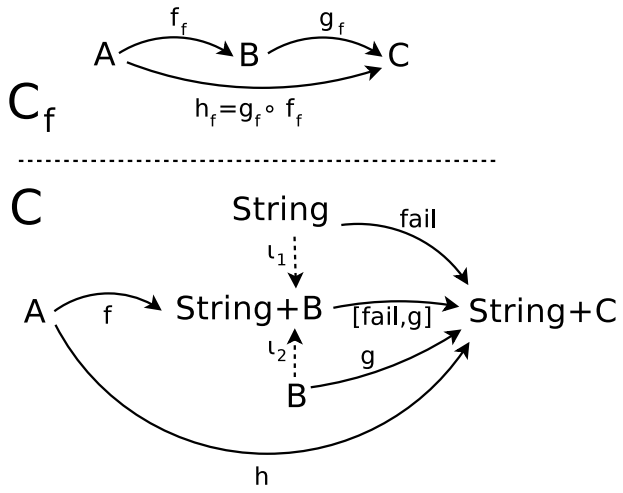


Abbildung 8.1: Komposition beim Fail-Funktor

Die Typabhängigkeit von `ArrowChoice` ist nötig, weil `FailFunctor ar` nur dann ein `Arrow` ist, wenn `ar` ein Coprodukt anbietet. Denn die Komposition in C_f ist definiert durch $g_f \circ f_f = ([\text{fail}, g] \circ f)_f$, was in Haskell

```
FailF g . FailF f = FailF (f >>> (arr Left ||| g))
```

entspricht. Tritt in `f` ein Fehler auf, wird dieser also wieder in ein `Left` eingepackt und zurückgegeben. Ansonsten wird das durch `|||` implizit aus dem `Right`-Konstruktor ausgepackte Ergebnis an `g` weitergereicht.

Abb. 8.1 zeigt die Komposition in $\mathbf{C}_f$ und deren Zurückführung auf ein Coprodukt in $\mathbf{C}$. Die Funktion, die f_f auf f abbildet, kann nicht Teil eines Funktors sein. Ein solcher müsste nämlich das Objekt B sowohl auf String $+ B$ (bei der Transformation von f als auch auf B selbst (bei g) abbilden.

Als Verwendungsbeispiel werden zwei Arrows definiert, die ungerade Zahlen bzw. Zahlen größer als 10 unverändert weiterreichen, ansonsten jedoch eine Fehlermeldung erzeugen:

```
checkOdd :: FailFunctor (->) Integer Integer
checkOdd = ifArrow (arr odd)
             id
             (fail "Odd number expected")

checkBig :: FailFunctor (->) Integer Integer
checkBig = ifArrow (arr (>10))
             id
             (fail "Number larger then 10 expected")
```

Hier wird als unterliegende Kategorie **Hask**, die Kategorie der puren Funktionen, verwendet und sowohl Ein- als auch Ausgabe sind vom Typ `Integer`. `ifArrow` ist eine in MagicL's Arrow-Bibliothek definierte Hilfsfunktion, die ein Prädikat und zwei Arrows erwartet, wovon bei Zutreffen des Prädikats der erste, ansonsten der zweite Arrow verwendet wird. Diese Arrows werden nun einmal per Komposition, welche sich in diesem Fall wie ein „Und“ verhält, und einmal mit dem Oder-Operator verknüpft:

```
checkOddAndBig = checkOdd >>> checkBig
checkOddOrBig  = checkOdd <+> checkBig
```

Mit `execFail` können aus den Arrows wieder normale Funktionen erzeugt werden, die im Fehlerfall einen Haskell-Error erzeugen. Diese Funktionen können nun mit einigen Beispielzahlen aufgerufen werden:

```
execFail checkOddAndBig 41    ==> 41
execFail checkOddAndBig 42    ==> Error: Odd number expected
execFail checkOddAndBig 9     ==> Error: Number larger then 10 expected
execFail checkOddAndBig 8     ==> Error: Odd number expected

execFail checkOddOrBig 41     ==> 41
execFail checkOddOrBig 42     ==> 42
execFail checkOddOrBig 9      ==> 9
execFail checkOddOrBig 8      ==> Error: Number larger then 10 expected
```

Interessant sind die Aufrufe mit 8, die in beiden Fällen zu verschiedenen Fehlermeldungen führt. Werden also zwei Arrows, die beide Fehler erzeugen, mit `>>>` verknüpft, erscheint die erste Fehlermeldung – mit `<+>` verknüpft hingegen die zweite. Dieses Verhalten entspricht in etwa der Short-Circuit-Evaluation von Und- und Oder-Operatoren in Programmiersprachen.

8.2 Arrows mit Zuständen

Zustände lassen sich ähnlich zu einem Arrow hinzufügen. Über Produkte wird der Zustandstyp S (z.B. eine Stream-Position oder der *Seed* eines Zufallszahlengenerators) an Domäne und Codomäne heran gehängt, was wieder in einer neuen Kategorie C_s versteckt wird. Ein Morphismus $f_s : A \to B$ der neuen Kategorie wird entsprechend abgebildet auf $f = A \times S \to B \times S$. Da hier im Gegensatz zum Fail-Funktor Domäne und Codomäne gleich abgebildet werden, ist diese Abbildung Teil eines Funktors $F_s : C_s \to C$. Abb. 8.2 zeigt die Komposition in $\mathbf{C}_s$ und die Rückführung auf die ursprüngliche Kategorie.

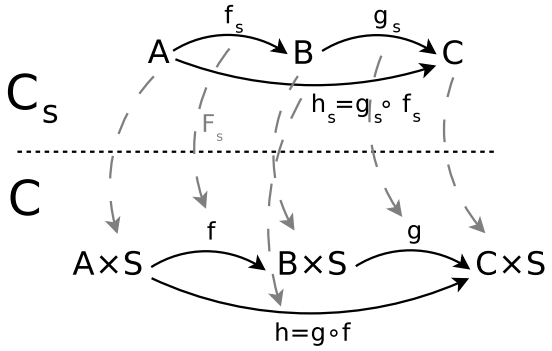


Abbildung 8.2: Komposition in $\mathbf{C}_s$ und Rückführung auf $\mathbf{C}$ durch F_s

In MagicL gibt es die Typklasse `StateFunctor`, die einen weiteren Typparameter `s` für den Zustand bekommt:

```
newtype StateFunctor s ar a b = StateF (ar (a, s) (b, s))

instance (Arrow ar) => Functor (StateFunctor s ar) ar where
    lift = StateF . first
```

Die „geliftete“ Version eines Arrows soll den Zustandsparameter unverändert weitergeben – dies entspricht genau der `first`-Funktion aus Abschnitt 6.4.1.

Das Lesen und Schreiben von Zuständen wird von den Funktionen `get` und `put` bereitgestellt, welche in der Typklasse `ArrowState` definiert werden:

```
class (Arrow ar) => ArrowState s ar | ar -> s where
  get :: ar a s
  put :: ar s ()

instance (Arrow ar) => ArrowState s (StateFunctor s ar)
  where
    get = StateF (arr (\ (_, state) -> (state, state)))
    put = StateF (arr (\ (state, _) -> ((), state)))
```

`get` ignoriert also den Eingabewert und gibt den aktuellen Zustand zurück, während `put` den übergebenen Zustand „abspeichert“ und (bis auf diesen) nichts zurückgibt.

Hier ein kleines Beispielprogramm, welches Zustände verwendet:

```
incState :: StateFunctor Integer (->) a a
incState = skip (get >>> arr (+1) >>> put)

double :: StateFunctor a (->) Integer Integer
double = arr (*2)

process :: StateFunctor Integer (->) () (Integer, Integer)
process = (get          >>>
           incState     >>>
           double       >>>
           incState     >>>
           double       >>>
           (id &&& get))

execState process 5     ==> (20, 7)
```

`incState` ist ein Arrow, der die normale Eingabe, welche von einem beliebigen Typ `a` sein kann, unverändert zurückgibt (dies erledigt `skip`), dabei allerdings den versteckt durchgereichten Zustand vom Typ `Integer` um eins erhöht. `double` hingegen benutzt den Zustand nicht – der diesmal einen beliebigen Typ haben kann – sondern verdoppelt die Eingabe. `process` ist ein zusammengesetzter Arrow, der keine Eingabe, jedoch einen vorhandenen Zustand erwartet und gleich ausliest, dann je zweimal die beiden anderen Arrows aufruft und zum Schluss in einem Tupel die normale Ausgabe sowie den Zustand zurückgibt. Nun wird `process` mittels `execState` in eine Funktion konvertiert, die ihr Argument als Zustand benutzt, und mit 5 als Parameter aufgerufen. Da `incState` und `double` keinerlei Wechselwirkungen miteinander aufweisen, ist der Effekt ein zweimaliges Verdoppeln der Eingabe (die am Anfang gleich dem Zustand 5 war) sowie ein Erhöhen des Zustandes um 2.

8.3 Der Parse-Funktor

Parser benötigen sowohl einen Zustand, der die Position im Eingabestream beinhaltet, als auch die Möglichkeit zu Scheitern sowie die Verwendung von Alternativen. Daher werden obige Kategorie-Erweiterungen zu einer Parser-Kategorie $\mathbf{C}_p = \mathbf{C}_{fs}$ zusammengesetzt, d.h. die ursprüngliche Kategorie wird zunächst um Fehler zu $\mathbf{C}_f$, danach um Zustände zu $\mathbf{C}_p$ erweitert. Die Reihenfolge hier ist wichtig, damit ein Morphismus $A \to B$ aus $\mathbf{C}_p$ auf $A \times S \to \text{String} + B \times S$ in $\mathbf{C}$ abgebildet wird und nicht auf $A \times S \to (\text{String} + B) \times S$, wie es andersherum wäre. Denn das Coprodukt muss „außen“ stehen, um im Kontrollfluss zuerst bearbeitet zu werden und somit ein Backtracking im Fehlerfall durch den Fail-Funktor zu ermöglichen.

Backtracking wird allerdings zu einem Problem, wenn sinnvolle Fehlermeldungen produziert werden sollen. Man stelle sich einen Parser für die (als regulärer Ausdruck beschriebene[2]) Grammatik `{.*}|[.*]` vor, die eine beliebige Zeichenkette in geschweiften

[2]Hierbei sollen eckige und geschweifte Klammern keine besondere Bedeutung besitzen, der Punkt für

oder eckigen Klammern beschreibt. Beim Eingabewort `{ABC` scheitert zunächst aufgrund der fehlenden schließenden Klammer die erste Alternative, woraufhin die zweite probiert wird. Das Wort beginnt aber nicht mit `[`, so dass der Parser beispielsweise mit der Meldung `"Expected [, got {"` scheitert. Diese Meldung beschreibt den Fehler schlecht. Stattdessen möchte man nach dem Lesen von `{` das Backtracking deaktivieren, denn bereits hier ist klar, dass die andere Alternative nicht mehr in Betracht kommt. Dies führt zu einer besseren Fehlermeldung wie `"Expected }, got end of stream"`.

Ein derartiger „nicht-auffangbarer“ Fehler lässt sich durch das Einbauen einer weiteren Möglichkeit des Scheiterns ermöglichen, wir definieren also $\mathbf{C}_p = \mathbf{C}_{ffs}$ und bieten eine Funktion `forceParser` an, die einen (Sub-)Parser derart modifiziert, dass, falls dieser scheitert, der Fehler in die „innere“ und damit nicht-recover-fähige Fail-Kategorie übernommen wird. Hiervon wird beispielsweise bei der Funktion `macro` (siehe Abschnitt 9.2.2) Gebrauch gemacht.

Der `ParseFunctor` ist in MagicL wie folgt definiert::

```
newtype ParseFunctor t ar a b =
  P (StateFunctor
     [t]
     (FailFunctor (FailFunctor ar))
     a
     b)
```

Als Zustandstyp wird ein Stream (als Liste repräsentiert) vom frei wählbaren Token-Typ `t` benutzt, so dass nicht nur Zeichenketten, sondern beliebige Listen durch Parser verarbeitbar sind. Dank der Implementation als Funktor ist die Kategorie, in der der Parser ausgeführt wird, frei wählbar. So können rein funktionale Parser in $\mathbf{Hask}_p$ konstruiert werden. Benötigt man aber Debug-Ausgaben (z.B. beim Suchen einer Endlosschleife) oder andere Seiteneffekte, kann $\mathbf{IO}_p$ benutzt werden. In MagicL werden diese Kategorien durch die Typaliase `FunParser` sowie `IOParser` bereitgestellt:

```
type FunParser t a b = ParseFunctor t (->) a b
type IOParser  t a b = ParseFunctor t IOArrow a b
```

Eine geeignete innere Kategorie könnte sogar interaktive Debugger, Netzwerktransparenz oder sonstige Features anbieten. Auch der variable Token-Typ erzeugt viele Möglichkeiten: Textdateien können mit `Char`, Bit-Streams mit `Bool` oder S-Expression-Streams mit `Sexp` verarbeitet werden.

8.4 Parser-Bibliothek

MagicL beinhaltet eine Parser-Bibliothek, in der u.a. viele simple Parser-Konstruktoren definiert sind. Hier ein paar Beipiele[3]:

- `empty :: ParseFunctor t ar a a`
  gibt seinen Eingabewert unverändert zurück, wenn der Stream leer ist.

ein beliebiges Zeichen und der Stern für beliebige Wiederholungen stehen.

[3] Abhängigkeiten der Typparameter werden hier der Übersichtlichkeit halber weggelassen.

- `takeWhen :: (t -> Bool) -> (t -> String) -> ParseFunctor t ar a t`
  testet den ersten Token mit einem Prädikat. Im Erfolgsfall (`True`) wird der Token zurückgegeben, ansonsten wird der Token an eine Funktion übergeben, die daraus eine Fehlermeldung aufbaut.

- `member :: [t] -> ParseFunctor t ar a t`
  testet, ob der erste Token in der übergebenen Liste vorkommt.

- `streamEq :: [t] -> ParseFunctor t ar a [t]`
  prüft, ob die ersten n Token mit den Elementen der übergebenen Liste übereinstimmen, und gibt diese im Erfolgsfall zurück. Für `t = Char` wird hier also ein String gelesen.

Alle diese Funktionen (und deren Negationen) geben Parser mit nützlichen Fehlermeldungen zurück.

Weiterhin gibt es viele Arrow-Kombinatoren, mit denen Parser, vergleichbar mit den Operationen der Erweiterten Backus-Naur-Form (EBNF) [23, S.43ff], verschaltet werden können. Mit den bisher eingeführten Operatoren sind bereits Konkatenation (`>>>`) und Alternative (`<+>`) abgedeckt, darüber hinaus gibt es für Instanzen von `ArrowPlus` unter anderem folgende Funktionen:

```
optional :: (...) => ar a b -> ar a (Maybe b)
optional f = (f >>^ Just) <+> constArrow Nothing
  where constArrow x = arr (\ _ -> x)

many :: (...) => ar a b -> ar a [b]
many f = many1 f <+> constArrow []

many1 :: (...) => ar a b -> ar a [b]
many1 f = consArrow f (many f)
  where consArrow :: ar a b -> ar a [b] -> ar a [b]
        consArrow f g = (f &&& g) >>^ (\ (x,xs) -> x : xs)

skip :: (...) => ar a b -> ar a a
skip f = (f &&& id) >>^ snd

sepBy :: (...) => ar a c -> ar a b -> ar a [b]
sepBy sep item =
  optional (consArrow item (many (skip sep >>> item))) >>^ unMaybeList
    where
      unMaybeList  Nothing  = []
      unMaybeList (Just xs) = xs
```

`optional` erzeugt einen Arrow, der, sollte der übergebene Arrow scheitern, `Nothing` zurückgibt – dabei aber „erfolgreich“ bleibt. Dies entspricht `[ ]` in EBNF. Hierbei erzeugt `constArrow Nothing` einen Arrow, der die Eingabe ignoriert und konstant `Nothing` zurückgibt. Eine (optionale) Wiederholung (in EBNF `{ }`) erzeugt `many`, `many1` verlangt

mindestens ein Vorkommen. Die Hilfsfunktion `consArrow` nimmt zwei Arrows an und erzeugt daraus einen, der das Element, welches der erste zurückgibt, vor die Liste hängt, die der zweite zurückliefert. `skip` ignoriert das Resultat eines Arrows und gibt stattdessen seine Eingabe zurück; Zustandsänderungen kann der ignorierte Arrow aber dennoch bewirken. `sepBy` ist ein komplizierteres Beispiel und ermöglicht das Parsen von Listen mit Trennzeichen. Dafür bekommt die Funktion zwei Arrows übergeben: `sep` soll ein Trennzeichen, `item` ein Element lesen. Es wird zunächst ein `item`, dann beliebig viele, von (ignorierten) `sep` angeführte `item`-Elemente gelesen. Das ganze ist noch in ein `optional` gebettet, damit auch eine leere Liste gelesen werden kann. Die Rückgabe wird mittels `unMaybeList` vom durch `optional` erzeugten `Maybe`-Datentyp befreit, so dass statt `Nothing` eine leere Liste zurückgegeben wird, wenn nichts gelesen werden kann.

8.5 Beispiel: Einlesen von CSV-Dateien

Als Beispiel wird ein Parser definiert, der CSV-Dateien (Comma-Separated Values) in Listen von Listen einliest. Eine CSV-Datei besteht aus vielen, durch `'\n'` getrennten Zeilen:

```
parseCSV = sepBy (eq '\n') line
```

Jede Zeile besteht aus durch Kommas getrennten Zellen:

```
line = sepBy (eq ',') cell
```

Eine Zelle beinhaltet eine beliebige Anzahl von Zeichen, die allerdings nicht die Trennzeichen sein dürfen:

```
cell = many (notMember ",\n")
```

Damit ist der Parser vollständig spezifiziert und kann mit einem Beispieldatensatz getestet werden:

```
runParser parseCSV "Erika,Mustermann,Karlsruhe\nSven,Schmidt,Mannheim"

-- ergibt [["Erika", "Mustermann", "Karlsruhe"]
           ["Sven",  "Schmidt",    "Mannheim"]]
```

8.6 Zusammenfassung

Parser werden in MagicL durch die Verschachtelung zweier Funktoren implementiert, die bestehende Arrows um bestimmte Möglichkeiten erweitern. Der `FailFunctor` realisiert die Möglichkeit, mit einer Fehlermeldung zu Scheitern und per Backtracking Alternativen zu probieren. Der `StateFunctor` setzt das Durchreichen von Zuständen um und wird benutzt, um den noch zu lesenden Stream als Liste zu verwalten. Der `ParseFunctor` kombiniert beides, wobei der `FailFunctor` doppelt verwendet wird, um auch Fehler zu ermöglichen, die nicht durch Backtracking zu beheben sind. Dies ist hilfreich, um teilweise sinnvollere Fehlermeldungen zu erhalten.

Eine Parser-Bibliothek stellt neben Konstruktoren wie `member` auch Kombinatoren wie `many` bereit, die eine EBNF-artige Formulierung ermöglichen. Damit ist die Anforderung einer Bibliothek für String-Parser erfüllt. Allerdings sind MagicL-Parser so allgemein definiert, dass neben Buchstaben auch andere Token-Typen verwendet werden können. Damit bildet die Parser-Bibliothek auch das Fundament für MagicL's Makrosystem, denn verallgemeinerte Makros werden in Abschnitt 9.2 als Parser auf S-Expressions definiert.

9 Weitere Komponenten

In diesem Kapitel werden die weiteren Kernkomponenten von MagicL vorgestellt. Zunächst wird die Generator-Bibliothek für Zeichenketten erläutert. Anschließend wird auf S-Expression-spezifische Funktionalität mit dem Schwerpunkt verallgemeinerter Makros eingegangen. Zuletzt wird die allgemeine Schnittstelle behandelt, mit der sich die verschiedenen Compiler-Typen in MagicL zusammenfügen lassen und feste Standardcompiler definiert werden können.

9.1 Generator-Bibliothek

Die meisten Frameworks für Code-Generierung erzeugen Quelltext aus String-basierten Templates und daher mit Low-Level-Methoden. Syntaktische Konstrukte werden daher oft wiederholt, was zu häufigen Syntaxfehlern und einem relativ hohen Wartungsaufwand führt. Es würde beispielsweise viel Arbeit bedeuten, alle in einer generierten Java-Klasse definierten Arrays in Collections umzuwandeln, wenn die `{A, B, C}`-Syntax überall direkt benutzt wurde. Hilfreich wäre es, wenn stattdessen überall ein Sub-Template `genList` aufgerufen würde. Nun müsste nur noch eine Stelle verändert werden um die erwünschte Anpassung durchzuführen. Diese Abstraktion ist auch mit einigen String-basierten Templates realisierbar. Ein weiteres Problem aber ist die Formatierung, denn auch generierter Quelltext wird eventuell gelesen werden. Ein Template für `for`-Schleifen kann den Schleifenkörper nicht korrekt einrücken, da der Einrückungslevel immer relativ zur – im Template unbekannten – äußeren Einrückung ist. Wenn vorhanden, kann der erzeugte Code daher an einen Formatierer wie Jalopy [26] übergeben werden. Für ein sprachunabhängiges Framework ist diese Option aber nicht zufriedenstellend.

MagicL verfolgt einen anderen Ansatz und definiert ein typisiertes Modell für formatierten Quelltext, welches durch eine Reihe von Generatorfunktionen konstruiert und anschließend von einem Compiler in eine Zeichenkette überführt wird. Hierbei handelt es sich um einen Nachbau des von Philip Wadler in [21, S.223ff] vorgestellten Pretty-Printers. Die elementaren Konstruktor-Funktionen sind:

```
newline :: Code
text    :: String -> Code
append  :: Code   -> Code -> Code
group   :: Code   -> Code
indent  :: Int    -> Code -> Code
```

`newline`, `text` und `append` sollten selbsterklärend sein. `group` versucht den übergebenen `Code` auf eine Zeile zu bekommen, indem es alle Zeilenumbrüche durch Leerzeichen ersetzt. Wird dabei allerdings die geforderte Zeilenlänge (im Standard-Compiler 70)

überschritten, wird die ursprüngliche Version mit Zeilenumbrüchen genommen. `indent` rückt den übergebenen `Code` bei jedem Zeilenumbruch um n Zeichen ein, z.B. würde

```
indent 2 (conc [text "hallo", newline, text "welt"])
```

den Text

```
hallo
  welt
```

ergeben. Dabei verbindet `conc :: [Code] -> Code` eine Liste mittels `append`. Darüber hinaus gibt es viele weitere allgemeine Funktionen wie

```
joinBy :: Code -> [Code]
```

die einen Separator benutzt, um eine Liste zusammenzufügen.

```
joinBy "; " [text "foo", text "bar", text "baz"]
```

würde also

```
foo; bar; baz
```

ergeben. Aus diesen allgemeinen Funktionen werden dann Abkürzungen für syntaktische Programmiersprachenelemente abgeleitet, wie `commaSep` für eine Komma-separierte Liste oder `parens`, `braces`, `brackets` und `angleBrackets` für die vier gängigen Arten der Klammerung.

Beispiel: Generierung von XML

Als Beispiel soll ein vereinfachter XML-Generator definiert werden. Als Eingabe wird dabei ein XML-Modell wie folgt definiert:

```
data XmlNode = Node String [Attribute] XmlInner
data Attribute = Attr String String
type XmlInner = Either [XmlNode] String
```

Ein Knoten besteht aus einem Namen, einer Attributliste und einem Inhalt. Jedes Attribut enthält einen Schlüssel und einen Wert. Das Innere eines Knotens kann entweder eine Liste von Kindknoten oder ein String sein. Ein Knoten soll wie folgt formatiert werden:

```
<node key1="value1" key2="value2">
  <inner-node></inner-node>
  <inner-node></inner-node>
</node>
```

Der Inhalt wird um zwei Zeichen eingerückt, es sei denn der Knoten passt komplett auf die aktuelle Zeile. Die Formatierungsfunktion ist durch folgendes Programm gegeben:

```
formatNode (XmlNode tagName attrs inner) =
  group (lines [indent2 (lines [openTag, innerCode]), closeTag])
  where openTag  = angleBrackets (words [text tagName, attrCode])
        closeTag = angleBrackets (append (text "/") (text tagName))
```

```
attrCode = words (map formatAttr attrs)
formatAttr (Attr key value) =
  conc [text key, text "=", string (text value)]
innerCode = case inner of
              Left childNodes -> lines (map formatNode childNodes)
              Right str       -> text str
```

Auf eine detaillierte Erklärung des Programms wird an dieser Stelle verzichtet. Ein einfacheres Beispiel wird mit dem Pretty Printer für S-Expressions in Abschnitt 9.2.1 ausführlich erläutert.

9.2 S-Expression-Bibliothek

S-Expressions sind in MagicL wie folgt definiert:

```
data Sexp = Symbol String
          | Node [Sexp]
```

Die S-Expression-Bibliothek enthält einen Parser und einen Generator, welche zwischen S-Expression-Syntax und dem Typen `Sexp` konvertieren können. Außerdem wird die Parser-Bibliothek um Funktionalität für Parser, die auf S-Expressions operieren, bereichert, wodurch Makros als Parser definiert werden können.

9.2.1 Ein- und Ausgabe von S-Expression-Syntax

Aufgrund der Einfachheit der S-Expression-Syntax ist es mit der Parser-Bibliothek möglich, in wenigen Zeilen einen Parser zu entwerfen, der als Text repräsentierte S-Expressions in die typisierte Repräsentation überführt:

```
whitespace = skip (many (member " \t\n"))
```

Zunächst wird Whitespace als beliebige Anhäufung von Spaces, Tabs und Newlines definiert, die ignoriert werden soll.

```
parseSymbol = many1 (notMember " \t\n()") >>^ Symbol
```

Ein Symbol ist eine Kette von Zeichen, die weder Whitespace noch Klammern sind. Dieses wird gleich an den `Symbol`-Konstruktor übergeben, um den Typ `Sexp` zu erhalten.

```
parseNode = skip (eq '(') >>> (many parseSexp >>^ Node) >>> skip (eq ')')
```

Ein Knoten ist eine von Klammern umgebene Liste von S-Expressions, die mittels `Node`-Konstruktor zu einem S-Expression wird.

```
parseSexp = whitespace >>> (parseSymbol <+> parseNode) >>> whitespace
```

Ein S-Expression ist nun entweder ein Symbol oder ein Knoten, wobei davor und danach Whitespace auftreten darf. Abb. 9.1 zeigt noch einmal den gesamten Parser. Ein Beispielaufruf könnte wie folgt aussehen:

```
whitespace  = skip (many (member " \t\n"))
parseSymbol = many1 (notMember " \t\n()") >>^ Symbol
parseNode   = skip (eq '(') >>> (many parseSexp >>^ Node) >>> skip (eq ')')
parseSexp   = whitespace >>> (parseSymbol <+> parseNode) >>> whitespace
```

Abbildung 9.1: Ein kompletter S-Expression-Parser in vier Zeilen

```
runParser parseSexp "(this is (an s-expr))"

ergibt:
Node [Symbol "this", Symbol "is", Node [Symbol "an", Symbol "s-expr"]]
```

Tritt beim Lesen ein Fehler auf, wird automatisch eine brauchbare Fehlermeldung ausgegeben, z.B.

```
runParser parseSexp "(test) )"  ==>  Empty stream expected: ")"
```

Dieser Parser, um S-Expression-spezifische Fehlermeldungen erweitert, wird in MagicL zum Einlesen aller S-Expression-DSLs verwendet.

Neben dem Parser enthält MagicL einen Generator für S-Expression-Syntax, welcher beispielsweise zum Debuggen neuer Compiler nützlich ist, da als Zwischenschritte erzeugte DSLs leserlich ausgegeben werden können. S-Expressions werden wie folgt formatiert: Passt der gesamte Ausdruck noch auf die aktuelle Zeile, werden normal Leerzeichen verwendet:

```
(a b (c d) e)
```

Wird die gewünschte Breite aber überschritten, soll jeder Teilausdruck auf einer eigenen Zeile stehen, wobei alle bis auf den ersten um zwei Zeichen eingerückt werden:

```
(a
  b
  (c d)
  e)
```

Dies lässt sich mit dieser rekursiven Funktion erreichen:

```
layoutSexp :: Sexp -> Code
layoutSexp (Symbol sym)    = text sym
layoutSexp (Node children) = format (map layoutSexp children)
  where format = group . parens . indent2 . lines
```

Um einen Knoten zu formatieren, werden zunächst die Unterknoten formatiert (`map layoutSexp children`) und jeweils auf eine eigene Zeile (`lines`) gesetzt, wobei nach Zeilenumbrüchen um zwei Zeichen eingerückt werden soll (`indent2`). Anschließend wird der Ausdruck geklammert (`parens`) und zuletzt überprüft `group`, ob der ganze Ausdruck nach Ersetzen der Zeilenumbrüche durch Leerzeichen noch auf die aktuelle Zeile passt; ansonsten werden die Zeilenumbrüche beibehalten.

9.2.2 Parser auf S-Expressions

Mit dem Begriff S-Expression-Parser werden in dieser Arbeit Parser bezeichnet, die Listen von S-Expressions als Eingabe und einen beliebigen Typ als Ausgabe haben – und nicht etwa der String-Parser für S-Expression-Syntax wie im vorigen Abschnitt. Entsprechend ist der Typ `SexpParser` in MagicL definiert als `IOParser` (für Seiteneffekten wie Debug-Ausgaben) vom Token-Typ `Sexp` ohne explizite Eingabe und mit beliebigem Rückgabewert:

```
type SexpParser a = IOParser Sexp () a
```

Die Parser-Bibliothek an sich kann bereits Streams von S-Expressions verarbeiten. Es gibt allerdings bisher keine komfortablen Operationen, um das Innere einer S-Expression zu testen oder zu verarbeiten. Da ein S-Expression selbst wieder eine Liste und damit ein Stream ist, bietet es sich an, diesen ebenfalls über einen „inneren Parser“ zu verarbeiten.

Beispielsweise könnte eine Liste von Personen folgenderweise repräsentiert sein[1]:

```
(person Franz)
(person Walter)
(person Heinz)
```

Ein Parser, der aus dieser Eingabe eine Liste von Namensstrings produzieren soll, würde mit der bisherigen Bibliothek so aussehen:

```
parsePerson :: SexpParser String
parsePerson = take >>^ proc >>> (fail ||| id)
  where proc (Node [Symbol "person", Symbol x]) = Right x
        proc y = Left ("Not a person: " ++ show y)
parsePersons :: SexpParser [String]
parsePersons = many parsePerson
```

Hierbei liest der Befehl `take` den nächsten Token bedingungslos ein.

Die Verarbeitung von Hand mittels `Node` und `Symbol` im Pattern-Matching ist relativ umständlich. Deswegen gibt es eine Reihe nützlicher Hilfsfunktionen für die Konstruktion von S-Expression-Parsern:

- `takeSexp` liest eine S-Expression vom Stream und wandelt diese für die Unterscheidung zwischen Symbolen und Knoten mittels `|||` nach `Either String [Sexp]` um.
- `takeSymbol` erwartet ein Symbol und gibt dieses als String zurück, ansonsten wird eine Fehlermeldung ausgegeben.
- `symbolMacro name = skip (eq (Symbol name))` akzeptiert nur das Symbol mit Namen `name`.
- `takeNode` erwartet einen Knoten und gibt diesen als Liste von S-Expressions zurück.

[1] Dieses Format ist unnötig redundant und sollte vermutlich besser durch `(persons Franz Walter Heinz)` ersetzt werden – aber es handelt sich ja lediglich um ein Beispiel.

- `compNode innerComp` verarbeitet einen Knoten, indem der übergebene Parser auf das Knoteninnere angewendet wird. Da Parser in MagicL gewöhnlich als Compiler definiert werden (siehe Abschnitt 9.3), heißt diese Funktion nicht `parseNode`.

- `macro name innerComp` konstruiert nun einen Parser, der wie ein Makro das erste Element des Knotens mit dem Symbol `name` vergleicht und im Erfolgsfall die restlichen Elemente mit `innerComp` verarbeitet. Danach wird sichergestellt, dass der Knoten auch wirklich vollständig gelesen wurde. Will man dies nicht, da man den Rest ignoriert, gibt es die Funktion `looseMacro` mit identischer Signatur.

Das obige Beispiel lässt sich nun umschreiben:

```
parsePerson  = macro "person" takeSymbol
parsePersons = many parsePerson
```

`macro` benutzt `forceParser` aus Abschnitt 8.3, um das Backtracking auszuhebeln sobald das Symbol übereinstimmt, was wieder für bessere Fehlermeldungen sorgt. Allerdings bedeutet dies, dass Code wie

```
macro "foo" (symbolMacro "bar") <+> macro "foo" (symbolMacro "baz")
```

nicht erwartungsgemäß funktioniert:

```
(foo bar)   => ()  d.h. Erfolg, kein Ergebnis
(foo baz)   => Symbol bar expected: baz
```

Es sollte deshalb immer nur ein Makro für jeden Begriff benutzt und eine etwaige Oder-Verknüpfung ins Innere verlegt werden – hier also:

```
macro "foo" (symbolMacro "bar" <+> symbolMacro "baz")
```

9.3 Generische Compiler-Schnittstelle

Mit Parsern und Funktionen gibt es derzeit in MagicL zwei Typen, mit denen Compiler realisiert werden können – es könnten auch zukünftig noch weitere hinzukommen. Damit verschieden implementierte miteinander kombiniert werden können, müssen diese in einen allgemeineren Compiler-Typ überführt werden. Der allgemeinste Compilertyp in MagicL ist `IOArrow a b`, die Menge der Funktionen mit möglichen Nebeneffekten, dessen Elemente mit Arrow-Kombinatoren wie `>>>` verknüpft werden können. Alle Typen `x`, die sich mit der polymorphen Funktion `toIO` in IO-Arrows umwandeln lassen, implementieren die Typklasse `Executable x a b`:

```
class Executable x a b | x -> a b where
  toIO :: x -> IOArrow a b
```

Dabei ist `x` selbst ein parametrisierter Typ, der die Ein- und Ausgabetypen `a` und `b` vollständig festlegt. Die einfachsten Instanzen von `Executable` werden durch `IOArrow` selbst sowie die pure Funktionen gebildet:

```
instance Executable (IOArrow a b) a b where
  toIO = id

instance Executable (a -> b) a b where
  toIO f = Kleisli (return . f)
```

Um hier einen `IOArrow` in einen `IOArrow` zu überführen, reicht die Identitätsfunktion. Eine pure Funktion hingegen wird zunächst mit `return` in die `IO`-Monade überführt und dann durch die Kleisli-Transformation in einen Arrow umgewandelt.

Auch Parser sind ausführbar, sofern der zugrundeliegende Arrow-Typ es ist:

```
instance (...) => Executable (ParseFunctor t ar () a) [t] a where
  toIO = toIO . execParser
```

Ein Parser von `()` nach `a` mit Token-Typ `t` wird demnach beim Ausführen zu einem `IOArrow` von `[t]` nach `a`. Der bei der Parser-Komposition noch implizite Stream-Parameter wird hier also explizit gemacht und der vom Parser implizit zurückgegebene Stream verworfen. Sollte es erforderlich sein, einen Parser mit weiteren Parametern aufzurufen oder auch den Rest-Stream zu erhalten, muss statt `Executable` direkt mit dem Parser-Typ gearbeitet werden.

Die Ausgabe von einem Parser kann daher wie folgt an einen als Funktion spezifizierten Compiler weitergeleitet werden:

```
toIO someParser >>> toIO someFunction
```

Darüber hinaus bietet die Compilerschnittstelle die Möglichkeit, spezielle Compiler fest mit einer Kombination von Ein- und Ausgabetyp zu verbinden, so dass für den Aufruf eines Compilers weder Name noch Implementationstyp bekannt sein müssen. In MagicL sind Parser und Generator für S-Expression-Syntax fest verdrahtet, daher kann der (in einen IO-Arrow umgewandelte) Parser mit
`compile :: IOArrow String Sexp` referenziert werden. Sind Ein- und Ausgabetyp durch den Kontext klar, kann die Typanschrift weggelassen werden.

Das Festlegen eines Standardcompilers von `a` nach `b` durch den Implementationstyp `x`, welcher `Executable` implementieren muss, erfolgt mit der Typklasse `Compilable`:

```
class (Executable x a b) => Compilable x a b | a b -> x where
  comp :: x
```

Da es nur einen ausgezeichneten Compiler von `a` nach `b` geben kann, legen `a` und `b` gemeinsam `x` fest. Gleichzeitig bestimmt `x` auch `a` und `b`, da dies bereits in `Executable` festgelegt ist. Es handelt sich also um eine 1:1-Beziehung.

Festverdrahtete Compiler können ebenfalls mittels `toIO` in IO-Arrows konvertiert werden. Dies geschieht bereits automatisch beim Aufruf mit `compile` aus der Typklasse `Compiler`, die automatisch für jede Instanz von `Compilable` instanziiert wird:

```
class Compiler a b where
  compile :: IOArrow a b

instance (Compilable x a b) => Compiler a b where
  compile = toIO comp
```

Als Beispiele können der Generator für S-Expression-Syntax aus Abschnitt 9.2.1 mit

```
instance Compilable (Sexp -> String) Sexp String where
  comp x = layout 70 (layoutSexp x)
```

und der entgegengesetzte Parser mit

```
instance Compilable (FunParser Char () Sexp) String Sexp where
  comp = parseSexp
```

als Standardcompiler festgelegt werden, woraufhin beide mit typisierten `compile`-Aufrufen als IO-Arrows angesprochen werden können. Manchmal ist es auch nützlich, den noch nicht mit `toIO` umgewandelten Compiler durch einen typisierten Aufruf von `comp` zu referenzieren.

Da Listen- und Maybe-Datentypen von Parsern fast immer mit `many` und `optional` verarbeitet werden, stellt MagicL mit jeden Standardcompiler nach `a` automatisch auch Standardparser nach `[a]` und `Maybe a` bereit, die auf diese Weise aus dem ursprünglichen Parsern erzeugt wurden:

```
instance (Compilable (ParseFunctor t ar () a) [t] a, ...) =>
  Compilable (ParseFunctor t ar () [a]) [t] [a] where
    comp = many comp

instance (Compilable (ParseFunctor t ar () a) [t] a. ...) =>
  Compilable (ParseFunctor t ar () (Maybe a)) [t] (Maybe a) where
    comp = optional comp
```

Die `comp`-Aufrufe auf der rechten Seite beziehen sich auf den ursprünglichen Parser, welcher von Haskell aus dem Typkontext zugeordnet wird. Damit können unter anderem auch Streams von S-Expressions durch den Aufruf von

```
compile :: IOArrow String [Sexp]
```

geparst werden, obwohl nur der Parser für einzelne S-Expressions spezifiziert wurde.

Leider ist es mit MagicL nicht möglich, sogar Kombinationen aus Compilern selbstständig zu finden. Hierfür müsste das Haskell-Typsystem eine Prolog-artige Tiefensuche unterstützen, was zumindest derzeit nicht der Fall ist. Auf der Compiler-Schnittstelle bauen einige Hilfsfunktionen für das konkrete Aufrufen von Compilern auf:

Um die Kompilation von Textdateien zu vereinfachen, wird die Funktion `compileStr` definiert, welche ein `Executable` von `a` nach `b` in eine imperative Haskell-Funktion von `String` nach `String` umwandelt, falls bereits Standardcompiler von `String` nach `a` und von `b` nach `String` bekannt sind.

```
compileStr :: (Compiler String a, Executable x a b, Compiler b String) =>
              x -> String -> IO String
compileStr f = runKleisli $ compile >>> toIO f >>> compile
```

Beispielsweise kann `a` für `Sexp` und `b` für `Code` stehen, da bereits Standard-Compiler von `String` nach `Sexp` (der normale S-Expression-Parser) sowie von `Code` nach `String` (der Layouter für eine Zeilenlänge von 70 Zeichen) existieren. Mit `compileStr layoutSexp`

könnte daher der S-Expression-Syntax-Generator in eine (imperative) Funktion umgewandelt werden, die zunächst aus einem String einen S-Expression parst, anschließend `layoutSexp` aufruft und das Ergebnis für 70 Zeichen layoutet. Man beachte, dass Parser und Generator automatisch durch die Typinferenz gefunden werden und nicht beim Aufruf angegeben werden müssen.

Für die Erzeugung von Compilern, die als eigenständiges Programm laufen können, benutzt `compiler` stdin als Eingabe- und stdout als Ausgabe-Stream:

```
compiler :: (Compiler String a, Executable x a b, Compiler b String) =>
            x -> IO ()
compiler f = do
  input <- getContents
  output <- compileStr f input
  putStr output
```

Um aus dem S-Expression-Formatierer ein Kommandozeilenprogramm zu erstellen, muss lediglich ein Modul mit einem Compiler als `main`-Funktion erstellt werden:

```
module SexpFormatter where
-- einige Import-Anweisungen

main = compiler layoutSexp
```

Dieses Modul kann dann kompiliert und von der Kommandozeile aufgerufen werden, wobei Ein- und Ausgabedateien per Umleitung angegeben sind:

```
bash$ ghc SexpFormatter.hs
bash$ ./SexpFormatter <inFile >outFile
```

9.4 Zusammenfassung

Die Erzeugung von Quelltext erfolgt in MagicL mit einer Generator-Bibliothek, indem funktional ein typisiertes `Code`-Modell erzeugt und anschließend mittels `layout` für eine bestimmte Zeilenlänge formatiert wird. Damit wurde unter anderem ein Generator für S-Expression-Syntax implementiert, mit dem generierter S-Expression-Code benutzerfreundlich formatiert werden kann. Darüber hinaus enthält die S-Expression-Bibliothek neben einem Parser für S-Expression-Syntax, der praktisch die Umkehrfunktion zum Generator ist, eine Reihe von Funktionen, die die Parser-Bibliothek um Funktionalität für das Einlesen von S-Expressions bereichert. Ein Beispiel hierfür ist `macro`, welches einen Parser erzeugt, der wie Lisp-Makros das erste Symbol eines Ausdrucks auf Übereinstimmung prüft. Damit ist die Anforderung von S-Expression-Verarbeitung durch verallgemeinerte Makros teilweise erfüllt. Was noch fehlt sind der Quasiquote-Operator zur praktischen S-Expression-Generierung und die Möglichkeit, automatisch rekursiv absteigende Makros wie in Lisp zu definieren. Außerdem ist die Metasprache noch Haskell und nicht selbst eine S-Expression-DSL, so dass es nicht möglich ist, Compiler selbst zu generieren und die Metasprache so zu erweitern.

Um die Anforderung der Kombinierbarkeit beliebiger Compiler zu entsprechen, stellt MagicL die Typklasse `Executable` bereit, mit der Parser und funktional spezifizierte Compiler in den Typ `IOArrow` überführt und anschließend mit den gewöhnlichen Arrow-Kombinatoren miteinander verknüpft werden können. Darüber hinaus kann jeder Kombination von Ein- und Ausgabetyp mit Hilfe der Typklasse `Compilable` ein Standardcompiler fest zugeordnert werden, so dass dieser anschließend durch einen typisierten Aufruf von `compile` als `IOArrow` angesprochen werden kann. Dieses zusätzliche Feature kommt in den Anforderungen nicht vor und kann als experimentell betrachtet werden.

10 Haskell-DSL

Die meisten Anforderungen sind mit den vorhergehenden Kapiteln bereits erfüllt worden. Was noch fehlt ist eine S-Expression-basierte DSL für die Definition von Compilern, die Konstrukte wie den Quasiquote-Operator, aber auch eine komplette Metaprogrammiersprache enthält. Damit in der DSL die in Haskell geschriebenen MagicL-Bibliotheken verwendet werden können, muss diese selbst nach Haskell übersetzt werden. Aus diesem Grund wird zunächst eine Haskell-DSL definiert, die Haskell möglichst eins zu eins abbildet. Diese kann dann direkt als Metasprache verwendet werden, eignet sich aber auch für die S-Expression-gestützte Generierung von Haskell-Code, wodurch in Kapitel 11 unter anderem der Quasiquote-Operator realisiert wird. Dieses Kapitel spezifiziert zunächst die Haskell-DSL genauer und stellt anschließend einen Compiler für diese vor, welcher ein typisiertes Haskell-Modell als Zwischenschritt benutzt und daher in zwei Kompenenten aufgeteilt wurde. Typisierte Modelle erhöhen den Entwicklungsaufwand[1], führen aber zu früherer Fehlererkennung in Compilern und ermöglichen feste Standardcompiler. Der Haskell-DSL-Compiler beschreitet diesen Weg und dient damit gleichzeitig als Anschauungsbeispiel für die Benutzung typisierter Modelle. Rein S-Expression-gestützte Kompilation hingegen wird in Kapitel 11 behandelt.

10.1 Spezifikation

Die Haskell-DSL soll im Wesentlichen eine in S-Expression gegossene Version von Haskell sein mit nur wenigen Anpassungen, wo diese sinnvoll erscheinen. Eine vollständige Abdeckung aller Haskell-Konzepte muss im Prototyp nicht gegeben sein, da diese bei Bedarf nachgepflegt werden können. Zudem sind einige Konzepte wie `let` und `where` redundant, so dass nur eines von beiden benötigt wird.

10.1.1 Funktionen

Funktions- oder Konstruktoraufrufe müssen in der DSL immer geklammert werden, können aber unverändert nach Haskell übernommen werden:

```
(myFun 1 4 52)              Haskell-DSL
=>
(myFun 1 4 52)              Haskell
```

[1] Die zukünftige Entwicklung einer geeigneten DSL zur Modellbeschreibung, aus der sowohl das typisierte Modell als auch Compiler generiert werden, könnte den Entwicklungsaufwand wieder reduzieren.

Die Pseudo-Syntax `=>` steht für „soll übersetzt werden nach“. Haskell-Infixoperatoren werden in Präfixnotation überführt und können in dieser beliebig viele Argumente annehmen wie in Lisp üblich:

```
(+ 1 4 (- 52 10 4))          Haskell-DSL
=>
(1 + 4 + (52 - 10 - 4))      Haskell
```

Als Infixoperatoren werden Symbole interpretiert, die ausschließlich aus Zeichen der Liste `!$%&/=?*+-.:<|>^` bestehen. Kommt ein solcher Operator jedoch nicht am Anfang eines Ausdrucks, sondern im Inneren vor, ist die (binäre) Funktion selbst gemeint, was in Haskell der Präfixformulierung durch Klammerung entspricht:

```
(foldr1 * xs)                Haskell-DSL
=>
(foldr1 (*) xs)              Haskell
```

Dies ergibt 24, falls `xs` den Wert `[1, 2, 3, 4]` hat. Die Definition neuer benannter Funktionen erfolgt weiterhin mit `=`, wobei mit `where` weiterhin lokale Definitionen eingeführt werden können:

```
(= (sumOfSquares x y)
   (+ x2 y2)
 (where                      Haskell-DSL
  (= x2 (* x x))
  (= y2 (* y y))))

=>

sumOfSquares x y = (x2 + y2)
  where
    x2 = (x * x)             Haskell
    y2 = (y * y)
```

Anonyme Funktionen werden mit `fun` definiert:

```
(map
  (fun x (+ x 5))            Haskell-DSL
  xs)

=>

(map
  (\ x -> (x + 5))           Haskell
  xs)
```

Mehrere Parametern können in der Form `(args x y z)` notiert werden.

10.1.2 Patterns

Wie in Haskell können Konstrukturen mittels Pattern Matching zerlegt werden:

```
(= (countMaybe (Just x)) x)
(= (countMaybe Nothing)  0)      Haskell-DSL

=>

countMaybe (Just x) = x
countMaybe Nothing  = 0          Haskell
```

Für Listen etc. stehen die „Pseudo-Konstruktoren“ `List`, `Cons`, `Tuple` und `Str` bereit, die wie die entsprechenden Haskell-Konstrukte sowohl zum Erzeugen als auch zum Matching verwendet werden können:

```
(= (listTo3Tuple (List x y z))
   (Tuple x y z))

(= (dropTwo (Cons x y xs))             Haskell-DSL
  xs)

(= helloWorld (Str Hello World))

=>

listTo3Tuple [x, y, z] = (x, y, z)

dropTwo (x : y : xs) = xs              Haskell

helloWorld = "Hello World"
```

Im Gegensatz zu richtigen Konstruktoren können diese Operatoren eine beliebige Anzahl von Parametern verarbeiten.

10.1.3 Typen

Typen werden eins zu eins in die Haskell-DSL übernommen, dabei werden parametrisierte Typen geklammert. Typdeklarationen erfolgen mit `hasType` anstelle des `::`-Operators in Haskell:

```
(hasType x Int)
                                       Haskell-DSL
(hasType m (Maybe a))

=>

x :: Int
                                       Haskell
m :: Maybe a
```

Funktionstypen werden mit `Fun`, Listentypen mit `List` denotiert. Letzteres erwartet im Gegensatz zum gleichnamigen Konstruktor nur einen Parameter und verhält sich somit wie der `[]`-Operator in Haskell:

```
(hasType map (Fun (Fun a b) (List a) (List b)))
=>
map :: (a -> b) -> [a] -> [b]
```

Typabhängigkeiten können mit `dep` eingebunden werden:

```
(hasType print (dep (Show a)) (Fun a (IO Unit)))
=>
print :: (Show a) => a -> IO ()
```

Der Typ `Unit` entspricht in Haskell `()` und damit `void` in anderen Sprachen. Typklassen und –instanzen sind ebenfalls direkt übernommen worden:

```
(class
  (MyClass a)
  (where
    (hasType printIt (Fun a (IO Unit)))
    (hasType isEqual (Fun a a Bool))))
                                                    Haskell-DSL
(instance (dep (Show a) (Eq a))
  (MyClass (List a))
  (where
    (= (printIt x)   ...)
    (= (isEqual x y) ...)))

=>

class MyClass a
  where
    printIt :: a -> IO Unit
    isEqual :: a -> a -> Bool
                                                    Haskell
instance (Show a, Eq a) => MyClass [a]
  where
    printIt x   = ...
    isEqual x y = ...
```

10.1.4 Module

Am Anfang jeder Haskell-Datei wird gewöhnlich das aktuelle Modul benannt und eine Liste referenzierter Module importiert. Darüber hinaus kann eine Liste zu exportierender Symbole angegeben werden, welche dann in anderen Modulen sichtbar sind. Ansonsten sind alle Symbole sichtbar. Um Namenskonflikte auflösen zu können, gibt es vier mögliche Arten ein Modul zu importieren:

- Es können alle (im anderen Modul exportierten) Symbole verfügbar gemacht werden.
- Es können nur bestimmte Symbole importiert werden.
- Es können alle Symbole bis auf Bestimmte importiert werden.
- Das Modul kann qualifiziert importiert werden, so dass alle Symbole daraus über einen vorangestellten Modul-Kurznamen referenziert werden können.

Die folgende Moduldefinition zeigt alle diese Möglichkeiten in der Haskell-DSL und die erforderliche Übersetzung nach Haskell:

```
(module MyModule
  (export myFun MyClass MyDataType)
  Prelude
  (Control.Category (only id))                          Haskell-DSL
  (Control.Monad (hiding liftM forM))
  (Control.Exception (qualified E)))

=>

module MyModule (myFun, MyClass, MyDataType) where
import Prelude
import Control.Category (id)
import Control.Monad hiding (liftM, forM)             Haskell
import qualified Control.Exception as E
```

10.2 Typisiertes Modell

Als erster Schritt ist das Haskell-Modell zu entwickeln. Der Einfachheit halber wird hier nur ein kleiner Ausschnitt des Modells vorgestellt. Alle nicht näher beschriebenen Datenstrukturen können dem MagicL-Quelltext entnommen werden.

Eine Haskell-Codedatei enthält eine optionale Modulbeschreibung sowie eine Liste von Toplevel-Definitionen:

```
data Haskell = Haskell (Maybe Module) [Toplevel]
```

Man beachte, dass hier sowohl der Typ als auch der Konstruktor `Haskell` heißen – ein übliches Vorgehen, falls ein Datentyp nur einen Konstrukter besitzt.

Eine Modulbeschreibung besteht aus einem Namen, einer optionalen Liste exportierter Symbole sowie einer Liste von Imports, wobei hier obige vier Arten vorkommen können:

```
data Module = Module String (Maybe Export) [Import]
data Export = Export [String]
data Import = Import String ImportArgs
data ImportArgs = Simple
                | Qualified String
```

```
                | Only [String]
                | Hiding [String]
```

Auf Toplevel-Ebene können Typzuweisungen, Funktionsdefinitionen, Typaliase sowie Deklarationen von Datentypen, Klassen und Instanzen stehen:

```
data Toplevel = TopHasType HasType
              | TopDef Def
              | TopTypeAlias TypeAlias
              | TopData Data
              | TopClass Class
              | TopInstance Instance
```

Hier wird lediglich auf Funktionen weiter eingegangen. Die linke Seite der Funktionszuweisung besteht aus einem Pattern, die rechte aus einem Ausdruck. Zusätzlich können mittels `where` lokale Definitionen erfolgen.

```
data Def = Def Pattern Expr (Maybe Where)
```

Ein Pattern kann eine Liste, ein Tupel, ein String oder ein Funktionsaufruf[2] sein.

```
data Pattern = ListPattern [Pattern]
             | TuplePattern [Pattern]
             | ConsPattern [Pattern]
             | StringPattern String
             | CallPattern Call
```

Ein Ausdruck kann eine Lambda-Funktion definieren, ein `do`-Ausdruck für Monaden[3], eine Typdeklaration oder ein Pattern sein. Die Parameterliste anonymer Funktionen benutzt ebenfalls Patterns.

```
data Expr = LambdaExpr  [Pattern] Expr
          | DoExpr      [DoCmd]
          | TypeExpr    HasType
          | PatternExpr Pattern
```

Als nächstes wird ein Parser benötigt, der die DSL in das Haskell-Modell überführt.

10.3 DSL-Parser

Compiler und Parser für größere, aus vielen Datentypen zusammengesetzte Modelle sollten möglichst immer in kleine Kompilationseinheiten (in Form von Arrows) aufgeteilt werden, die jeweils nur für einen Datentyp zuständig sind. Diese können auf zwei Arten definiert werden: Zum einen können benannte Arrows verwendet und von Hand verknüpft werden, was für den Haskell-DSL-Parser wie folgt aussehen könnte:

[2] Streng genommen dürfen in Patterns nur Konstruktoren (großgeschrieben) aufgerufen werden, in Ausdrücken hingegen Konstruktoren und Funktionen. Dies wird im Modell zur Vereinfachung nicht unterschieden, könnte allerdings beim Einlesen überprüft werden.

[3] Die Umsetzung der `do`-Notation in der DSL wird in diesem Kapitel nicht näher erläutert. Auch diese ist allerdings sehr nah am Original.

```
compHaskell = liftA2 Haskell (optional compModule) (many compToplevel)

compModule = macro "module" (liftA3 Module
                                    compString
                                    (optional compExport)
                                    (many compImport))

compToplevel = (liftA1 TopHasType   compHasType)   <+>
               (liftA1 TopDef       compDef)       <+>
               (liftA1 TopTypeAlias compTypeAlias) <+>
               (liftA1 TopData      compData)      <+>
               (liftA1 TopClass     compClass)     <+>
               (liftA1 TopInstance  compInstance)
```

Kombinatoren der Form `liftA`n erzeugen aus einer puren Funktion mit n Parametern sowie n Arrows einen Arrow, der diese Arrows nacheinander aufruft und deren Rückgaben im Erfolgsfall an die Funktion weiterreicht. Um optionale Ausdrücke und Listen einzulesen, werden die benutzten Parser mittels `many` und `optional` transformiert.

Alternativ ist es auch möglich, die Typklasse `Compilable` aus Abschnitt 9.3 zu verwenden und den polymorphen Arrow `comp` zu überladen. Dadurch erfolgt die Auswahl des passenden Parsers zur Kompilationszeit mittels Typinferenz anhand der Modelldefinitionen. Dies hat unter anderem den Vorteil, dass Aufrufe von `many` und `optional` nicht mehr nötig sind. Andererseits bedeuten die vielen Instanzdeklarationen einen erhöhten Schreibaufwand[4]. Der Haskell-Parser ist auf diese Art implementiert worden. Wie das Modell wird dieser hier nur ausschnittsweise vorgestellt:

```
instance Compilable (SexpParser Haskell) [Sexp] Haskell where
    comp = liftA2 Haskell comp comp

instance Compilable (SexpParser Module) [Sexp] Module where
    comp = macro "module" (liftA3 Module comp comp comp)

instance Compilable (SexpParser Export) [Sexp] Export where
    comp = macro "export" (liftA1 Export comp)



instance Compilable (SexpParser Import) [Sexp] Import where
    comp  = (comp >>^ \ n -> Import n Simple) <+>
             compNode (liftA2 Import comp compArgs)
        where compArgs =
                  ((empty >>> constArrow Simple)               <+>
                   (macro "qualified" comp >>> arr Qualified)  <+>
                   (macro "only" (many comp) >>> arr Only)     <+>
                   (macro "hiding" (many comp) >>> arr Hiding))
```

[4] Auch hier wäre also eine DSL wünschenswert, die diese Arbeit sowie ein automatisches Funktionsliften erledigt.

```
instance Compilable (SexpParser Toplevel) [Sexp] Toplevel where
    comp = (liftA1 TopHasType comp)   <+>
           (liftA1 TopDef comp)       <+>
           (liftA1 TopTypeAlias comp) <+>
           (liftA1 TopData comp)      <+>
           (liftA1 TopClass comp)     <+>
           (liftA1 TopInstance comp)
```

Der Import-Parser ist so konstruiert, dass als Import sowohl Symbole, welche automatisch in einen `Simple`-Import übersetzt werden, als auch Listen übergeben werden können. Der Aufruf `comp >>^ \ n -> Import n Simple` liest ein Symbol ein[5] und leitet dies an eine anonyme Funktion weiter, die den `Import`-Aufruf erzeugt.

Als Beispiel sei folgende Moduldefinition gegeben:

```
(module MyModule
  Data.List
  (Prelude (hiding id Functor))
  (Control.Exception (qualified E)))
```

Hier wird ein Modul `MyModule` definiert, welches keine Exports angibt und drei Module importiert: `Data.List` wird komplett importiert, während aus `Prelude` die Funktion `id` und der Datentyp `Functor` ignoriert werden, um einen Namenskonflikt mit eigenen Definitionen gleichen Namens zu vermeiden. Das Modul `Control.Exception` wird benannt als `E` importiert, so dass beispielsweise die Funktion `throw` mittels `E.throw` angesprochen werden kann.

Der Parser erzeugt daraus nun ein typisiertes Haskell-Objekt:

```
(Module "MyModule"
         Nothing
         [Import "Data.List" Simple,
          Import "Prelude" (Hiding ["id", "Functor"]),
          Import "Control.Exception" (Qualified "E")])
```

Dieses Objekt kann nun vom Haskell-Generator in Quelltext überführt werden.

10.4 Haskell-Generator

Für die eigentliche Quelltext-Erzeugung wird die Generator-Bibliothek verwendet, um das Haskell-Modell in den Datentyp `Code` aus Abschnitt 9.1 umzuwandeln, woraus anschließend der Quelltext erzeugt wird. Dieser Kompilationsschritt kann ebenfalls auf verschiedene Arten durchgeführt werden. Am simpelsten und pragmatischsten wäre wohl eine direkte Verwendung in Haskell-Funktionen, die sich gegenseitig referenzieren. Stattdessen wurde der Compiler allerdings wieder anhand der Typklasse `Compilable` konstruiert, um zu demonstrieren, wie neben Parsern auch Funktionen in diesem Rahmen

[5] MagicL beinhaltet einen Standard-Parser von `Sexp` nach `String`, welcher nur Symbole akzeptiert und ebenfalls durch Typinferenz aus `comp` erzeugt wird.

verwendet werden können. Generell können alle Typen verwendet werden, für die die Klasse `Executable` aus Abschnitt 9.3 implementiert ist. Dies bedeutet wieder einen erhöhten Aufwand, da für jede Funktion, die einen Datentyp verarbeitet, eine umständliche Instanziierung von `Compilable` benötigt wird. Dies wird auch durch den Vorteil, wieder eine generische `comp`-Funktion benutzen zu können, vermutlich nicht ausgeglichen. Im Gegensatz zu Parsern gibt es hier auch keine Standard-Compiler für Listen und `Maybe`-Datentypen, da diese nicht zwangsläufig für jede Sprache auf die gleiche Art verarbeitet werden. Listen von Funktionen beispielsweise werden im Haskell-Modell auf Toplevel-Ebene als Absätze mit Leerzeilen formatiert, Listen von lokalen Definitionen hingegen als Zeilen ausgegeben und Listen von Tupel-Elemente mittels Kommas auf einer einzigen Zeile verknüpft. Der eigentliche Vorteil dieses Ansatzes ist die Einheitlichkeit, die damit erreicht wird, wenn jegliche Verarbeitung typisierter Modelle in einem gemeinsamen Interface angesprochen wird. Dies ist auch nützlich, um derartigen Code später aus einer abstrakteren DSL generieren zu können.

Konkret wird ein Haskell-Objekt in eine Reihe von Absätzen formatiert – das geschieht mittels `paragraphs`, was je zwei Zeilenumbrüche zwischen seine Argumente einfügt. Der erste Absatz ist die kompilierte Modulbeschreibung (falls vorhanden). Anschließend kommen alle kompilierten Toplevel-Konstrukte:

```
instance Compilable (Haskell -> Code) Haskell Code where
  comp (Haskell mayMod tops) =
    paragraphs $ compMod mayMod ++ map comp tops
      where compMod Nothing  = []
            compMod (Just m) = [comp m]
```

Um den Rahmen dieses Kapitels nicht zu sprengen, wird hier ebenfalls nur die Verarbeitung der Modulbeschreibung vertieft. Die Modulbeschreibung wird mit `lines` auf mehrere Zeilen verteilt:

```
instance Compilable (Module -> Code) Module Code where
  comp (Module name exports imports) =
    (lines $
     [words $ [text "module", text name] ++
              compExports exports ++
              [text "where"]]
     ++ map comp imports)
    where compExports Nothing              = []
          compExports (Just (Export funs)) = [tuple $ map text funs]
```

Das Schlüsselwort `module`, der Modulname, die Exports (falls vorhanden) sowie das Schlüsselwort `where` kommen in die erste Zeile – jeweils durch Leerzeichen getrennt, was durch `words` geschieht. Alle Imports belegen ebenfalls je eine Zeile. Exportierte Symbole werden durch die Funktion `tuple` mit Kommas getrennt und zusammen in runde Klammern gesetzt. Um streng einheitlich zu bleiben, müsste die Umwandlung von `Export` in ein Tupel in einer separaten `comp`-Funktion erfolgen – da es sich aber um eine sehr kurze Funktion handelt, wurde diese einfach in den Compiler von `Module` eingebettet.

Jede Import-Zeile besteht aus dem Schlüsselwort `import` sowie dem Modulnamen. Je nach Import-Art gibt es dazwischen (`betweenElts`) sowie am Ende (`endElts`) noch weitere Elemente:

```
instance Compilable (Import -> Code) Import Code where
  comp (Import modName impArgs) =
    words $ [text "import"] ++ betweenElts ++ [text modName] ++ endElts
      where (betweenElts, endElts) = case impArgs of
        Simple        -> ([], [])
        Qualified abb -> ([text "qualified"], [text "as", text abb])
        Only onlys    -> ([], [tuple (map text onlys)])
        Hiding hides  -> ([], [text "hiding", tuple (map text hides)])
```

Beim einfachen Import sind beide Listen leer. Qualifizierte Imports beinhalten das Schlüsselwort `qualified` sowie am Ende `as` und die Modul-Abkürzung. Sollen nur bestimmte Symbole importiert oder nicht importiert werden, werden diese in Tupel-Notation angefügt, wobei in letzterem Fall noch das Schlüsselwort `hiding` nötig ist.

Da MagicL bereits einen Standard-Compiler von `Code` nach `String` bereitstellt, kann ein vollständiger Datei-Compiler von der Haskell-DSL nach Haskell durch Verkettung von Parser und Generator sowie Einbettung in einen Aufruf von `compiler` bereitgestellt werden:

```
compiler ((compile :: IOArrow [Sexp] Haskell) >>>
          (compile :: IOArrow Haskell Code))
```

10.5 Zusammenfassung

Mit der Haskell-DSL stellt MagicL eine funktionale Programmiersprache bereit, welche im Wesentlichen eine geklammerte Haskell-Teilmenge darstellt und nach Haskell übersetzt wird. Diese DSL dient einerseits als S-Expression-basierte Metaprogrammiersprache für MagicL-Benutzer, erleichtert andererseits aber auch die Generierung von Haskell-Code, was in Kapitel 11 für die Compiler-DSL genutzt wird. Gegenüber Haskell beinhaltet die DSL nur wenige Umbenennungen und andere Anpassungen wie beispielsweise die Erweiterung aller binärer Operatoren auf beliebig viele Argumente.

Implementiert wurde die Sprache durch ein typisiertes `Haskell`-Modell sowie zwei Compiler zur Verarbeitung: Der erste ist ein Parser von S-Expressions, der zweite ein rein funktionaler Compiler nach `Code`, woraus MagicL letztlich den Quelltext generiert. Abb. 10.1 zeigt den Vorgang als Kategoriendiagramm, wobei die interne Vor- bzw. Weiterverarbeitung auf String-Ebene grau eingezeichnet ist. Der gesamte Haskell-DSL-Compiler ist die Komposition all dieser Compiler und damit ein Arrow von `String` nach `String`.

Sowohl Parser als auch Generator sind mittels `Compilable`-Implementationen für jeden Datentyp des Haskell-Modells konstruiert. Dies hat neben einer hohen Einheitlichkeit den Vorteil, dass beliebige Sub-Compiler mit `comp` angesprochen und durch Haskell anhand der Modelldefinition gefunden werden können. Bei Parsern können auch Listen

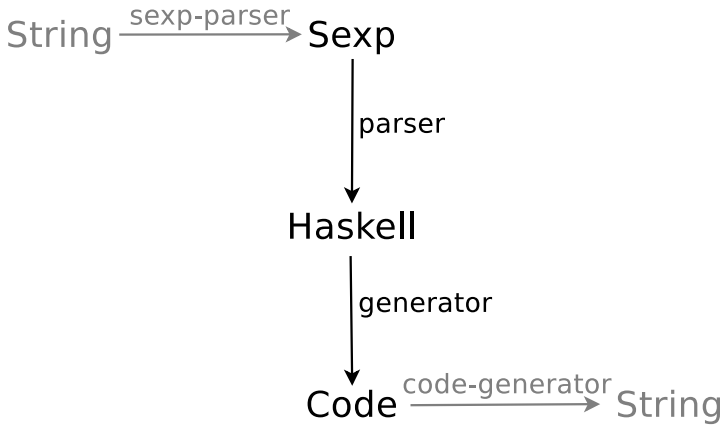


Abbildung 10.1: Kompilationsprozess der Haskell-DSL

und `Maybe`-Datentypen ohne Mehraufwand verarbeitet werden; bei anderen Compilern allerdings nicht, da hier das gewünschte Verhalten nicht für alle Modelle gleich ist. Die Kehrseite dieses Ansatzes sind die umständlichen `Compilable`-Instanzdefinitionen, die vor allem für kleine Compiler nicht sinnvoll erscheinen – eine DSL für Modellverarbeitung könnte dies in Zukunft allerdings vereinfachen.

Allgemein haben statisch typisierte Compiler den Vorteil einer frühzeitigen Typfehlererkennung und erlauben komplexe Algorithmen im Rahmen der Verarbeitung – auf der anderen Seite steht immer mit Modelldefinitionen und der Aufteilung in mehrere Compiler verbundene Aufwand, der insbesondere für kleine Sprachen oder Spracherweiterungen nicht sinnvoll erscheint. Deswegen stellt MagicL alternativ die Möglichkeit der direkten, untypisierten Verarbeitung von S-Expressions bereit, welche im folgenden Kapitel präsentiert wird. Auch die Haskell-DSL hätte auf diese Weise vermutlich einfacher verarbeitet werden können, wurde aber als Verwendungsbeispiel für typisierte Verarbeitung in dieser Form beibehalten.

11 Compiler-DSL

MagicL-Benutzer sollen neue Compiler für weitere DSLs ausschließlich in einer Compiler-DSL[1] schreiben. Diese DSL ist eine echte Erweiterung der Haskell-DSL, daher kann letztere an beliebigen Stellen vorkommen und so als Metaprogrammiersprache verwendet werden.

Zunächst werden die Bestandteile der Compiler-DSL im nächsten Abschnitt spezifiziert. Anschließend wird ein Compiler-Compiler entwickelt, der die Compiler-DSL in die Haskell-DSL übersetzt. Da der Compiler-Compiler sich selbst wiederum am besten in der Compiler-DSL formulieren lässt, erfolgt die Implementation metarekursiv.

11.1 Spezifikation

Die wesentlichen Bestandteile der Compiler-DSL sind das Quasiquote sowie die Möglichkeit, automatisch rekursiv absteigende Makros wie in Lisp zu spezifizieren. Darüber hinaus sind einige Operatoren enthalten, mit denen sich Makros und zusammengesetzte Compiler kürzer als mit den bisherigen Konstrukten definieren lassen.

11.1.1 Quasiquote

Während das Einlesen von S-Expressions dank der in Abschnitt 9.2.2 vorgestellten Funktionen bereits recht komfortabel geht, ist die Erzeugung von Instanzen des `Sexp`-Datentyp bisher noch sehr umständlich. Soll beispielsweise eine Funktion `genSlide title content`, mit den Parametern `"Hello"` und `(list one two)` (als bereits fertig konstruierter S-Expression) aufgerufen, den Ausdruck

```
(slide (title Hello) (list one two))
```

zurückliefern, wäre mit der Haskell-DSL folgende Funktionsdefinition erforderlich:

```
(= (genSlide title content)
   (Node (List (Symbol (Str slide))
               (Node (List (Symbol (Str title)) (Symbol title)))
               content)))
```

Der entsprechende Haskell-Code liest sich aufgrund der syntaktischen Konstrukte `""` und `[]` statt `(Str)` und `(List)` etwas leichter, ist aber davon abgesehen genauso komplex:

[1] Es ist durchaus möglich, wie im Haskell-DSL-Compiler direkt Haskell als Implementationssprache zu verwenden - im Normalfall sollte aber die Compiler-DSL praktischer sein.

```
genSlide title content =
   Node [Symbol "slide",
         Node [Symbol "title" , Symbol title],
         content]
```

Um die Konstruktion von S-Expression-Objekten zu vereinfachen, bieten Lisp-Dialekte die Quasiquote-Syntax mit ‘ und , an (siehe Abschnitt 3.8). In Common Lisp ließe sich diese Funktion daher sehr elegant formulieren:

```
(defun gen-slide (title content)
  `(slide (title ,title) content))
```

Ein derartiger Operator soll nun konstruiert werden. Da MagicL die einfachstmögliche S-Expression-Syntax und damit keine syntaktischen Erweiterungen wie ‘ benutzt, muss die Notation allerdings im Rahmen der gewöhnlichen S-Expression-Syntax erfolgen. Zudem wird nicht zwischen Quote und Quasiquote unterschieden[2] und daher einheitlich der Quote-Operator ’ benutzt. Die Funktion `genSlide` schreibt sich in der Compiler-DSL wie folgt:

```
(= (genSlide title content)
   (’ (slide (title (, title)) ,content)))
```

Splicing einer Liste erfolgt entsprechend mit `(,@)`:

```
(’ (list 1 2 (,@ xs) 5))
```

Dies ergibt `(list 1 2 3 4 5)`, falls `xs` eine Liste der Symbole 3 und 4 ist.

Wie Abschnitt 4.2 erläutert, sollen MagicL-Makros beliebig viele S-Expressions zurückgeben können, die per Splicing an die Stelle des Makro-Aufrufs gesetzt werden. Dies passt auch gut zu einer Definition von Makros als Parser auf S-Expressions, da Parser immer Streams (und damit Listen) als Eingaben haben, so dass Ein- und Ausgabetyp nun übereinstimmen:

```
type LispMacro = SexpParser [Sexp]

-- Typ eines kompilierten Makros
myMac        :: LispMacro
(toIO myMac) :: IOArrow [Sexp] [Sexp]
```

Damit sind Lisp-Makros, nachdem sie mittels `toIO` aus der Typklasse `Executable` umgewandelt wurden, IO-Arrows von `[Sexp]` nach `[Sexp]`, wodurch auch die Hintereinanderschaltung mehrerer Makros reibungslos funktioniert. Um vernünftig in diesen Makros verwendet werden zu können, muss der Quasiquote-Operator immer eine Liste von S-Expressions zurückgeben und so mehrere Rückgaben ermöglichen:

```
(’ 1 2 3)     =>   [Symbol "1", Symbol "2", Symbol "3"]
(’ (1 2 3))   =>   [Node [Symbol "1", Symbol "2", Symbol "3"]]
```

[2] Ein statisches Quote, das nicht auf innere Unquotes reagiert, wird in Lisp nur für verschachtelte Quasiquotes wirklich benötigt. Diese Problematik wird in MagicL allerdings auf anderem Weg behoben, wie in Kürze erläutert wird.

Ein weiteres Problem aller Lisp-Dialekte ist der Umgang mit verschachtelten Quasiquotes, welche typischerweise entstehen, wenn aus Makros wieder Makros generiert werden. Beispielsweise könnten zwei verschachtelte Makros in Common Lisp durch

```
(defmacro x ()
  `(defmacro y (a)
     `(foo ,a)))
```

gegeben sein. Hier ist `a` ein Parameter des inneren Makros und es wird mit `,a` das innere Quasiquote aufgehoben. Soll `a` stattdessen ein Parameter des äußeren Makros sein, muss das äußere Unquote aufgehoben werden:

```
(defmacro x (a)
  `(defmacro y ()
     `(foo ,',a)))
```

Aufgrund der Lisp-Evaluationsregeln muss `,',a` verwendet werden – dies ist auch die einzige Verwendung, für die Common Lisp den (Nicht-Quasi-)Quote-Operator wirklich benötigt. Es gibt noch weitere Fälle dieser Art, und nur die wenigsten Lisp-Programmierer beherrschen all diese Konstrukte vollständig.

MagicL geht hier einen einfacheren Weg: Es werden neben den normalen Quote-Operatoren `' , ,@` anders benannte Operatoren `'1 ,1 ,@1` etc. angeboten, was zu einer eindeutigen Quote-Unquote-Zuordnung führt. Damit wird das erste Beispiel durch

```
(defmacro x ()
  (' (defmacro y (a)
       ('1 (foo (,1 a))))))
```

und das zweite durch

```
(defmacro x (a)
  (' (defmacro y ()
       ('1 (foo (, a))))))
```

ausgedrückt, was deutlich intuitiver ist. Für den Vergleich mit Common Lisp wurde hier ebenfalls `defmacro` verwendet, obwohl es diesen Operator in MagicL nicht gibt[3]. Stattdessen werden die Operatoren `mac` und `automac` angeboten, welche im nächsten Abschnitt erläutert werden.

11.1.2 Makro-Definition

Die Definition von Lisp-Makros (vom Typ `LispMacro = SexpParser [Sexp]`) ist mit den bisherigen Mitteln bereits möglich:

```
(hasType compPredefine LispMacro)
(= compPredefine (macro (Str predefine)
                   (liftA1 gen compSymbol)
                 (where (= (gen name)
                           (' (= (, name) undefined))))))
```

[3]Es ist allerdings vorstellbar, zukünftig zur Vereinfachung wieder ein `defmacro` bereitzustellen, was in den entsprechenden MagicL-Code übersetzt werden kann.

Dieses Beispielmakro ermöglicht es, `(predefine myVar)` statt `(= myVar undefined)` zu verwenden. Das Makro `mac` bietet für die Definition von Lisp-Makros etwas „syntaktischen Zucker“ und beinhaltet bereits die Typdefinition:

```
(mac compPredefine predefine
     (liftA1 ...)
   (where ...))
```

Hier muss man zwischen dem Compiler-Namen `compPredefine` und dem Symbolnamen `predefine` für den Makroaufruf unterscheiden, da ersterer eine Haskell-Variable erzeugt und damit Sonderzeichen wie - ausschließt.

Diese Makros müssen immer explizit aufgerufen werden und unterscheiden sich damit von Lisp-Makros, die in einer globalen Liste verwaltet und automatisch gegen jeden zu expandierenden Ausdruck geprüft und bei Namensgleichheit angewendet wird (siehe Abschnitt 3.8). Beide Ansätze haben Vor- und Nachteile: Der direkte Aufruf bringt eine höhere Flexibilität und ermöglicht die in Abschnitt 4.2 geforderderten lokalen Makros. Lisp-Makros sparen dafür in Fällen, wo globale Makros unproblematisch sind, unnötige Schreibarbeit. Aus diesem Grund stellt MagicL zusätzlich den Operator `autoMac` bereit, mit dem (für einen Compiler) globale und automatisch expandierte Makros definiert werden können:

```
(autoMac compSomething something
    (compileSomething)
  (where ...))
```

`autoMac` erwartet an erster Stelle weiterhin einen Haskell-Namen, obwohl dieser im Normalfall aufgrund der automatischen Anwendung nicht benötigt wird. Es gibt allerdings besondere Situationen, in denen ein direkter Aufruf dennoch erforderlich ist. Alle in einem Compiler definierten Auto-Makros werden zu einen rekursiv absteigenden Compiler namens `compAuto` kombiniert, der wieder direkt aufgerufen werden kann.

Für die einfachere Moduldefinition von Compilern steht der Operator `compiler` bereit, welches alle benötigten MagicL-Imports automatisch einbindet. Beispielsweise wird

```
(compiler Comp2Haskell (imports Data.Maybe Data.List)
```

in folgenden Haskell-DSL-Code übersetzt:

```
(module
  Comp2Haskell
  (Prelude (hiding id lines words take))
  Data.IORef
  (Control.Category (only id))
  Control.Arrow
  System.IO.Unsafe
  Util
  Arrows
  Sexp
  Parser
  Model
```

```
  Data.Maybe
  Data.List)
```

11.2 Compiler-Compiler

Der Compiler für die Compiler-DSL sollte selbst möglichst einfach formuliert werden. Da S-Expressions verarbeitet werden, ist eine Implementation als `SexpParser` sinnvoll. Darüber hinaus werden aber auch S-Expressions erzeugt, so dass hier auch Quasiquote-Notation und Lisp-Makros bereits nützlich wären. Es wird sich daher um einen metarekursiven Compiler handeln, d.h. einen Compiler, der sich selbst übersetzt. Um hier das Henne-Ei-Problem zu lösen, wird wie im Compilerbau üblich später ein Bootstrapping[4] erforderlich sein. Metarekursive Programme sind in der Lisp-Welt nicht unüblich – bereits der erste LISP[5]-Interpreter wurde als `eval`-Funktion in LISP selbst implementiert und dann manuell in Maschinensprache übersetzt [31]. Ein anderes prominentes Beispiel ist das Common Lisp Object System (CLOS) [28], welches selbst objektorientiert mit CLOS implementiert ist und über Makros in ein „objektfreies Common Lisp“ übersetzt werden kann.

Gehen wir also zunächst davon aus, solch ein Compiler wäre bereits vorhanden und könnte für die Implementation genutzt werden. Die Verarbeitung vom Quasiquote lässt sich dann durch ein komplexes Auto-Makro realisieren:

```
(autoMac compQuote ' inners
  (where
    ...
```

`inners` muss beliebig viele S-Expressions verarbeiten und daraus einen Haskell-Ausdruck erzeugen, der eine Liste von S-Expressions repräsentiert:

```
    (hasType inners LispMacro)
    (= inners
       (>>^ (many (<+> unquoteAll
                       (>>^ inner quoteList)))
            (>>> concat quoteAppend)))
```

`inners` erwartet jeweils entweder ein Unquote-All oder einen einzelnen Ausdruck, welcher mit `inner` verarbeitet wird. Alle normalen Ausdrücke werden mit `quoteList` in eine Liste eingebettet und anschließend durch `quoteAppend` mit `++` verknüpft, um die Semantik vom Unquote-All zu erreichen. Beispielsweise wird `inners` die Ausdrücke

```
1 2 (,@ xs) 6
```

wie folgt in die Haskell-DSL übersetzen:

```
(++ (List (Symbol (Str 1)))
    (List (Symbol (Str 2)))
```

[4] Der Begriff Bootstrapping ist eine Anspielung auf Baron Münchhausen, der sich an seinen Schnürsenkeln aus dem Sumpf gezogen haben will.

[5] LISP ist der erste Dialekt der Sprachfamilie Lisp.

```
xs
(List (Symbol (Str 6))))
```

`inner` verarbeitet einen einzelnen Ausdruck und gibt als Ergebnis Haskell-DSL-Code für einen S-Expression zurück:

```
(hasType inner LispMacro)
(= inner (<+> unquote
              (>>^ takeSymbol quoteSymbol)
              (>>^ (compNode inners) quoteNode)))
```

`inner` erwartet entweder ein Unquote, ein Symbol oder einen Knoten. Symbole werden durch die Funktion `quoteSymbol` quotiert, Knoten rekursiv mit `inners` verarbeitet und anschließend als Knoten quotiert. `unquote` und `unquoteAll` sind bis auf die verschiedenen Matcher-Symbole identisch und verarbeiten einen einzigen S-Expression, den sie mit der Funktion `single x = [x]` in eine Liste einbetten um dem Rückgabetyp `[Sexp]` für Lisp-Macros zu entsprechen:

```
(mac unquote    ,  (>>^ take single))
(mac unquoteAll ,@ (>>^ take single))
```

Alle vier Funktionen zur Quotierung von Haskell-DSL-Ausdrücken können selbst mit einem Quote implementiert werden:

```
(= (quoteSymbol sym)
   (' (Symbol (Str (, (Symbol sym))))))
(= (quoteNode sexps)
   (' (Node (,@ sexps))))
(= (quoteList sexps)
   (' (List (,@ sexps))))
(= (quoteAppend sexps)
   (' (++ (,@ sexps)))))
```

`quoteSymbol` erzeugt die Repräsentation eines Symbols, indem es Aufrufe von `Symbol` und `Str` auf Haskell-DSL-Ebene ineinander verschachtelt. Auf der Ebene der Metasprache wird ein weiterer Aufruf von `Symbol` benötigt, um den String `sym` in einen S-Expression umzuwandeln. Generell muss jedes Argument eines Unquote vom Typ `Sexp` und jedes Argument eines Unquote-All vom Typ `[Sexp]` sein. Die anderen drei Quote-Funktionen erzeugen jeweils einen benannten Knoten, in den sie alle als Liste übergebenen Ausdrücke einfügen, und unterscheiden sich nur im erzeugten Knotennamen.

Damit ist die Implementation des Quasiquote-Operators abgeschlossen. Die Definitionen der benannten Varianten wie `'1` sind bis auf die benutzten Makro-Matcher `'1 ,1 ,@1` etc. statt `' , ,@` identisch. Um Code-Duplikation zu vermeiden, benutzt MagicL hier ebenfalls ein Makro, welches alle Operatoren `'` bis `'3` aus einem einzigen Template definiert.

Der Operator `mac` lässt sich einfach implementieren:

```
(autoMac compMac mac
  (liftA4 gen take take take (many take))
  (where (= (gen fun sym cmd cmds)
```

```
          (' (hasType (, fun) LispMacro)
             (= (, fun) (macro (Str (, sym)) (, cmd)) (,@ cmds))))))
```

Damit wird beispielsweise

```
(mac compSomething macroName
    (doSomething ...)
  (where ..))
```

nach

```
(hasType compSomething LispMacro)
(= compSomething (macro (Str macroName)
                    (do-something ...))
  (where ...))
```

übersetzt. Die Implementation von `autoMac` ist schwieriger. Zunächst muss dafür die Makroexpansion-Funktionsweise von Common Lisp nachgebildet werden, welche versucht, eine (in Lisp globale) Liste von Makros verodert auf den Eingabe-Stream anzuwenden. Ist die Anwendung eines der Makros erfolgreich, wird dessen Rückgabe erneut als Eingabe der Makro-Liste verarbeitet. Kann keines der Makros auf einen Knoten angewendet werden, wird versucht, dessen Kinder zu expandieren. Symbole werden in diesem Fall unverändert zurückgegeben. Dieses Verhalten wurde in MagicL in der Haskell-Funktion `lispTraverse`[6] implementiert:

```
lispTraverse :: [LispMacro] -> LispMacro
lispTraverse macs = combinedMacs
  where combinedMacs = many (foldr (<+>) defaultMac recMacs) >>^ concat
        recMacs      = map (\ m -> applyParser m combinedMacs) macs
        defaultMac   = ((compSymbol >>^ single) <+>
                        (compNode combinedMacs >>^ (Node >>> single)))
```

Die Eingabeliste `macs :: [LispMacro]` wird in das Ausgabemakro `combinedMacs` umgewandelt, indem alle Makros aus `recMacs` und zuletzt `defaultMac` verodert werden. `recMacs` entsteht aus `macs`, indem jedes darin enthaltene Makro `m` mittels `applyParser` so modifiziert wird, dass seine Ausgabe im Erfolgsfall wieder als Eingabe für `combinedMacs` benutzt wird. `defaultMac` lässt Symbole unverändert und steigt rekursiv in Knoten hinab. Alle Aufrufe von `concat` und `single` in `lispTraverse` werden benötigt, um den Ausgabetyp `[Sexp]` zu bekommen.

Auto-Makros können nun mit Hilfe von `lispTraverse` realisiert werden. Dafür müssen zunächst alle in einem Compiler verwendeten `autoMac`-Deklarationen gefunden und deren Haskell-Name gemerkt werden. Anschließend wird ein Makro mit dem Haskell-Namen `compAuto` erzeugt, welches alle gefundenen Namen an `lispTraverse` übergibt. Da das Scannen von Auto-Makros bereits vor der normalen Makroexpansion von `autoMac` geschieht, kann `autoMac` selbst sich genau wie `mac` verhalten.

[6] `lispTraverse` ist eigentlich Teil der S-Expression-Bibliothek, wird allerdings erst hier vorgestellt da dies thematisch besser passt.

Auf die konkrete Implementation des Toplevel-Compiler-Compilers wird an dieser Stelle nicht weiter eingegangen – sie kann dem MagicL-Code für das Modul `Comp2Haskell` entnommen werden.

11.3 Beispiel: Ein Slide-Compiler

Als Beispiel wird ein primitiver, aber vollständiger Compiler angegeben, der eine Slide-DSL in eine Latex-DSL übersetzt. Beispielsweise soll der Slide-Code

```
(presentation (text My Presentation)
  (slide First
     (list one two three))
  (slide (text Second Slide)
     (list foo
           bar
           (text Hello World))))
```

in den Latex-DSL-Code

```
(cmd documentclass beamer)
(cmd title (text My Presentation))
(env document
  (cmd maketitle)
  (env (frame First)
    (env itemize
       (cmd item) one
       (cmd item) two
       (cmd item) three))
  (env (frame (text Second Slide))
    (env itemize
       (cmd item) foo
       (cmd item) bar
       (cmd item) (text Hello World))))
```

übersetzt werden, woraus ein fiktiver Latex-DSL-Compiler den Latex-Ausgabecode

```
\documentclass{beamer}
\title{My Presentation}
\begin{document}
  \maketitle
  \begin{frame}{First}
    \begin{itemize}
    \item one
    \item two
    \item three
    \end{itemize}
  \end{frame}
  \begin{frame}{Second Slide}
```

```
    \begin{itemize}
    \item foo
    \item bar
    \item Hello World
    \end{itemize}
  \end{frame}
\end{document}
```

generieren könnte, aus dem schließlich die in Abb. 11.1 gezeigte Präsentation erzeugt wird. Der Slide-Compiler kann mit MagicL wie folgt implementiert werden:

Abbildung 11.1: Die fertige Beispielpräsentation

```
(compiler SlideCompiler)

(mac compPres presentation
  (liftA2 genPres take (>>^ (many slide) concat)
  (where (= (genPres title slides)
            (' (cmd documentclass beamer)
               (cmd title (, title))
               (env document
                 (cmd maketitle)
                 (,@ slides)))))))

(mac compSlide slide
  (liftA2 genSlide take compAuto)
  (where (= (genSlide title contents)
            (' (env (frame (, title))
                 (,@ contents))))))

(autoMac compList list
  (>>^ compAuto genList)
  (where
    (= (genList items)
       (' (env itemize
            (,@ (concat (map genItem Items))))))
    (= (genItem text)
       (' (cmd item)
```

```
          (, text)))))

(= slideCompiler compPres)
```

Hierbei werden sowohl normale als auch Auto-Makros verwendet. An äußerster Stelle darf nur ein `presentation`-Knoten vorkommen, der daher als Makro realisiert wurde und einen Präsentationstitel sowie beliebig viele `slide`-Knoten enthält. Dieses Makro erzeugt eine Dokumentklassendeklaration, setzt den Titel und bettet die Folien in eine `document`-Umgebung ein wie in Latex erforderlich. Da Folien nur direkt im `presentation`-Knoten vorkommen dürfen, werden diese ebenfalls mit einem gewöhnlichen Makro verarbeitet, welches einen Titel und beliebig viel Inhalt erwartet und daraus eine `frame`-Umgebung erzeugt. Alle Folieninhalte werden mit Auto-Makros kompiliert, da hier normalerweise beliebige Verschachtelungen zugelassen werden sollen. Zudem können so später weitere Makros wie z.B. `image` hinzugefügt werden, ohne das `slide`-Makro anpassen zu müssen. Diese primitive Version die Slide-DSL enthält mit `list` nur einen einzigen Befehl für Folieninhalte, welcher in eine entsprechende `itemize`-Umgebung übersetzt wird. Allerdings kann im Folieninneren bereits beliebiger Latex-DSL-Code integriert werden, so dass auch hiermit bereits sinnvolle Präsentationen möglich sind. Die Definition von `slideCompiler` als `compPres` dient lediglich der Klarheit beim Aufruf aus externen Modulen.

11.4 Zusammenfassung

Für benutzerdefinierte Compiler bietet MagicL die Compiler-DSL an, die eine Erweiterung der Haskell-DSL ist, so dass eine Haskell-Variante in S-Expression-Syntax als Metaprogrammiersprache verwendet werden kann. Darüber hinaus beinhaltet die Compiler-DSL für die Verarbeitung von S-Expressions einen (Quasi-)Quote-Operator `'` sowie ein Lisp-ähnliches Makrosystem. Der Quote-Operator unterscheidet sich in einer Reihe von Punkten von bestehenden Lisps: Zunächst wird nicht zwischen Quote und Quasiquote unterschieden, da der einzige Fall, in dem beispielsweise Common Lisp ein gewöhnliches Quote benötigt, die komplizierte Auflösung von Unquotes in verschachtelten Quasiquotes ist. Dieses Problem wird in MagicL jedoch einfach durch die Existenz verschieden benannter Quote/Unquote/Unquote-All Tripel wie `'1 ,1 ,@1` gelöst. Ein weiterer Unterschied ist, dass Makros und Quotes nicht einen einzelnen Ausdruck, sondern immer eine Liste von S-Expressions zurückliefern, die vom Makrosystem mittels Splicing in die Ausgabe eingefügt wird. Auf diese Weise können Makros auch mehrere oder gar keine Ausdrücke zurückliefern, was die Ausdrucksmächtigkeit erhöht. Während normale Makros in MagicL gezielt über ihren Haskell-Namen aufgerufen werden, erlauben Auto-Makros die für Lisp typische automatische Auswertung an beliebiger Tiefe sowie die wiederholte Makroexpansion von Rückgabeausdrücken anderer Makros.

12 Diskussion

Dieses Kapitel diskutiert MagicL in Hinblick auf die Anforderungen aus Kapitel 4 und seine Vor- und Nachteile gegenüber bestehenden Werkzeugen. Gegliedert ist die Diskussion nach den drei unterstützten Modelltypen, deren Verwendung jeweils erörtert wird. Anschließend wird auf die allgemeine Schnittstelle für Compiler eingegangen.

12.1 Zeichenketten

Zeichenketten können mit Hilfe der Parser-Bibliothek eingelesen werden. Parserdefinitionen sind syntaktisch etwas aufwändiger als eine EBNF-Grammatik. Beispielsweise würde die EBNF-Definition

```
Zahl = [ '-' ] ZifferAusserNull { Ziffer }  | '0'
```

durch den Parser

```
(= zahl (<+>
          (>>> (optional (eq '-'))
               zifferAusserNull
               (many ziffer))
          (eq '0')))
```

abgebildet werden. Andererseits sind Parser deutlich mächtiger als Grammatiken, da hier nicht nur zwischen akzeptiert und nicht akzeptiert unterschieden wird, sondern gleichzeitig eine Verarbeitung zu einer Ausgabe erfolgt. Zudem ermöglichen Parser die Definition eigener Kombinatoren wie `commaSep` für Komma-separierte Listen, welche komplexe Definitionen deutlich vereinfachen können.

Ähnlich verhält es sich mit der Erzeugung von Zeichenketten, die statt durch Templates rein funktional geschieht. Auf der einen Seite ist dies bei einfachen Beispielen umständlicher und auch weniger lesbarer als beispielsweise die Verwendung von StringTemplate (siehe Abschnitt 3.3). Ein Generator, der aus einem Namen und einer Liste von Methoden eine Java-Klasse erzeugt, kann mit StringTemplate durch

```
genClass = new StringTemplate("public class $name$ {
  $methods; separator=\"\n\n\"$
}");
```

in einem Java-Programm beschrieben werden. Mit MagicL müsste derselbe Generator als

```
(= (genClass name methods)
   (words (List (text (Str public class))
```

```
                name
                (braces (paragraphs methods)))))
```

spezifiziert werden, was deutlich aufwändiger wirkt. Auf der anderen Seite sind Generatoren in MagicL gewöhnliche Haskell-Funktionen, wodurch der Aufruf mit

```
layout 80 (genClass className classMethods)
```

einfacher als in StringTemplate, wo die Befehle

```
genClass.setAttribute("name", className);
genClass.setAttribute("methods", classMethods);
genClass.toString();
```

benötigt werden. So können MagicL-Generatoren einfacher in viele kleine, wiederverwendbare Komponenten aufgeteilt werden und entsprechend dem funktionalen Paradigma mit Funktionen höherer Ordnung kombiniert werden. Eine genauere Bewertung kann erst vorgenommen werden, wenn beide Systeme für die Definition von komplexen Code-Generatoren vewendet wurden. Auf jeden Fall aber wäre es möglich, Code-Generierung in MagicL durch eine Code-DSL noch weiter zu vereinfachen.

Die Zeichenketten-Ebene muss vom Metaprogrammierer allerdings nur für die Unterstützung neuer externer Ein- und Ausgabeformate betreten werden. Im Idealfall[1] können Compiler komplett auf der einfachen S-Expression-Ebene spezifiziert werden.

12.2 S-Expressions

S-Expressions werden ebenfalls durch Parser verarbeitet, welche allgemeiner und mächtiger sind als gewöhnliche Lisp-Makros, letztere aber dennoch als Spezialfall einschließen. Dies ermöglicht beispielsweise lokale Makros oder Übersetzer, die nicht auf genau einen Knotennamen anspringen, sondern beispielsweise alle binären Operatoren einer Sprache. Die größere Allgemeinheit und Kontrolle geht allerdings derzeit mit etwas erhöhtem Aufwand für die einfachen, in bestehenden Lisps unterstützen Fälle einher. Beispielsweise könnte ein Makro, welches eine Foliendefinition der Slide-DSL in die Latex-DSL übersetzt, die Eingabe

```
(slide MySlideTitle
  (image image.png)
  (list one two three))
```

in die Ausgabe

```
(env (frame MySlideTitle)
  (image image.png)
  (list one two three))
```

[1] Idealfall meint hier, dass ausschließlich S-Expression-DSLs verwendet werden, alle Backends für zu generierende Zielsprachen bereits vorhanden sind und die Verarbeitungsalgorithmen nicht so komplex sind, dass getypte Objekte benötigt werden.

überführen[2]. Ein derartiges Makro wäre in Common Lisp durch

```
(defmacro slide (title &rest contents)
  `(env (frame ,title)
     ,@contents))
```

gegeben. Hier wurden die Variablen `title` und `contents` von Lisp an den ersten bzw. alle weiteren Parameter gebunden. Mit der Compiler-DSL hingegen würde das Makro als

```
(mac compSlide slide
  (liftA2 genSlide take (many take))
  (where (= (genSlide title contents)
            (' (env (frame (, title))
                 (,@ contents))))))
```

definiert werden, wo explizite Aufrufe von `take` bzw. `many take` benötigt werden und per `liftA2` an eine Funktion übergeben werden müssen, um denselben Effekt zu erzielen. Es wäre daher durchaus wünschenswert die Compiler-DSL noch weiter zu vereinfachen, allerdings ist nicht klar wie genau dies aussehen soll. Es könnte beispielsweise ein komplett mit Common Lisp übereinstimmendes `defmacro` angeboten werden, welches kurze Definitionen ermöglicht, im Gegenzug aber an Allgemeinheit verliert. Welche Sprache wirklich den besten Kompromiss zwischen Allgemeinheit und Programmkürze ermöglicht bleibt daher eine offene Frage, die sich vermutlich erst nach häufiger Verwendung näher beantworten lassen wird.

Generierung von S-Expressions erfolgt mit dem Quasiquote-Operator, der im Gegensatz zu Lisp keine eigene Syntax benutzt, sondern an gewöhnliche S-Expression-Notation angepasst wurde. Dieser Operator enthält zwei Verbesserungen gegenüber Lisp. Zum einen ist es damit möglich, aus einem Makro mehrere Ausdrücke zu erzeugen, die alle per Splicing an der ursprünglichen Stelle eingefügt werden. Beispielsweise wäre es praktisch, in der Slide-DSL mehrere gleichbenannte, automatisch nummerierte Folien mit dem Befehl `multi-slide` definieren zu können, so dass

```
(multi-slide Title
  (s (items one two three))
  (s (image diagram.png)
     (text This is a diagram)))
```

übersetzt würde nach

```
(slide (text Title 1)
  (items one two three))

(slide (text Title 2)
  (image diagram.png)
  (text This is a diagram))
```

[2]Im Gegensatz zum Beispiel in Abschnitt 11.3 wird hier angenommen, dass die Latex-DSL bereits einen `list`-Befehl enthält, so dass dieser nicht übersetzt werden muss.

Ein derartiges Makro ist in MagicL möglich, in Lisp hingegen nicht. Dort müsste die Slide-DSL wie Lisp selbst einen `progn`-Befehl enthalten, der es ermöglicht, mehrerere Ausdrücke zu einem zusammenzufassen, so dass die Ausgabe nun von der Form

```
(progn
  (slide ...)
  (slide ...))
```

wäre. Manchmal ist es auch erforderlich, aus einem Makro überhaupt keine Ausgabe zu erzeugen. Beispielsweise könnte eine DSL Kommentare der Form `(comment ...)` an beliebigen Stellen zulassen, die vom Compiler ignoriert werden sollen. Dies kann in MagicL durch ein simples Makro mit einem leeren Quasiquote erreicht werden:

```
(autoMac compComment comment
  (liftA1 genComment (many take))
  (where (= (genComment comments)
            (’))))
```

In Lisp ist derartiges selbst mit `progn` nicht möglich, da ein leeres `progn` gleichbedeutend mit `nil` ist, welches von Lisp an vielen, aber nicht allen Stellen ignoriert wird, so dass auf diese Weise z.B. keine Kommentare in Parameterlisten vorkommen dürften.

Der andere Vorteil des MagicL-Quasiquotes zeigt sich, falls mittels Quasiquote Code generiert werden soll, der selbst Quasiquotes enthält. Dies wäre beispielsweise dann der Fall, wenn Benutzer selbst Erweiterungen für die Compiler-DSL implementieren wollen. In Common Lisp sind verschachtelte Quasiquotes teilweise sehr schwierig zu verstehen (siehe Abschnitt 11.1.2). MagicL löst dieses Problem hingegen einfach durch die Bereitstellung verschieden benannter Quote/Unquote-Paare.

12.3 Objekte

Mit der in die Compiler-DSL integrierte Haskell-DSL bietet MagicL eine statisch typisierte Programmiersprache an, mit der komplexere Algorithmen elegant im funktionalen Paradigma realisiert werden können.

Programme in der Haskell-DSL sind aufgrund der Umsetzung auf reine S-Expressions ohne weitere Syntax etwas länger als Haskell-Programme selbst, so dass beispielsweise der Haskell-Ausdruck `["a" "b" "c"]` in der DSL als `(List (Str a) (Str b) (Str c))` geschrieben wird. Dies ist der Nachteil, der mit der leichteren Generierbarkeit von Code sowie der reibungslosen Integrierbarkeit mit anderen DSLs einhergeht. Es wäre durchaus möglich, die S-Expression-Syntax um Notationen für Strings und Arrays erweitern. Damit die Modellkomplexität dabei nicht zunimmt, könnte diese Syntax von einem Präprozessor wieder auf S-Expressions abgebildet werden, so dass beispielsweise `[a b]` denselben S-Expression ergibt wie `(List a b)` – einen derartigen Präprozessor können MagicL-Nutzer auch selbst implementieren. Es ist allerdings nicht einfach zu entscheiden, welche zusätzliche Syntax für alle DSLs gleichzeitig optimal ist. DSL-spezifische Zusatzsyntax hingegen würde die Integration verschiedener DSLs erschweren und gewissermaßen die

Grundidee von MagicL ad absurdum führen. Ein alternativer Ansatz wäre die Beibehaltung der puren S-Expressions und die Bereitstellung einer Entwicklungsumgebung, die DSL-spezifische Anzeige- und Bearbeitungsmodi anbietet, die bestimmte Ausdrücke benutzerfreundlich visualisieren können. Auf diesen Gedanken wird in Abschnitt 13.2.3 weiter eingegangen.

12.4 Compiler-Schnittstelle

MagicL's modellspezifische Hilfsmittel sind frei miteinander kombinierbar. Beispielsweise können Parser auf S-Expressions oder Strings beliebige Haskell-Objekte zurückliefern, oder Haskell-Funktionen S-Expressions mittels Quasiquote und Strings mittels Generator-Bibliothek erzeugen. Alle mit MagicL definierbaren Compilertypen lassen sich hintereinanderschalten, indem sie zunächst mit `toIO` in Arrows der Kategorie von Funktionen mit Nebeneffekten umgewandelt und anschließend mittels `>>>` verknüpft werden. Dies führt gemeinsam mit dem Ansatz vieler, teilweise ineinander übersetzbarer S-Expression-basierter DSLs zu einer starken Komponententrennung, bei der Compiler und DSLs häufig wiederverwendet werden können.

Ein experimentelles Feature von MagicL's generischer Compilerschnittstelle ist die Möglichkeit, spezielle Compiler fest als Standardcompiler zwischen zwei Typen zu verdrahten. Damit ist es unter anderem möglich, Parser für ein zusammengesetztes Modell mit vielen Untertypen auf eine Weise zu konstruieren, bei der Aufrufe von Sub-Parsern nicht explizit, sondern implizit durch Typinferenz erfolgen, was das Schreiben derartiger Parser erleichtern kann. Außerdem können automatisch Standardparser für Listen und optionale Werte aus normalen Parsern abgeleitet werden, wobei die abgeleiteten Standardparser durch die Parserkombinatoren `many` und `optional` aus den ursprünglichen entstehen. Die Notation für derartige Standardcompiler ist allerdings noch etwas umständlich, wie die Umsetzung des Haskell-DSL-Compilers in Kapitel 10 zeigt – hier könnte eine Erweiterung der Compiler-DSL helfen. Ein größeres Problem ist, dass die – recht komplizierte – Realisierung dieses Features mit der Typklasse `Compilable` bereits an die Grenzen von Haskells Typsystem stößt, so dass bereits nicht-standardisierte Erweiterungen des Haskell-Compilers GHC [3] benötigt werden. Auch ist es aufgrund von Einschänkungen im Typsystem nicht möglich, automatisch erzeugte Listen-Standardcompiler mit eigenen Listen-Compilern zu überschreiben. Eine Nachfolgeimplementierung von MagicL sollte daher entweder auf ein derartiges Feature verzichten oder statt Haskell eine dynamisch typisierte Programmiersprache verwenden.

12.5 Zusammenfassung

MagicL erfüllt alle in Kapitel 4 gestellten Anforderungen, wie Abb. 12.1 zeigt. Als einziger Kritikpunkt bleibt die Länge von Metaprogrammen, die noch nicht ganz mit Lisp konkurrieren kann. Daher könnte die Compiler-DSL künftig um Konstrukte erweitert werden, die bisher aufwändige Aufgaben vereinfachen. Da aber MagicL deutlich allge-

meiner ist als Lisp-Makrosysteme, können nicht einfach die dort bestehenden Mechanismen eins zu eins übernommen werden - vielmehr kann erst die Erfahrung mit mehr Anwendungsbeispielen zeigen, welche genauen Ergänzungen sinnvoll sind. Diese Erweiterungen können MagicL-Nutzer auch selbst implementieren, denn wie jedes Lisp ist MagicL selbst durch Makros erweiterbar. Hierfür müsste lediglich ein Compiler definiert werden, der die erweiterte Compiler-DSL in die bestehende übersetzt.

Werkzeug	**MagicL**	**erfüllt?**
Eingabemodell	S-Expression / beliebig	ja
Eingabevalidierung	beliebig	ja
Ausgabemodell	S-Expression / beliebig	ja
Ausgabevalidierung	beliebig	ja
Zielsprache	beliebig	ja
vollständiges Quasiquote	ja	ja
Code-Objekt verfügbar	ja	ja
Sprachmächtigkeit	maximal	ja
Metaprogrammlänge	kurz	ja, aber noch nicht optimal
Template-Aufruf	verallgemeinertes Makro	ja
Komponententrennung	unterstützt	ja
Template-Generierung	einfach durch `'1` etc.	ja

Abbildung 12.1: Erfüllung der Anforderungen an MagicL

13 Schluss

Zum Abschluss werden die wichtigsten Aspekte dieser Arbeit noch einmal zusammengefasst. Anschließend werden im Ausblick weitere Ideen und Ansatzpunkte für Fortsetzungsarbeiten vorgestellt.

13.1 Zusammenfassung der Arbeit

Ziel dieser Diplomarbeit ist die prototypische Entwicklung eines universellen Frameworks zur Code-Generierung, welches möglichst viele Vorteile bestehender Technologien im Rahmen eines klaren, einheitlichen Formalismus in sich vereint. Insbesondere die mächtigen und praktischen Metaprogrammierungskonzepte der Lisp-Sprachfamilie haben diese Arbeit inspiriert und sollten in einer verallgemeinerten Form aufgegriffen werden.

Zunächst wurden hierfür in Kapitel 2 die Modelltypisierungsgrade String, Objekt und Baumstruktur für die Repräsentation von Quelltext behandelt mit dem Ergebnis, dass alle drei für spezielle Aufgaben nützlich sind und daher gemeinsam verwendbar sein sollten: Während Strings am Anfang und Ende der Verarbeitungskette benötigt werden, ansonsten aber eine häufig zu niedrige, fehleranfällige Arbeitsebene bedeuten, sind typisierte Objekte aufgrund ihrer Sicherheit und Performanz für komplexere Verarbeitungsschritte geeignet, bringen allerdings den Overhead der Deklaration sowie eine Programmiersprachenabhängigkeit mit sich. Baumstrukturen bieten oft einen Kompromiss aus Flexibilität und Strukturierung und sind daher für einfache Verarbeitungsschritte geeignet, bei denen ein typisiertes Modell übertrieben erscheint. Als konkrete Baumrepräsentation wurden S-Expressions aufgrund ihrer Einfachheit und Redundanzfreiheit insbesondere gegenüber XML vorgezogen.

In Kapitel 3 wurde eine Reihe von Code-Generierungswerkzeugen vorgestellt und verglichen. Gemein ist all diesen Werkzeugen die Benutzung des Template-Konzepts, ansonsten sind die strukturellen Unterschiede allerdings groß. Manche Frameworks können Code in beliebigen Sprachen generieren, andere beschränken sich auf eine einzelne Sprache oder Sprachfamilie. Alle drei Modelltypen aus Kapitel 2 werden in verschiedenen Systemen verwendet, allerdings gibt es keines, das diese gleichzeitig anbietet. Die Bandbreite der einzelnen Metasprachen reicht von kaum mehr als dem bloßen Template-Formalismus aus Quasiquote und Unquote bis hin zu vollständigen Programmiersprachen. Werkzeuge, in denen Meta- und Zielsprache übereinstimmen können, erlauben prinzipiell auch die Generierung von Meta-Code selbst, woraus sich interessante Möglichkeiten ergeben. Hier kann es allerdings zu einer Mehrdeutigkeit bei der Auswertung von Unquotes kommen, die meist nur umständlich umgangen werden kann. Das Aufrufen von Templates

verläuft meist explizit wie bei einem Funktionsaufruf. Alternativ treten auch Makros auf, die implizit über den Namen des zu verarbeitenden Ausdrucks zugeordnet werden. XSLT unterstützt hier sogar ein Pfad-basiertes Matching.

Von den untersuchten Werkzeugen konnten Lisp-Makros aufgrund ihrer Flexibilität, Mächtigkeit, Modularisierbarkeit und Einfachheit am stärksten überzeugen. Lisp-Dialekte stellen jedoch noch kein universelles Code-Generierungswerkzeug dar, da diese ausschließlich Lisp-Code generieren können. Die Übertragung auf beliebige Zielsprachen ist daher die wichtigste der in Kapitel 4 besprochenen Anforderungen. Auch externe Eingabeformate sollten durch die Konstruierbarkeit spezifischer Parser unterstützt werden können. Darüber hinaus wurden gegenüber Lisp weitere Verallgemeinerungen gefordert: So sollten neben S-Expressions auch Strings und Objekte als Modelle verwendet werden können, obwohl der Fokus auf S-Expressions bleibt. Makros sollten sowohl explizit aufrufbar sein als auch über ein Pattern-Matching gegen den zu verarbeitenden Ausdruck, welches über einen Vergleich mit dem Knotennamen hinausgeht. Auch „lokale Makros“, welche nur im Kontext anderer Makros ausgewertet werden, sollten realisierbar sein, um auf ein komplexes „Code-Walking“ – wie in Lisp nötig – möglichst verzichten zu können.

Eine weitere Forderung war die Möglichkeit, Verarbeitungseinheiten wie Makros, Parser und andere Compiler nach dem Baukastenprinzip auf verschiedene Weisen und in einer Semantik kombinieren zu können. Das theoretische Fundament dieser Semantik liefert die Kategorientheorie, deren Grundzüge in Kapitel 5 vorgestellt wurden. Kategorien bestehen aus Morphismen zwischen Objekten – wobei beide Begriffe axiomatisch definiert und somit zunächst bedeutungsneutral sind. Die häufigste Interpretation für Morphismen und Objekte in der Mathematik sind Funktionen und Mengen, im Rahmen dieser Arbeit kommen jedoch ausschließlich Verarbeitungsprozesse und Typen vor. Kategorien können anschaulich als Diagramme visualisiert werden. Innerhalb einer Kategorie können neben der Komposition Produkte und Coprodukte verwendet werden, um einfache Einheiten zu komplexeren zusammenzusetzen. Auch ganze Kategorien können über Funktoren aufeinander abgebildet werden. Monaden sind spezielle Funktoren auf einer Kategorie, die eine spezielle, durch natürliche Transformationen beschriebene Struktur besitzen.

Die funktionale Sprache Haskell, welche in Kapitel 6 präsentiert wurde, bietet in ihren Bibliotheken mit den Typklassen `Arrow` und `Monad` programmiersprachliche, auf Verarbeitungsprozesse eingeschränkte Umsetzungen von Morphismen und Monaden. Dies ist einer der Gründe, weshalb MagicL in Haskell implementiert wurde. Monaden und Arrows sind aufgrund der Kleisli-Kategorie generell gegeneinander austauschbar, wobei Arrows allerdings allgemeiner sind. Haskell selbst verwendet häufig Monaden, mit denen die Verarbeitung bestimmter parametrisierter Typen einheitlich beschrieben werden kann. Beispiele hierfür sind Listen sowie Variablen, die Null-Pointer enthalten dürfen sowie das Weiterreichen von Zuständen. Die `IO`-Monade lässt sich als versteckter, hypothetischer Welt-Zustand verstehen und wird in Haskell verwendet, um Operationen mit Seiteneffekten in die ansonsten pur funktionale Sprache integrieren zu können.

Die grundlegende Architektur von MagicL wurde in Kapitel 7 erläutert. Alle Compiler zwischen den drei unterstützten Modelltypen sind Arrows oder lassen sich in Arrows konvertieren, und können daher mit dem Kompositionsoperator `>>>` hintereinandergeschal-

tet werden, so dass Compiler oft in mehrere wiederverwertbare Komponenten aufgeteilt werden können. Benutzerdefinierte Compiler werden in einer Compiler-DSL geschrieben, die mit der Haskell-DSL eine komplette funktionale Programmiersprache enthält. Für die Verarbeitung von Zeichenketten stehen eine Parser- sowie eine Generator-Bibliothek bereit. S-Expressions werden ebenfalls durch Parser verarbeitet, was MagicL's Antwort auf die Forderung verallgemeinerter Makros ist. Die Erzeugung von S-Expressions erfolgt mit dem Quasiquote-Operator, dessen Syntax und Semantik sich leicht von Lisp unterscheidet. Die Verarbeitung von getypten Objekten geschieht durch gewöhnliche Haskell-Funktionen.

In Kapitel 8 wurde gezeigt, wie Parser-Arrows durch eine Verschachtelung der einfachen Typen `FailFunctor` und `StateFunctor` konstruiert werden können. `FailFunctor` kapselt hierbei die Möglichkeit, mit der Rückgabe einer Fehlermeldung zu scheitern, während `StateFunctor` die Weitergabe eines Zustandes analog zur `IO`-Monade beschreibt und für den Eingabe-Stream verwendet wird. Darüber hinaus wurde eine mit EBNF vergleichbare, jedoch allgemeinere und erweiterbare Kombinatorbibliothek entwickelt, mit der Parser-Arrows auf funktionale Weise zusammengesetzt werden können.

In Kapitel 9 wurde zunächst die Generator-Bibliothek vorgestellt, mit dem zu erzeugender Quelltext durch ein strukturiertes Code-Modell repräsentiert wird, was unter anderem hilft, Syntaxfehler oder Probleme mit der Einrückung zu vermeiden. Wie die Parser-Bibliothek kann auch die Generator-Bibliothek vom Benutzer einfach um allgemeine oder sprachspezifische Abstraktionen erweitert werden. Eine S-Expression-Bibliothek enthält neben einem Parser und einem Generator für S-Expression-Syntax S-Expression-spezifische Erweiterungen der Parser-Bibliothek wie das von Lisp-Makros bekannte Matching von Knotennamen. Darüber hinaus wurde eine universelle Compiler-Schnittstelle spezifiziert, mit der verschiedene Compiler-Typen in den allgemeinsten Compiler-Typ `IOArrow` überführt und so miteinander kombiniert werden können. Außerdem können Verarbeitungseinheiten fest verdrahtet und so implizit über ihre Ein- und Ausgabetypen angesprochen werden.

Mit den vorangehenden Werkzeugen ist es bereits möglich, eigene Parser, Makros etc. in Haskell zu definieren und zu benutzen. In Kapitel 10 wurde damit eine S-Expression-Syntax für Haskell konstruiert, die als Metaprogrammiersprache für Compilerdefinitionen gedacht ist und darüber hinaus die S-Expression-basierte Generierung von Haskell-Code aus anderen DSLs ermöglicht. Die Implementation des Haskell-DSL-Compilers dient gleichzeitig als Anwendungsbeispiel für Compiler in MagicL, die mit typisierten Objekten als Zwischenmodellen arbeiten. Hierfür wandelt ein Parser die S-Expressions zunächst in ein Haskell-Modell um, woraus ein funktionaler Compiler schließlich Haskell-Quelltext generiert. Beide Compiler-Teile benutzen die universelle Compiler-Schnittstelle aus Kapitel 9.

Aufbauend auf S-Expression-Haskell wurde in Kapitel 11 die Compiler-DSL entwickelt, die die Haskell-DSL um einige syntaktische Konstrukte für die S-Expression-Verarbeitung erweitert. Hierfür wurde zunächst ein Quasiquote-Operator für S-Expressions wie in Lisp üblich entwickelt, der sich allerdings – neben der etwas anderen Syntax – in zwei Punkten unterscheidet: Er ist allgemeiner, da auch mehrere oder überhaupt keine Rückgaben ermöglicht werden, und löst Namensprobleme bei der Generierung von

weiteren Templates einfach durch verschieden benannte Quasiquote-Operatoren. Darüber hinaus wurden sogenannte Auto-Makros bereitgestellt, welche die Lisp-Semantik einer globalen Makro-Liste, die automatisch rekursiv von außen nach innen auf eines Syntaxbaum angewendet wird, wiederspiegelt.

Eine Diskussion des entwickelten Frameworks in Hinblick auf Erfüllung der Anforderungen und praktische Verwendbarkeit wurde in Kapitel 12 geführt. Alle Anforderungen wurden erfüllt, so dass MagicL allgemeiner und flexibler ist als alle bekannten bestehenden Werkzeuge. Potential für Verbesserungen besteht hauptsächlich bei der Compiler-DSL, mit der Metaprogramme noch nicht so kurz formuliert werden können wie in Lisp. Es sind daher weitere Abstraktionen wünschenswert, die allerdings nicht einfach die Lisp-Konstrukte nachbauen sollten, da dies die Allgemeinheit wieder einschänken würde. Welche Konstrukte die beste Erweiterung der Compiler-DSL wären, bleibt daher eine offene Frage, die erst nach dem Sammeln von mehr Erfahrung bei der Implementation von MagicL-Compilern beantwortet werden kann.

13.2 Ausblick

Dieser Abschnitt soll einen Überblick über weitere Möglichkeiten geben, wie MagicL zukünftig noch verbessert werden kann. Dabei wird diskutiert, welche Entwurfsentscheidungen in einem möglichen Nachfolger anders getroffen werden könnten und an welchen Stellen noch Nachbesserungen oder Erweiterungen möglich sind. Darüber hinaus sollen Ideen für S-Expression-gestützte Modellierung und Verarbeitung aufgezählt werden, die hier nicht weiter verfolgt wurden, aber interessante Forschungsrichtungen für die Zukunft darstellen könnten.

13.2.1 Nutzung in der realen Welt

Generell lag bei der Implementation von MagicL die Priorität auf theoretischer Korrektheit und klarer Semantik, jedoch weniger auf konkreten Anwendungen. Dies spiegelt sich in der Wahl von Haskell und Kategorientheorie als Basistechnologien wieder, welche zwar mathematisch elegant sind, sicherlich jedoch die meisten Anwender abschrecken würden. Hierfür könnte eine einfachere Compiler-DSL nützlich sein, die ohne mathematisches Hintergrundwissen benutzbar ist. Diese DSL könnte dann entweder in die bestehende Arrow-Implementation überführt oder komplett anders realisiert werden – und dabei eventuell effizienter sein als die verschachtelten Parser-Funktoren, die vergleichsweise langsam laufen.

Ein weiterer Aspekt der Realweltnutzung besteht in der Integrierbarkeit in bestehende Systeme. MagicL-Compiler können mit einem installierten Haskell-System interpretiert oder mit dem Glasgow Haskell Compiler (GHC) [3] in native Programme übersetzt werden, welche von beliebigen Build-Skripten aufgerufen werden können. Dennoch wären Interoperabilität und Portabilität bei einer Umsetzungssprache wie Java deutlich höher.

Auch fehlen bis auf Haskell noch jegliche Backends für Programmier- und Auszeichnungssprachen – derzeit müssen diese komplett vom Benutzer selbst erstellt werden.

13.2.2 Theoretische Verallgemeinerungen

Es gibt auch einige Ideen, wie die grundlegende Architektur von MagicL noch universeller und mächtiger angelegt werden könnte.

Ein Beispiel hierfür ist der Umstieg von S-Expressions, deren Blätter immer Strings sind, auf beliebige Bäume:

```
data Tree a = Node [Tree a]
            | Leaf a
```

Damit könnten Parser, Makros etc. unabhängig von der Frage nach dem Blatttyp konstruiert werden. Die bisherigen S-Expressions entsprechen dann `Tree String`, eventuell ist allerdings auch eine Repräsentation als `Tree Char` sinnvoll, bei der Symbole selbst als Stream von Buchstaben aufgefasst werden und daher ohne zusätzlichen Aufwand auch mit String-Parsern gelesen werden könnten. Dies würde allerdings nicht zur Konvention passen, das erste Element einer Liste immer als Knotennamen zu verstehen.

Erweiterungspotenzial gibt es ebenfalls bei der generischen Compiler-Schnittstelle, welche zwar das automatische Finden von Compilern über Typen, nicht aber die selbstständige Suche von Pfaden über mehrere Stationen ermöglicht. Dies würde nur funktionieren, wenn das Haskell-Typsystem eine Prolog-artige Tiefensuche [11] unterstützen würde oder eine künftige Implementation nicht über Haskell-Typen, sondern eigene Datenstrukturen erfolgt. Eine Umsetzung in einer dynamisch typisierten Sprache wie Clojure [22] könnte hilfreich sein.

Elemente von Prolog ergeben auch an anderen Stellen interessante Möglichkeiten. Beispielsweise könnte versucht werden, Makros oder allgemein Parser als Relationen zu definieren, wodurch diese bidirektional verwendbar würden. Ein Parser könnte daher gleichzeitig als Generator dienen, und Sprach-Backends könnten ein gewisses Maß an Roundtrip-Engineering [13] ermöglichen. Bei Mehrdeutigkeiten oder überhaupt keinen Lösungen sollte ein solches System Benutzerinteraktionen zur Auflösung oder Fehlersuche ermöglichen.

13.2.3 S-Expression-Entwicklungsumgebung

Generell könnte eine „S-Expression-orientierte“ Form der Softwareentwicklung sehr stark von darauf zugeschnittenen, integrierten Entwicklungsumgebungen profitieren. Es gibt keinen Grund, weshalb S-Expressions weiterhin altmodisch als Textdateien bearbeitet werden müssen. Vielmehr wäre ein struktureller Baumeditor zu bevorzugen – analog zur Präferenz von S-Expression-Templates gegenüber Strings. Der ParEdit-Mode [5] für Emacs [9] ist ein Schritt in diese Richtung, da dieser zwar weiterhin Text anzeigt, seine Steuerung allerdings fast komplett Baum-basiert ist. Damit wird es beispielsweise unmöglich, unbalancierte Klammern zu erzeugen.

Noch besser wäre allerdings der komplette Umstieg auf Bäume im Editor, womit beispielsweise gezieltes Ausblenden von Sub-Ausdrücken erleichtert würde. Vor allem aber würde dies eine Model-View-Trennung für Programmiersprachen bedeuten, in denen das Modell aus S-Expressions besteht, die graphische Visualisierung aber komplett überschreibbar ist, was nahezu unbegrenzte Möglichkeiten nach sich zieht. Beispielsweise

könnten Wurzeln, Brüche etc. in mathematischen Programmen graphisch ansprechend visualisiert oder programmiererspezifische Themes angeboten werden, welche die syntaktischen Strukturen gewohnter Programmiersprachen für bessere Lesbarkeit simulieren.

13.2.4 Datenbanken und Vernetzung

S-Expressions müssen genauso wenig als Text gespeichert wie visualisiert werden. Vielmehr können sie auch direkt persistent und effizient in geeigneten lokalen oder entfernten Datenbanken abgelegt werden. Als Backend hierfür könnten sowohl schemafreie Systeme wie CouchDB [6] als auch klassische SQL-Datenbanken zum Einsatz kommen, wobei letztere mit einem allgemeinen S-Expression-Schema oder modellspezifischen, generierten Schemas verwendet werden könnten.

Auch für das Problem der Versionierung in verteilten Entwicklungsszenarien könnten S-Expressions Vorteile gegenüber textbasierten Versionierungssystemen wie Subversion [38] oder Git [10] mit sich bringen, wenn man beispielsweise an die immer wieder auftretenden Probleme mit Whitespace denkt. Auch dies ist weiter untersuchenswert.

Ein weiteres Arbeitsgebiet könnte die Möglichkeit von Echtzeitkollaboration durch Netzwerktransparenz von S-Expressions oder kommunizierende Datenbanken darstellen, vor allem im Zusammenhang mit der skizzierten Entwicklungsumgebung.

Generell erscheinen die Möglichkeiten, die sich aus der Idee einer Vereinheitlichung existierender neuer Programmier- und Auszeichnungssprachen in eine einfache Struktur wie S-Expressions ergeben, nahezu grenzenlos. Neue DSLs, die in die Bestehenden übersetzt werden, könnten so immer abstraktere und einfachere Notationen für Domänen anbieten und so das Erstellen von Programmen, Webseiten und beliebigen anderen Daten vereinfachen. Auch wenn die Vision einer großen, einheitlichen und redundanzfreien Sammlung von S-Expression-DSLs für jegliche Anwendungsdomäne noch in weiter Ferne scheint, geht diese Arbeit durch die Diskussion und prototypische Umsetzung eines geeigneten Rahmenwerks für derartige DSL-Compiler einen ersten Schritt in diese Richtung.

Literaturverzeichnis

[1] *Boost C++ Libraries*. http://www.boost.org. Letzter Zugriff: 24.11.2010.

[2] *Computing Fibonacci Numbers at Compile Time*. http://tinyurl.com/33u3hbw. Letzter Zugriff: 24.11.2010.

[3] *The Glasgow Haskell Compiler*. http://www.haskell.org/ghc/. Letzter Zugriff: 24.11.2010.

[4] *Template Haskell Tutorial*. http://www.haskell.org/bz/thdoc.htm. Letzter Zugriff: 24.11.2010.

[5] *EmacsWiki: ParEdit*. http://www.emacswiki.org/emacs/ParEdit, 2009. Letzter Zugriff: 24.11.2010.

[6] Anderson, J. Chris, Jan Lehnardt und Noah Slater: *CouchDB: The Definitive Guide*. O'Reilly, Sebastopol, 2010.

[7] Apache Software Foundation: *The Apache Velocity Project*. http://velocity.apache.org, 2008. Letzter Zugriff: 24.11.2010.

[8] Bongers, Frank: *XSLT 2.0 und XPath 2.0*. Galileo Computing, Bonn, 2. Auflage, 2008.

[9] Cameron, Debra, Bill Rosenblatt und Eric Raymond: *Learning GNU Emacs: A Guide to the World's Most Extensible, Customizable Editor*. O'Reilly, Sebastopol, 2004.

[10] Chacon, Scott: *Pro Git*. Apress, New York, 2009.

[11] Clocksin, William. F und Christopher. S Mellish: *Programming in Prolog. Using the ISO Standard*. Springer, Berlin, 2003.

[12] Dalheimer, Kalle und Karsten Günther: *LaTeX - kurz & gut*. O'Reilly, Köln, 3. Auflage, 2008.

[13] Dirkner, Ragna: *Roundtrip-Engineering im PAOSE-Ansatz*. Diplomarbeit, Department Informatik, Universität Hamburg, 2006.

[14] Ehrig, Hartmut, Bernd Mahr, Felix Cornelius, Martin Grosse-Rhode und Philip Zeitz: *Mathematisch-strukturelle Grundlagen der Informatik*. Springer, Berlin, 2. Auflage, 1999.

[15] Federano, John: *Lisp is a Chameleon*. http://www.paulgraham.com/chameleon.html, September 1991. Letzter Zugriff: 24.11.2010.

[16] Flanagan, david: *JavaScript: The Definite Guide*. O'Reilly, Sebastopol, 1998.

[17] Forman, I.R. und N. Forman: *Java reflection in action*. In Action series. Manning, Greenwich, 2004.

[18] Forte, Stephen: *Using Oslo to Speed Up Database Development*. http://msdn.microsoft.com/en-us/library/ee424598.aspx, 2009. Letzter Zugriff: 24.11.2010.

[19] Fung, Jane, Christina Lau, Ellen McKay, Valentina Birsan, Colin Yu, Joe Winchester, Gili Mendel, Gary Flood, Peter Walker, Timothy deBoer, Yen Lu und James Hunter: *An Introduction to IBM Rational Application Developer: A Guided Tour*. IBM Press, Indiana, 2005.

[20] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley Verlag, München, 2004.

[21] Gibbons, Jeremy und Oege de Moor (Herausgeber): *The Fun of Programming*. Palgrave Macmillan, Hampshire, 2003.

[22] Halloway, Stuart: *Programming Clojure*. The Pragmatic Programmers, LLC, Raleigh und Dallas, 2009.

[23] Hedtstück, Ulrich: *Einführung in die Theoretische Informatik: Formale Sprachen und Automatentheorie*. Oldenbourg, München, 4. Auflage, 2007.

[24] Hudak, Paul, John Hughes, Simon Peyton Jones und Philip Wadler: *A history of Haskell: being lazy with class*. In: *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, Seiten 12–1–12–55, New York, 2007. ACM.

[25] Hughes, John: *Generalising monads to arrows*. Science of Computer Programming, 2000.

[26] Hunsicker, Marco: *Jalopy User Manual*. http://jalopy.sourceforge.net/existing/manual.html, 2002. Letzter Zugriff: 24.11.2010.

[27] Kecher, Christoph: *UML 2: Das umfassende Handbuch*. Galileo Computing, Bonn, 3. Auflage, 2009.

[28] Kiczales, Gregor und Jim Des Rivieres: *The Art of the Metaobject Protocol*. MIT Press, Cambridge, USA, 1991.

[29] Kuchana, Partha: *Software architecture design patterns in Java*. CRC Press, Boca Raton, 2004.

[30] Leijen, Daan und Erik Meijer: *Parsec: A Practical Parser Library*. Microsoft Research, Redmond, 2001. http://research.microsoft.com/en-us/um/people/emeijer/Papers/Parsec.pdf. Letzter Zugriff: 24.11.2010.

[31] McCarthy, John: *The Implementation of LISP*. http://www-formal.stanford.edu/jmc/history/lisp/node3.html. Letzter Zugriff: 24.11.2010.

[32] McCool, M.D. und S.D. Toit: *Metaprogramming GPUs with Sh*. Ak Peters Series. A K Peters, London, 2004.

[33] McDermott, Drew: *YTools: A Package of Portable Enhancements to Common Lisp*. http://www.cs.yale.edu/homes/dvm/papers/ytdoc.pdf, 2008. Letzter Zugriff: 24.11.2010.

[34] Naftalin, Maurice und Philip Wadler: *Java Generics and Collections*. O'Reilly, Sebastopol, 2006.

[35] Norvig, Peter: *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, San Fransisco, 1992.

[36] Parr, Terence: *Enforcing Strict Model-View Separation in Template Engines*. http://www.cs.usfca.edu/ parrt/papers/mvc.templates.pdf. Letzter Zugriff: 24.11.2010.

[37] Perrotta, P.: *Metaprogramming Ruby: Program Like the Ruby Pros*. Pragmatic Bookshelf Series. The Pragmatic Programmers, LLC, Raleigh und Dallas, 2010.

[38] Pilato, C. Michael, Ben Collins-Sussman und Brian W. Fitzpatrick: *Version Control with Subversion. Next Generation Open Source Version Control*. O'Reilly, Sebastopol, 2. Auflage, 2008.

[39] Ritchie, Dennis M.: *The development of the C language*. SIGPLAN Not., 28(3):201–208, 1993.

[40] Sedach, Vladimir: *ParenScript Tutorial*. http://common-lisp.net/project/parenscript/tutorial.html, 2009. Letzter Zugriff: 24.11.2010.

[41] Seibel, Peter: *Practical Common Lisp*. Apress, New York, September 2007.

[42] Steele, Guy L. und Richard P. Gabriel: *The evolution of Lisp*. In: *History of programming languages II*, Seiten 233–330. ACM, New York, 1996.

[43] Tantau, Till: *User Guide to the Beamer Class*. http://www.ctan.org/tex-archive/macros/latex/contrib/beamer/doc/beameruserguide.pdf, 2007. Letzter Zugriff: 24.11.2010.

[44] World Wide Web Consortium: *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. http://www.w3.org/TR/2008/REC-xml-20081126, 2008. Letzter Zugriff: 24.11.2010.

Abbildungsverzeichnis

Selbstständigkeitserklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Departments Informatik einverstanden.

Hamburg, 27. November 2010

Benjamin Teuber