HUMBOLDT-UNIVERSITÄT ZU BERLIN
MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT
INSTITUT FÜR INFORMATIK

Über die Konstruktion von Compilern für höhere Zielsprachen

Diplomarbeit

zur Erlangung des akademischen Grades
Diplom-Informatiker (Dipl. Inf.)

eingereicht von: Dorian Weber
geboren am: 21. Februar 1986
geboren in: Berlin

Gutachter: Prof. Dr. sc. Joachim Fischer
Prof. Dr. rer. nat. Jens-Peter Redlich

eingereicht am: ..... verteidigt am: .....

Why program by hand in five days what you can spend five years of your life automating?

— Terence Parr

Danksagung

Mein aufrichtiger Dank geht an Prof. Joachim Fischer für seine Beratung und Begleitung meiner Arbeit. Ich bin dankbar dafür, dass er mein Potential erkannt und mir ein interessantes Problem zur Bearbeitung überlassen hat, und auch für die Möglichkeit in einem stimulierenden, wissenschaftlichen Umfeld mit ehrlicher, schneller und direkter Rückmeldung zu arbeiten. Seine Geduld mit mir ist ein wichtiger Faktor dafür, dass ich das Thema gründlich untersuchen konnte.

Weiterhin möchte ich mich bei Dr. Klaus Ahrens und Ingmar Eveslage bedanken, deren Betreuung einen wesentlichen Einfluss auf die Struktur und den Inhalt dieser Arbeit ausgeübt hat. Sie waren jederzeit für kompetente Hilfestellungen bereit und haben mich durch ihre Anleitung unterstützt. Die Mitarbeiter des Lehrstuhls von Prof. Fischer haben meine Arbeit mit ihrer objektiven und direkten Kritik verbessert.

Mein Dank geht an Dr. Andreas Kunert für seine ausführlichen Korrekturen von typographischen und Formatierungsfehlern sowie Ungenauigkeiten in der vorläufigen Version des Haupttextes. Adrian Petrov und Lukas Zühl bin ich dankbar für ihre mathematische bzw. elektrotechnische Sichtweise bei der Bewertung von Konsistenz und Verständlichkeit der Arbeit sowie die fruchtbaren Diskussionen bei der Bearbeitung der Aufgabenstellung.

Ich möchte die Gelegenheit nutzen, meinen Großeltern, Bärbel und Manfred Hopp, meine unendliche Dankbarkeit für ihre kontinuierliche Unterstützung aller meiner Unternehmungen auszudrücken. Ohne sie wäre der erfolgreiche Abschluss meines Studiums nicht möglich gewesen.

Meiner Freundin, Tatyana Povolotsky, gebührt großer Dank für den inspirierenden, motivierenden und stetigen Rückhalt, auf den ich mich immer verlassen konnte und der mich sehr entlastet hat.

Inhaltsverzeichnis

Programm-Listings

Abbildungsverzeichnis

Tabellenverzeichnis

Zusammenfassung

Bei der Implementierung von Compilern für Hochsprachen mit textueller Notation ergibt sich häufig die Situation, dass bereits Übersetzer für die Quellsprache existieren, die jedoch nicht in die gewünschte Zielsprache abbilden. Das Thema dieser Arbeit ist die Analyse, Diskussion und Verbesserung bestehender Ansätze, die die Codegenerierungskomponente eines Übersetzers durch jene einer anderen Zielsprache austauschen, ohne dessen Analyse- und Optimierungsphasen zu modifizieren. Dafür vergleiche ich spezifikationsbasierte (*UML*), bibliotheksbasierte (*C*++-Standardbibliothek), schablonenbasierte (StringTemplate), DSL-basierte (*Cheetah* und *Xtend*) sowie metamodellbasierte (*Xtext*+*Xtend*) Ansätze auf ihre Eignung und leite daraus Anforderungen für eine domänenspezifische Sprache zur Codegenerierung ab.

Unter Verwendung dieser Erkenntnisse entwerfe ich die Sprache *Glue*, diskutiere Designentscheidungen und Herausforderungen bei der Implementation und demonstriere ihren Einsatz anhand der Übersetzung eines *ASN.1*-Sprachfragmentes nach *C*.

1. Einleitung

Der Lehrstuhl für Systemanalyse von Prof. Fischer am Institut für Informatik der Humboldt-Universität zu Berlin beschäftigt sich im Rahmen verschiedener Projekte mit selbstorganisierenden Sensornetzwerken. In diesem Umfeld ergeben sich theoretische und praktische Fragestellungen, deren Untersuchung voraussetzt, dass in den untersuchten Netzwerken grundlegende Funktionen wie etwa Versand, Empfang oder Weiterleitung von Nachrichten bereitgestellt werden. Für die Kommunikation in einem Netzwerk ohne geteilten Zugriff auf gemeinsamen Speicher müssen Daten in serialisierter Form vorliegen. Die derzeitige Lösung dieses Problems ist im Bereich der Telekommunikation die Beschreibungssprache *ASN.1*, die die standardisierte Beschreibung und Serialisierung komplexer Datenstrukturen ermöglicht und vereinheitlicht. Die für den Zweck der Serialisierung definierten Regeln erzeugen eine maschinenunabhängige Repräsentation von Datenstrukturen, so dass sich diese von Rechnern beliebiger Hard- und Softwarekonfiguration wieder deserialisieren lassen, was insbesondere in heterogenen Netzwerken von Bedeutung ist. Die Sprache adressiert auch Probleme beim gleichzeitigen Betrieb verschiedener Versionen einer Spezifikation, wie sie etwa in einem aktiven Sensornetzwerk auftritt, das während der Aktualisierungsphase nicht heruntergefahren wird. Ebenfalls

Grammatiktyp	Einschränkung
Typ 0: unbeschränkt	$\alpha \rightarrow \beta$
Typ 1: kontextsensitiv	$\alpha \rightarrow \beta$ für $\lvert\alpha\rvert \leq \lvert\beta\rvert$
Typ 2: kontextfrei	$A \rightarrow \beta$
Typ 3: regulär	$A \rightarrow aB$ oder $A \rightarrow a$
$\alpha \in \{T \cup N\}^+; \beta \in \{T \cup N\}^*; A, B \in N; a \in T$	

Tabelle 1: *Hierarchie für Grammatiken $G = (N, T, P, S)$ nach Chomsky [1].*

unterstützt wird die Notation von *Constraints* – Beschränkungen im Wertebereich von Variablen – die sich auf atomare und zusammengesetzte Datenstrukturen sowie die Relationen der Komponenten untereinander beziehen können.

Die große Ausdruckskraft führt zu einer hohen Sprachkomplexität, die ein Grund dafür ist, dass nur wenige nicht-kommerzielle automatische Übersetzer existieren. Das Ziel meiner Arbeit ist die Entwicklung geeigneter Werkzeuge für die Konstruktion eines solchen Übersetzers für mehr als eine Zielsprache.

1.1. Formale Sprachen

Unter einer *formalen Sprache* versteht man die Menge aller syntaktisch erlaubten Ausdrücke bezüglich einer Chomsky-Grammatik. Ein *Alphabet* ist eine beliebige, endliche Menge von Symbolen. Eine *Chomsky-Grammatik* $G = (N, T, P, S)$ besteht aus einem Alphabet von Nichtterminalsymbolen N, einem Alphabet von Terminalsymbolen T, einer Menge von Produktionen P der Form $\alpha \rightarrow \beta$ und einem Startsymbol S. Mithilfe des Startsymbols und den Produktionen lassen sich Elemente der Sprache ableiten, indem das Wort auf der linken Seite einer Produktion durch das auf der rechten ersetzt wird. Die Sprache definiert sich über den Abschluss dieser Operation mit der Einschränkung, dass nur noch Terminalsymbole vorkommen dürfen.

Durch weitere Einschränkungen in der Form der Produktionen können Sprachen in die *Chomsky-Hierarchie* eingeteilt werden, die in Tabelle 1 dargestellt ist. Im Rahmen dieser Arbeit beschränke ich mich bei der Verwendung des Sprachbegriffes auf die Klasse der kontextfreien Typ 2 Sprachen, die durch formale, semiformale oder informale Semantikbeschreibungen angereichert wurden. In diese Kategorie fällt ein Großteil der

in der Praxis eingesetzten Programmier- und Beschreibungssprachen, da effiziente Algorithmen und Werkzeuge für diese Art der Sprachdefinition möglich sind und entwickelt wurden.

Der *Abstraktionsgrad* einer Programmiersprache ist ein Maß für die Distanz zu direktem, imperativ ausgeführten Maschinencode mit der Metrik, dass größere Entfernungen mit der Auslassung von mehr Implementationsdetails bei der Beschreibung von Programmen korrelieren. Eine Sprache, die von konkretem Maschinencode abstrahiert, wird als *Hochsprache* bezeichnet. Beim Vergleich zwischen zwei formalen Sprachen wird die Sprache mit dem größeren Abstraktionsgrad *höher* genannt.

Grundsätzlich besteht das Problem, dass höhere Sprachen bei der Ausführung oftmals mehr Ressourcen, etwa Rechenzeit oder Speicherplatz, benötigen und/oder an Ausdruckskraft einbüßen. Eine Möglichkeit diese negativen Effekte einzudämmen besteht mit der Benutzung *domänenspezifischer Sprachen* (*domain-specific language*: DSL). Durch die Interpretation von Programmen im Kontext einer Domäne kann ein höheres Abstraktionsniveau erreicht und Effizienzverluste minimiert werden, da die Beschränkung auf eine spezifische Domäne den Einsatz spezialisierter Konzepte erlaubt. Die Sprache ist dann allerdings weniger nützlich oder gänzlich ungeeignet für die Lösung von Problemen außerhalb dieser Domäne.

1.2. Übersetzer und Interpreter

Ein Programm, das eine formale Sprache in eine andere transformiert ohne die Semantik zu verändern, wird als *Übersetzer* bezeichnet. Die Eingabesprache heißt *Quellsprache*, die Ausgabesprache heißt *Zielsprache* und die Implementationssprache des Übersetzers heißt *Implementationssprache*. Falls die Quellsprache einen höheren Abstraktionsgrad als die Zielsprache hat, wird der Übersetzer *Compiler* genannt. Ein *optimierender Compiler* versucht die negativen Effekte des höheren Abstraktionsgrades auf den Ressourcenaufwand zur Ausführungszeit durch statisches Wissen abzuschwächen, indem Veränderungen an der Ausgabe durchgeführt werden, die die Semantik unverändert lassen, sich jedoch positiv auf den Ressourcenverbrauch auswirken. *Statisches Wissen* bezeichnet Erkenntnisse, die entweder universell gelten (wie etwa mathematische Zusammenhänge) oder sich aus der konkreten Konfiguration des zu übersetzenden Codes ergeben. Ein *Interpreter* übersetzt eine Hochsprache in Maschinencode und führt diesen direkt aus.

Abbildung 1 zeigt den typischen Aufbau eines Compilers. Die Eingabe besteht aus einer oder mehrerer Quelldateien in Textform, die verschie-

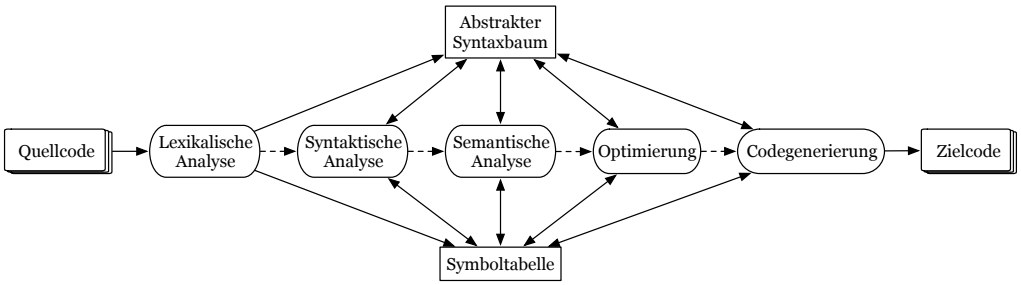


Abbildung 1: *Kanonische Compilerarchitektur nach Garlan u. Shaw [5]. Rechtecke stehen für Datenstrukturen, gestapelte Rechtecke entsprechen externen Dateien und abgerundete Rechtecke markieren Phasen. Gestrichelte Pfeile zeigen den Programmfluss, ungestrichelte den Datenfluss an. Pfeile auf eine Datenstruktur erlauben deren Veränderung, Pfeile ausgehend von einer Datenstruktur gestatten Lesezugriff.*

dene Phasen der Validierung und Transformation durchlaufen. In den Analysephasen werden die Quelldateien auf ihre syntaktische und semantische Korrektheit geprüft und ein *abstrakter Syntaxbaum* konstruiert. Dabei handelt es sich um einen gerichteten, zyklenfreien Graphen, der die eingelesenen Quelldateien ohne Verweise auf die konkrete Syntax der Quellsprache repräsentiert. Der Syntaxbaum enthält häufig Referenzen auf Bezeichner aus der Quelldatei, welche in der *Symboltabelle* zusammen mit deren Attributen hinterlegt sind. Der abstrakte Syntaxbaum und die Symboltabelle dienen der Optimierungsphase als Eingabe, deren Ziel die Verringerung des Ressourcenverbrauches bei der Ausführung des erzeugten Artefaktes ist. Dazu werden für die Knoten des Syntaxbaumes Informationen berechnet, die sich statisch aus der Konfiguration ergeben und welche für verschiedene, semantikneutrale Baumtransformationen verwendet werden. In der Codegenerierungsphase wird der resultierende Syntaxbaum in Zielcode umgewandelt und ausgegeben.

1.3. ASN.1

ASN.1 steht für *Abstract Syntax Notation One* und ist sowohl ein Sprachstandard zur Beschreibung von Datenstrukturen [16, 17, 18, 19] als auch eine Sammlung regelbasierter Verfahren mit denen sich diese Strukturen serialisieren und deserialisieren lassen [20, 21, 22, 23]. Was *ASN.1* vor vergleichbaren Notationen auszeichnet, umfasst nach Larmouth [9] die folgenden Punkte:

- Es ist eine international standardisierte, hersteller-, plattform- und sprachunabhängige Beschreibungssprache für Datenstrukturen auf einem hohen Abstraktionsniveau.

- Es werden plattform- und sprachunabhängige Regeln vorgegeben, aus denen sich die konkreten Bitmuster der Werte dieser Datenstrukturen ableiten und effizient repräsentieren lassen.

- Es steht eine Vielzahl kommerzieller Compiler für die meisten Plattformen und verschiedene Programmiersprachen zur Verfügung, die neben der Abbildung der Datenstrukturen auch Code für die Kodierungen konkreter Belegungen erzeugen.

- Die Sprache adressiert Probleme bei der Versionierung von Spezifikationen und deren gleichzeitigen Betrieb, indem „Version 1“-Systeme aufwärtskompatibel zu „Version 2“-Systemen spezifiziert werden können.

- Es werden Mechanismen bereitgestellt, mit denen generische oder unvollständige Spezifikationen entwickelt werden können, die sich dann von anderen Entwicklergruppen spezialisieren lassen.

- Potentielle Probleme zwischen großen Systemen, die in der Lage sind auch lange Zeichenketten, große Zahlen und komplexe Strukturen zu verarbeiten und kleinen Systemen, die diese Fähigkeit nur in geringerem Maße besitzen, werden erkannt.

- Die Auswahl der eingebauten Datenstrukturen ist im Allgemeinen deutlich größer als für normale Programmiersprachen. Beispielsweise lassen sich Zahlen beliebiger Größe und Genauigkeit spezifizieren und Zeichenketten können auf bestimmte Längen oder Alphabete eingegrenzt werden. Dies erlaubt eine hohe Präzision in der Beschreibung der zu serialisierenden Werte, was platzeffiziente Kodierungen ermöglicht.

Ein *Netzwerkprotokoll* besteht aus einer Menge von Semantik-tragenden Nachrichten sowie Regeln, die bestimmen unter welchen Voraussetzungen die Nachrichten zwischen den Akteuren im Netzwerk ausgetauscht werden. *ASN.1* dient der Spezifikation von Nachrichtentypen und legt damit den Grundstein für ein Netzwerkprotokoll.

Anbieter	Lizenz	Zielsprachen	Kodierungsregeln
Lev Walkins	BSD	*C*	BER, PER, XER
Ericsson	EPL	*Erlang*	BER, PER
OSS Nokalva	kommerziell	*C*, *C++*, *C#*, *Java*	BER, PER, OER
Object Systems	kommerziell	*C*, *C++*, *C#*, *Java*	BER, PER, XER
Marben	kommerziell	*C*, *C++*, *Java*	BER, PER, XER
uniGone	kommerziell	*C#*, *Java*	BER, PER, XER
Talura	kommerziell	*C*	BER, XER
BER: X.690 [20], PER: X.691 [21], XER: X.693 [22], OER: X.696 [23]			

Tabelle 2: *ASN.1-Compiler mit Unterstützung für die X.680 Serie von Standards basierend auf [8]. Kommerzielle Lizenzen müssen mit den Anbietern ausgehandelt werden und sind daher nur als kommerziell ausgewiesen. Freie und kommerzielle Implementationen sind durch eine horizontale Linie getrennt.*

1.4. Motivation

Am Lehrstuhl für Systemanalyse wurde ein Netzwerkprotokoll für ein Netzwerk seismischer Sensoren entwickelt, für das die Nachrichtentypen in *ASN.1* spezifiziert sind. Der Versand der Nachrichten sollte dabei in der Programmiersprache *C++* ausgedrückt werden, deshalb erschien es sinnvoll die *ASN.1*-Spezifikation der Nachrichtentypen in diese Sprache zu übersetzen. In meiner Studienarbeit [14] habe ich unter anderem verschiedene, freie *ASN.1*-Compiler auf ihre Eignung für diese Aufgabe überprüft. Ein Resultat dieser Arbeit ist, dass es keine freie Alternative gibt, die den seit 2002 aktuellen *ASN.1*-Standard X.680 [16] umsetzt und objektorientierten *C++*-Code produziert.

Tabelle 2 listet alle durch die ITU in [8] gelisteten *ASN.1*-Compiler auf, die mindestens diesen Standard umsetzen. Nicht enthalten sind Compiler, die nur *ASN.1*-Fragmente oder den veralteten Standard X.208 [15] implementieren sowie solche, die nur Bibliotheksfunktionen zur Kodierung und Dekodierung bereitstellen oder lediglich äquivalente Strukturen einer Zielsprache erzeugen, ohne auf Kodierungsregeln einzugehen. Es ist festzustellen, dass es im Bereich der Open-Source-Compiler nur Unterstützung für *C* und *Erlang* gibt. Auch kommerzielle Übersetzer erlauben nur die Übersetzung in populäre Programmiersprachen, mutmaßlich, weil diese in der Telekommunikationsindustrie primär eingesetzt werden. Der große Sprachumfang und die umfangreiche Auswahl von Kodierungsstandards

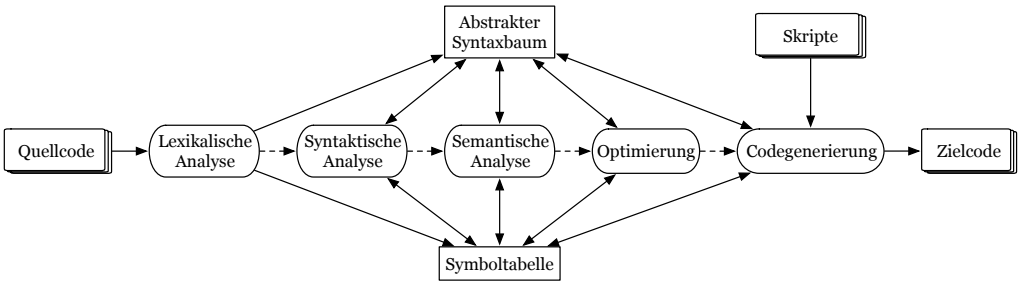


Abbildung 2: *Kanonische Compilerarchitektur mit generischer Codegenerierungskomponente. Die Semantik der Markierung entspricht der aus Abbildung 1. Im Vergleich dazu gestattet die Codegenerierungskomponente in diesem Architekturbild die Parametrisierung der Ausgabe durch den Einsatz von Skriptdateien.*

machen es notorisch kompliziert einen *ASN.1*-Compiler zu implementieren, was sicherlich ein Faktor für deren geringe Verfügbarkeit ist.

Ein weiteres Resultat meiner Studienarbeit ist, dass es einen geeigneten Kandidaten mit guter Unterstützung für aktuelle *ASN.1*-Standards auf der Analyseseite des Übersetzungsvorganges gibt, dessen Codegenerierungsphase darüber hinaus entkoppelt werden kann: den `asn1c` von Walkins [13]. Zunächst schien dessen Anpassung für C^{++}- anstelle von *C*-Code realistisch zu sein, allerdings wurde dabei klar, dass die Codegenerierungskomponente zu komplex und verflochten für direkte Modifikationen ist. Da die komplette Ersetzung der Komponente mit einigem Aufwand verbunden ist, ergab sich die Gelegenheit, ganz grundsätzlich darüber nachzudenken, wie sich diese Arbeit für zukünftige Änderungs- und Erweiterungswünsche minimieren ließe.

1.5. Problemstellung

Gegeben ist das Fragment eines Compilers, das Code einer Quellsprache liest, dessen syntaktische und semantische Korrektheit prüft und in eine interne Repräsentation überführt. Gewünscht ist eine generische Codegenerierungskomponente, die dieses Fragment zu einem vollständigen Compiler macht und einen Mechanismus bereitstellt, mit dem neue Zielsprachen eingeführt werden können.

Mein Lösungsvorschlag ist in Abbildung 2 dargestellt und folgt einem DSL-basierten Ansatz: die Codegenerierung wird durch einen Interpreter unterstützt, der auf der einen Seite eine Schnittstelle für den Import von Informationen durch den Compilerentwickler bereitstellt und diese auf

der anderen Seite einem Zielsprachenentwickler in Form von Variablen zugänglich macht. Letzterer kann diese in Skriptdateien abfragen und Zielsprachencode ableiten. Die Interaktion erfolgt aus Sicht des Compilerentwicklers ausschließlich über die programmatische Schnittstelle des Interpreters, die er zu konfigurieren hat. Aus Sicht des Zielsprachenentwicklers erfolgt die Interaktion über vordefinierte Variablen in Skriptdateien. Dieser Ansatz separiert Modell (interne Repräsentation des Quellcodes) und Präsentation (Zielsprachencode).

1.6. Aufbau der Arbeit

In Abschnitt 2 werden bestehende Ansätze kategorisiert, verglichen und ihre Eignung untersucht. Basierend darauf werden in Abschnitt 3 Anforderungen für eine DSL-basierte Lösung formuliert, eine Auswahl von Implementationsherausforderungen diskutiert und die Lösung an einem *ASN.1*-Sprachfragment demonstriert. Abschnitt 4 enthält ein Fazit und den Ausblick auf zukünftige Arbeiten.

2. Aktueller Stand der Technik

2.1. Bemerkungen

In diesem Abschnitt werden unterschiedliche Lösungsansätze für eine generische Codegenerierungskomponente untersucht und verglichen. Dafür wird das Spektrum möglicher Lösungen in Kategorien unterteilt, für jede Kategorie ein Repräsentant ausgewählt und anhand eines Beispieles untersucht. Die Untersuchung konzentriert sich auf die verwendeten Paradigmen und Konzepte, die von allen Ansätzen der entsprechenden Kategorie geteilt werden, statt auf spezifische Implementationsmerkmale und Designentscheidungen einzelner Kandidaten. Jeder Lösungsansatz bietet prinzipiell die Möglichkeit der Problemlösung, muss allerdings zu variierenden Graden ergänzt werden.

Die verwendete Metrik für die Einschätzung der Position einer Lösung innerhalb des Spektrums misst dessen Abstraktionsgrad, also wieviele Annahmen über die Natur des Problems gemacht werden. Auf der einen Seite des Spektrums befinden sich Ansätze, die lediglich voraussetzen, dass die Implementationssprache einen imperativen Charakter hat, auf der anderen Seite sind solche, die annehmen, dass der Entwickler einen spezifischen Satz von Werkzeugen verwendet. Zwischen diesen Polen sind verschiedene Abstufungen möglich, die im Folgenden genauer beleuchtet werden sollen.

Grundsätzlich zeigt die Untersuchung drei Dinge:

1. Die Ansätze sind nicht nur nach ihrem Abstraktionsgrad geordnet, sondern auch nach der Unterstützung für den Compilerautoren. Konkretere Ansätze nutzen zusätzliche Annahmen aus, um einen höheren Grad an Automatisierung zu erreichen.

2. Zwischen den Ansätzen besteht eine Teilmengenbeziehung: Lösungen mit größerem Abstraktionsgrad gehen als Komponenten in solche mit kleinerem ein.

3. Es besteht ein Bedarf an domänenspezifische Skriptsprachen in der Codegenerierungsdomäne.

Einschränkungen

Anwendungsszenarien für einen Compiler, der unterschiedlichen Zielcode produzieren kann, bestehen in der Erzeugung:

```
1  MyProtocol DEFINITIONS ::= BEGIN
2
3      Integer-Types ::= ENUMERATED {
4          char(8),
5          short(16),
6          int(32),
7          ...
8      }
9
10 END
```

Listing 1: *Beispieleingabe für einen ASN.1-Compiler. Die Spezifikation enthält die Definition des Moduls* MyProtocol*. In Zeile 3 wird ein Enumerationstyp mit drei Einträgen und einem Extension-Marker definiert. Der Extension-Marker in Zeile 7 dient der Markierung von Typen, die durch spätere Protokollversionen erweiterbar sind. Im Falle der Enumeration wären das etwa weitere legale Werte.*

- von Maschinencode für verschiedene Hard- und Softwareplattformen,
- eines Programms für verschiedene Hochsprachen oder
- eines Dokuments, das bestimmte Aspekte des Quellcodes wiedergibt.

Für den ersten Fall gibt es einen allgemein akzeptierten Ansatz: das Programm der Quellsprache wird in eine Zwischenrepräsentation (*Intermediate Representation*, IR) übersetzt und dann mithilfe eines Assemblers an die Zielplattform angepasst. Der Assemblerschritt ist zwar auch eine Form der Übersetzung, allerdings keine, bei der das Abstraktionsniveau verringert wird, daher ist ein Assembler kein Compiler. Der eigentliche Schritt zu einer weniger abstrakten Darstellung ist der Schritt von der Quellsprache in die Zwischenrepräsentation. Da es eher unüblich ist, mehr als eine Zwischenrepräsentation zu erzeugen, lasse ich diesen Fall in meiner Analyse unbeachtet. Ich gehe also im Folgenden davon aus, dass die Zielsprachen eine textuelle Entsprechung in einer Hoch- oder Beschreibungssprache besitzen.

Beispielszenario

Um die verschiedenen Ansätze direkt miteinander vergleichen zu können, habe ich mich für das *ASN.1*-Beispiel aus Listing 1 entschieden, für das

C-Code erzeugt werden soll. Die Schwierigkeit bestand darin, ein aussagekräftiges Szenario zu wählen, das einfach genug ist, um Lösungen direkt in den Text dieser Arbeit einzubetten und interessant genug, um symptomatische Probleme herauszustellen. Das von mir gewählte Beispiel hat Vor- und Nachteile, die ich an dieser Stelle dokumentieren möchte.

Vorteilhaft sind:

- Wichtige Hürden sind zu bewältigen:
  1. *ASN.1*-Enumerationen sind variabel lang und müssen systematisch durchlaufen werden.
  2. Die Serialisierung des Bezeichners „Integer-Types“ muss aufgrund des Bindestrichs angepasst werden.
  3. Die Elemente der Enumeration sind reservierte Schlüsselworte in der Zielsprache.
  4. Am Ende der Struktur befindet sich ein Extension-Marker, der als Element im abstrakten Syntaxbaum auftaucht und dessen Behandlung von der eines Wertes abweicht.
- Der Einsatz des gleichen Szenarios gestattet den direkten Vergleich verschiedener Ansätze.

Das Szenario lässt aufgrund seiner Kürze wichtige Probleme aus:

- *ASN.1*-Enumerationen können sich nicht selbst enthalten oder aufeinander verweisen. Dadurch wird der Umgang mit Rekursion und Abhängigkeiten ausgelassen.
- Die Form der Spezifikation ist regulär und enthält keine optionalen Elemente. Damit ist keine Fallbehandlung beim Verarbeiten eines Enumerationswertes notwendig.

Diese Probleme habe ich zugunsten des vereinfachten Verständnisses ausgelassen, sie lassen sich jedoch in allen Varianten lösen und das Beispiel ist für den Vergleich im Rahmen dieser Arbeit ausreichend.

2.2. Entwurfsmuster und Schnittstellen

Entwurfsmuster sind Lösungsschablonen von bekannten softwaretechnischen Problemen, wie sie auch beim Entwurf einer generischen Codegenerierungskomponente auftreten. *Schnittstellen* treten an der Grenze

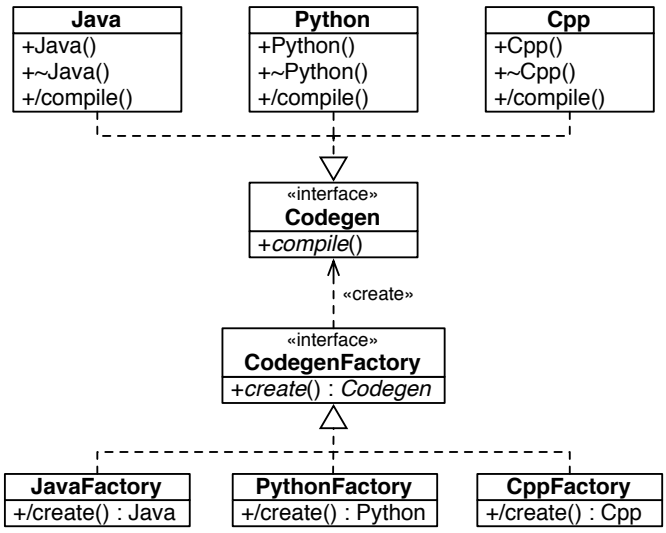


Abbildung 3: *UML-Klassendiagramm für generische Codegenerierungskomponenten. Implementationen der* CodegenFactory*-Schnittstelle haben eine* `create()`*-Methode, mit der sich Objekte, die die* Codegen*-Schnittstelle realisieren, erzeugen lassen. An diesen lässt sich* `compile()` *rufen, das zielsprachenabhängigen Code erzeugt.*

zwischen zwei Programmkomponenten auf und definieren den Rahmen für deren Interaktion. Repräsentativ für diesen Ansatz der Problemlösung verwende ich Entwurfsmuster, die durch Gamma, Helm, Johnson, u. Vlissides [2] beschrieben wurden. Unter Verwendung der Schablonenmethode [3] (*template method pattern*), der Fabrikmethode [4] (*factory method pattern*) und entsprechenden Schnittstellen lässt sich eine kanonische Architektur entwerfen, deren *UML*-Klassendiagramm in Abbildung 3 dargestellt ist.

Das Compilerfragment wählt zur Laufzeit die den Wünschen des Anwenders entsprechende Ableitung der *CodegenFactory* aus und ruft deren spezialisierte `create()`-Methode. In `create()` erzeugt das Fabrikobjekt eine Instanz des Zielsprachengenerators, die die *Codegen*-Schnittstelle implementiert und gibt diese zurück. Der Konstruktor dient der dynamischen Initialisierung der Instanz und kann von Informationen des Compilerfragmentes abhängig sein. Er hat deshalb wie `create()` eine nicht näher spezifizierte Parameterliste. Das Compilerfragment kann daran nun die `compile()`-Methode ausführen. Im Destruktor können dynamische Datenstrukturen, die bei der Abarbeitung von `compile()` oder während der Initialisierung entstanden sind, aufgeräumt werden. Die `compile()`-Methode nimmt in einem kanonischen Compiler mindestens den abstrakten Syn-

taxbaum und die Symboltabelle als Parameter (siehe Abbildung 1 auf Seite 10), allerdings hängt die konkrete Form von der bereits bestehenden Architektur ab. Mithilfe dieser Informationen ist die Methode nun in der Lage, Zielcode der ausgewählten Sprache zu erzeugen.

Eine Stärke dieses Ansatzes ist das hohe Abstraktionsniveau und die daraus resultierende Flexibilität in der Wahl der Programmiersprache, von der unter Verwendung der genannten Entwurfsmuster lediglich angenommen wird, dass sie ihrer Natur nach imperativ ist. Diese Einschränkung ist akzeptabel, da Compiler komplexer Sprachen aus Effizienzgründen häufig in maschinennahen, imperativen Programmiersprachen geschrieben werden. Die Kehrseite dieser Flexibilität ist der geringe Grad an konkreter Unterstützung: es wird lediglich ein Strukturmuster angegeben, jedoch prinzipbedingt keine Angabe dazu gemacht, wie die Implementation aussehen könnte.

2.3. Programmbibliotheken

Ein höherer Grad an Automatisierung kann durch die Einbindung von *Programmbibliotheken* erreicht werden. Programmbibliotheken sind Sammlungen wiederverwendbarer Komponenten einer Programmiersprache und erleichtern die Implementation komplexer Software. Übliche Komponententypen sind *Funktionen*, *Klassen* und *Module*. Funktionen kapseln Algorithmen, Klassen beinhalten Datenstrukturen zusammen mit assoziierten Funktionen und Module sind Sammlungen von Klassen und Funktionen.

Es gibt keine generell akzeptierte Bibliothekslösung für die Implementation einer Codegenerierungskomponente für Zielcode einer Hochsprache, allerdings betreffen einige wiederkehrende Probleme den Umgang mit Zeichenketten, den systematischen Ablauf komplexer Datenstrukturen, die Interaktion mit Dateien, die Aggregation von Informationen sowie die Behandlung von Fehlerzuständen. Unterstützung für diese Art von Problemen ist weit verbreitet und häufig in Standardbibliotheken von Programmiersprachen anzutreffen.

Ein repräsentatives Beispiel dafür findet sich in Gestalt der *Standard Template Library* (STL) der Sprache *C++*. Dabei handelt es sich um eine standardisierte Programmbibliothek mit Unterstützung für Zeichenketten (`string` und `stringstream`), den systematischen Ablauf komplexer Datenstrukturen (`iterator`), die Interaktion mit Dateien (`fstream`), Datenstrukturen für die Aggregation von Informationen (`deque` und `map`) und strukturierte Fehlerbehandlung (`exception`).

```
126 /*
127  * For all ENUMERATED types and for those INTEGER types which
128  * have identifiers, print out an enumeration table.
129  */
130 if(expr->expr_type == ASN_BASIC_ENUMERATED || el_count) {
131         eidx = 0;
132         REDIR(OT_DEPS);
133         OUT("typedef enum ");
134                 out_name_chain(arg, ONC_avoid_keywords);
135         OUT(" {\n");
136         TQ_FOR(v, &(expr->members), next) {
137                 switch(v->expr_type) {
138                 case A1TC_UNIVERVAL:
139                         OUT("\t");
140                         out_name_chain(arg, ONC_noflags);
141                         OUT("_%s", MKID(v));
142                         OUT("\t= %" PRIdASN "%s\n",
143                                 v->value->value.v_integer,
144                                 (eidx+1 < el_count) ? "," : "");
145                         v2e[eidx].name = v->Identifier;
146                         v2e[eidx].value = v->value->value.v_integer;
147                         eidx++;
148                         break;
149                 case A1TC_EXTENSIBLE:
150                         OUT("\t/*\n");
151                         OUT("\t * Enumeration is extensible\n");
152                         OUT("\t */\n");
153                         if(!map_extensions)
154                                 map_extensions = eidx + 1;
155                         break;
156                 default:
157                         return -1;
158                 }
159         }
160         OUT("} e_");
161                 out_name_chain(arg, ONC_noflags);
162         OUT(";\n");
163         assert(eidx == el_count);
164 }
```

Listing 2: *Fragment in C für die Generierung von C-Code aus ASN.1. Abgebildet ist der Ausschnitt einer Funktion aus [13] mit den originalen Zeilennummern, in dem für benannte Zahlen und Aufzählungen die Deklaration eines Enumerationstyps generiert wird. Die Schleife von Zeile 136 bis 159 durchläuft alle Einträge der Enumeration, wobei die Elemente zwischen* `UNIVERVAL` *(Wert) und* `EXTENSIBLE` *(Extension-Marker) unterschieden werden (Zeile 137). Im ersten Fall werden Name und Wert, im zweiten ein Kommentar ausgegeben. Die Funktion* `out_name_chain()` *dient der Anpassung von Bezeichnern an die Zielsprache.*

Mithilfe dieser Abstraktionen lässt sich das Entwurfsmuster aus Abbildung 3 implementieren:

- ▷ Für den systematischen Ablauf des abstrakten Syntaxbaumes können vom Compilerentwickler Iteratoren implementiert werden.
- ▷ Informationen lassen sich mit Hilfe von Listen und Abbildungen aggregieren.
- ▷ Zeichenketten können mit gesammelten Informationen kombiniert und verkettet werden.
- ▷ Programmdateien der Zielsprache lassen sich durch den Einsatz von Dateistreams erzeugen und beschreiben.
- ▷ Probleme, die während der Codegenerierung auftreten, müssen nicht am Ort der erstmaligen Erkennung behandelt werden, sondern lassen sich durch Kontextinformationen ergänzen und an zentraler Stelle lösen.

Vorteilhaft bei der Verwendung dieses Ansatzes ist der verringerte Implementationsaufwand, allerdings wird die Auswahl der Bibliothek durch die Implementationssprache eingeschränkt, da prinzipiell nur Bibliotheken in Frage kommen, für die Schnittstellen für die Implementationssprache bereitgestellt werden. Nachteilhaft ist die Vermischung der Implementations- mit der Zielsprache, was im Verlust der Struktur des generierten Codes resultiert. Listing 2 illustriert das Problem mit einem Codefragment. Die Implementation lässt sich nur schwer nachvollziehen, da die Struktur des Zielcodes von der Struktur der Implementation überdeckt wird. In der Folge gestaltet sich die Wartung und Erweiterung eines solchen Programms mühevoll.

2.4. Schablonen

Um die Lesbarkeit von generiertem Code zu erhöhen, bietet sich die Verwendung von *Codeschablonen* an. Eine Codeschablone ist eine Sequenz von statischem Text, Platzhaltern und bedingten Verweisen auf weitere Codeschablonen. Ein Programm zur Auswertung einer Codeschablone heißt *Template Engine* und beinhaltet im Allgemeinen sowohl einen Algorithmus für die Berechnung der Ausgabe als auch einen Zustand in Form einer Kollektion von Codeschablonen, auf die verwiesen werden kann. Die Sprache,

```
1  import stringtemplate3
2
3  def generate(expr):
4      enum = stringtemplate3.StringTemplate(
5          """
6              typedef enum <expr.name> {
7              <expr.member:{it|<if (it.univerval)>
8                  <it.name>  =  <it.value>,
9              <elseif (it.extensible)>
10                 /*
11                  * Enumeration is extensible
12                  */
13             <endif>}>
14             } e_<expr.name>;
15         """,
16         stringtemplate3.StringTemplateGroup("C", lexer = "angle-bracket")
17     )
18
19     enum["expr"] = expr
20     return str(enum)
```

Listing 3: *Beispiel für die schablonenbasierte Generierung einer ASN.1 Enumeration nach C im Kontext eines Python-Programmes. Das Beispiel benutzt die Syntax beschrieben durch Parr [11] und eingesetzt in seiner* StringTemplate*-Engine. Platzhalter und Verweise auf andere Codeschablonen werden durch Dreiecksklammern von statischem Text separiert. Startend in Zeile 4 wird ein neues String Template definiert. In den Zeilen 7 bis 13 wird eine Schleife über die implizierte Liste von* `expr.member` *ausgeführt, auf dessen Elemente mittels der Variablen* `it` *zugegriffen werden kann. Die Zeilen 8 und 10 bis 12 zeigen die bedingte Inklusion eingebetteter Schablonen, basierend auf der Belegung der* `univerval` *und* `extensible` *Attribute des* `expr`*-Platzhalters. In Zeile 19 wird* `expr` *mit dem Wert des Parameters belegt und in Zeile 20 wird die Schablone durch Umwandlung in eine Zeichenkette durch die Template Engine ausgeführt.*

mit der Schablonen formuliert werden, nenne ich *Schablonendefinitionssprache*. Ich gehe für diese Kategorie davon aus, dass deren Ausdruckskraft auf die Formulierung von Bedingungen für die bedingte Inklusion von weiteren Codeschablonen beschränkt ist, jedoch keine turingvollständigen Ausdrucksmittel bereitstellt.

Prinzipiell wird eine Template Engine in Form einer Programmbibliothek für die Implementationssprache bereitgestellt. Mithilfe zielsprachenspezifischer Codeschablonen lässt sich damit im Rahmen des Codegenerators Zielcode erzeugen, wobei die konkreten Serialisierungen und abgeleiteten Werte durch den Codegenerator berechnet werden. Listing 3 zeigt ein Beispiel für eine Codeschablone mit einer zu Listing 2 äquivalenten Ausgabe.[1]

Parr skizziert in [11] einen Beweis, dass diese Art der Codeschablone jede kontextfreie Sprache erzeugen kann. Da für viele Programmiersprachen[2] eine kontextfreie Grammatik existiert und das Programm die Werte der Platzhalter frei berechnen kann, eignet sich die Template Engine für die Generierung von Hochsprachencode.

Wesentliche Vorteile sind:

- Die Schablonen vermeiden eine Vermischung von Implementations- und Zielcode und erhöhen dadurch die Lesbarkeit, was in der Konsequenz zu vereinfachter Wartung und Erweiterung führt.

- Die Template Engine kann in eine Bibliothek integriert werden und ist damit flexibel genug, um in bestehende Projekte importiert zu werden.

- Schablonen können in externe Dateien ausgelagert und ohne erneute Übersetzung des Compilers angepasst werden.

- Modell (abstrakter Syntaxbaum und Symboltabelle) und Präsentation (Zielcode oder -dokument) sind voneinander getrennt, was es prinzipiell möglich macht, diese Komponenten von verschiedenen Gruppen entwickeln zu lassen.

1 Die Ausgabe unterscheidet sich in einem Detail: im Gegensatz zur Schablone generiert der *C*-Code nach dem letzten Element der Enumeration kein Komma. Da *C* an dieser Stelle ein optionales Komma erlaubt, habe ich die Schablone vereinfacht.

2 Eine Ausnahme dieser Regel ist die Sprache TEX, was die Generierung bestimmter TEX-Programme umständlich macht. Es lassen sich jedoch alle kontextfreien Untermengen erzeugen, was turingmächtige Sprachfragmente mit einschließt.

- Durch die Möglichkeit der Schachtelung von Schablonen ist es für den Entwickler einer neuen Zielsprache möglich, Teile dieser zu faktorisieren, was Änderungen an unterschiedlichen, aber textuell identischen Positionen an eine gemeinsame Stelle verlagert und dadurch Redundanz abbaut.

Nachteilhaft sind:

- Der Entwickler des Modells muss voraussehen, welche Attribute und Informationen im Zielcode benötigt werden, da sich in Schablonen ohne Weiteres keine abgeleiteten Werte berechnen lassen.

- Der Entwickler des Modells muss für alle exportierten Attribute zielsprachenabhängige Serialisierungen anbieten.

  - Das betrifft beispielsweise die unterschiedliche Darstellung von Zeichenketten in *C* und *XML*: die konstante Zeichenkette „*<foobar>*“ lässt sich in *C* genau so darstellen, nicht jedoch in *XML*, da die Dreiecksklammern syntaktische Bedeutung haben und Zeichenketten nicht maskiert werden.

  - Das gleiche Problem tritt für Bezeichner auf, die in Zielsprachen reserviert oder andernfalls illegal sein können, wie etwa der gültige *ASN.1*-Bezeichner „*Integer-Types*“ in *C*.

- Daten müssen in einer für die Template Engine zugänglichen Form vorliegen. Implementationssprachen, die keine kanonischen Schnittstellen für Attributszugriffe, standardisierte Iteratoren oder Introspektion anbieten, müssen Abbildungen in Template-Engine-spezifische Strukturen bereitstellen.

Die Konsequenz aus den ersten zwei Nachteilen ist, dass nicht-triviale Änderungen am Zielcode nur synchron mit der Erweiterung von Implementationscode praktikabel sind, was die Trennung von Modell und Präsentation verwässert. Sie lassen sich durch eine Erweiterung der Schablonendefinitionssprache um in der Summe turingmächtige Ausdrücke kompensieren. Diese Variante fällt in meiner Kategorisierung in den Bereich der domänenspezifischen Sprachen und wird im nächsten Abschnitt diskutiert. Der dritte Nachteil lässt sich durch weitere Mechanismen ausgleichen, die in Abschnitt 2.6 diskutiert werden.

2.5. Domänenspezifische Sprachen

Um die Berechnung abgeleiteter Werte in Codeschablonen zu unterstützen, sind folgende Erweiterungen möglich:

1. Die Template Engine wird durch eine turingmächtige Sprache ergänzt, die direkt in Schablonen notiert wird.

2. Die Template Engine wird mit einer turingmächtigen Sprache kombiniert, indem sie in die Sprache eingebettet wird.

Die resultierende Sprache nenne ich *Synthesesprache*. Im Gegensatz zur Schablonendefinitionssprache ist diese turingmächtig und eignet sich damit sowohl zur Generierung als auch zur Berechnung abgeleiteter Werte. Ein Beispiel für den ersten Ansatz findet sich in Form der Template Engine *Cheetah*, für die in Listing 4 eine Schablone zur Übersetzung einer *ASN.1*-Enumeration nach *C* gezeigt ist. Zum Vergleich findet sich in Listing 5 ein Beispiel mit identischer Ausgabe für den zweiten Ansatz in *Xtend*. Beide Beispiele berücksichtigen den Fall, dass der verwendete *ASN.1*-Bezeichner kein legaler *C*-Bezeichner ist und passen dessen Serialisierung entsprechend an.

Vorteile dieses Ansatzes sind:

- Die Integration einer Template Engine auf der Sprachebene erlaubt direkten Zugriff auf Variablen im umliegenden Gültigkeitsbereich. In der Bibliothekslösung in Listing 3 mussten Variablen vor dem Ausführen der Template Engine explizit belegt werden.

- Die Codesynthese ist vollständig von der Analyse separiert und kann getrennt davon gestaltet werden. Der Architekt des Compilerfragmentes kann die erzeugten Strukturen in Strukturen der Synthesesprache abbilden, ohne die Zielsprache berücksichtigen zu müssen. Auf der anderen Seite der Schnittstelle hat der Zielsprachenentwickler Zugriff auf die bereitgestellten Strukturen, kann deren Serialisierungen festlegen und abgeleitete Werte berechnen.

Nachteilhaft am Einsatz einer zusätzlichen Sprache ist, dass deren Artefakte nicht unbedingt binärkompatibel zu denen der Implementationssprache sind. *Xtend* läuft beispielsweise in der Java Virtual Machine (JVM) und ist daher nicht binärkompatibel zu den Artefakten, die ein *C*-Compiler produziert. In diesem Fall ist eine Kompatibilitätsschicht für die Interaktion der beiden Sprachen notwendig, die im Zweifelsfall durch den Entwickler des

```
1  #import re
2  ## prepare a list of reserved words
3  #attr $keywords = { "auto", "break", "case", "char", "const", "continue", "default",
↪  "do", "double", "else", "enum", "extern", "float", "for", "goto", "if", "inline", "int", "long",
↪  "register", "restrict", "return", "short", "signed", "sizeof", "static", "struct", "switch",
↪  "typedef", "union", "unsigned", "void", "volatile", "while" }
4  ## define a function that ensures the validity of an identifier
5  #def mkid(str)
6        #set $mask = re.sub("[^a-zA-Z0-9]+", "_", str)
7        #if $mask in $keywords
8              #set $mask = $mask.capitalize()
9        #end if
10       #return $mask
11 #end def
12 ## generate the enumeration
13 typedef enum ${mkid($expr.name)} {
14 #for $it in $expr.member
15       #if $it.univerval
16       ${mkid($it.name)}       =       $it.value,
17       #elif $it.extensible
18       /*
19        * Enumeration is extensible
20        */
21       #end if
22 #end for
23 } ${mkid('e_' + $expr.name)};
```

Listing 4: *Beispiel für die skriptbasierte Generierung einer ASN.1-Enumeration nach C unter Einsatz von Cheetah. Die Template Engine ist turingmächtig, was zielsprachenspezifische Anpassungen ermöglicht. Auszuführende Anweisungen werden durch das Hashzeichen markiert, Kommentaren werden zwei Hashzeichen vorangestellt und Variablen beginnen mit einem Dollarsymbol. In Zeile 1 wird das Python-Modul für reguläre Ausdrücke importiert, welches zusammen mit der Menge der C-Schlüsselworte aus Zeile 3 für die Definition der Funktion* `mkid()` *in Zeile 5 verwendet wird. Diese Funktion dient der Anpassung von Bezeichnern an die Zielsprache. In Zeile 13 beginnt die Codegenerierung: von Zeile 14 bis 22 wird über die Elemente der Enumeration iteriert, wobei abhängig von deren Typ entweder eine Wertzuweisung (Zeile 16) oder ein C-Kommentar (Zeile 18–20) ausgegeben wird.*

```
1  // prepare a list of reserved words
2  val keywords = #{
3          "auto", "break", "case", "char", "const", "continue", "default", "do", "double",
   ↪      "else", "enum", "extern", "float", "for", "goto", "if", "inline", "int", "long", "register",
   ↪      "restrict", "return", "short", "signed", "sizeof", "static", "struct", "switch",
   ↪      "typedef", "union", "unsigned", "void", "volatile", "while"
4  };
5  // define a function that ensures the validity of an identifier
6  def mkid(String str) {
7          var mask = str.replaceAll("[^a-zA-Z0-9]+", "_");
8
9          if (keywords.contains(mask))
10                 mask = mask.toFirstUpper();
11
12         return mask;
13 }
14 // generate the enumeration
15 def generate(Enumeration expr) '''
16         typedef enum «mkid(expr.name)» {
17                 «FOR it: expr.member»
18                         «IF it.univerval»
19                                 «mkid(it.name)»        =        «it.value»,
20                         «ELSEIF it.extensible»
21                                 /*
22                                  * Enumeration is extensible
23                                  */
24                         «ENDIF»
25                 «ENDFOR»
26         } «mkid("e_" + expr.name)»;
27 '''
```

Listing 5: *Beispiel für die sprachbasierte Generierung einer ASN.1-Enumeration nach C in Xtend. Die Sprache unterstützt eingebettete Codeschablonen, in denen Guillemets zur Markierung der dynamischen Anteile verwendet werden. In Zeile 2 wird die Menge der C-Schlüsselworte definiert, die durch die Funktion* `mkid()` *in Zeile 6 verwendet wird, um die Serialisierung von Bezeichnern an die Zielsprache anzupassen. Ab Zeile 15 beginnt die Codegenerierung für die Enumeration mit der Definition der Schablone. Xtend verwendet eine eingeschränkte, funktionale Sprache innerhalb von Schablonen, die den Ruf von Funktionen (und damit das Ausführen beliebigen Xtend-Codes) gestattet. Zwischen Zeile 17 und 25 wird über die Elemente der Enumeration iteriert und abhängig von deren Typ eine Wertzuweisung (Zeile 19) oder ein Kommentar (Zeile 21–23) generiert.*

Compilerfragmentes bereitzustellen ist. Abschnitt 2.6 beinhaltet eine Diskussion über die Verwendung von Metamodellen zur Teilautomatisierung des Transformationsprozesses.

Diskussion

Parr argumentiert in [11], dass die Erweiterung einer Template Engine um turingmächtige Ausdrücke kontraproduktiv sei, da Programmierer in Versuchung gebracht würden, Teile der Ausführungslogik in Schablonen zu verlagern, was Modell und Repräsentation vermischen würde.[3] Grundsätzlich stimme ich dieser Argumentation zu, allerdings kann im Falle der Codegenerierung argumentiert werden, dass die Trennung aufrecht erhalten bleibt, falls das Modell (abstrakter Syntaxbaum und Symboltabelle) *unveränderlich* bereitgestellt wird. Damit wird die Serialisierung in die Domäne der Codegenerierung verlagert, nicht jedoch die Erzeugung oder Optimierung des Modells.

Welcher Ansatz für die Erweiterung der Template Engine geeignet ist, hängt von der konkreten Architektur des Compilerfragmentes ab. Beide benötigen zur Ausführung eine Turingmaschine und reflektieren unterschiedliche Perspektiven. Der erste Ansatz impliziert, dass das Ergebnis eine Zeichenkette ist, die danach weiterverarbeitet werden soll, der zweite betrachtet die Codegenerierung wie eine Rückrufroutine, der an vordefinierten Stellen die Kontrolle übergeben wird. Da die Erweiterung konzeptionell den Platz der Codegen-Schnittstelle aus Abbildung 3 auf Seite 18 einnimmt und unklar ist, wie das Compilerfragment die resultierende Zeichenkette verwenden würde, präferiere ich den zweiten Ansatz. Es ist denkbar, die Architektur so zu gestalten, dass der erste Ansatz als sinnvoller erscheint. Da beide Ansätze eine äquivalente Ausdrucksmächtigkeit besitzen, handelt sich um eine Designentscheidung.

2.6. Metamodelle

Mithilfe einer domänenspezifischen Synthesesprache lässt sich die Codegenerierung kompakt und elegant beschreiben, allerdings muss die interne Repräsentation des Quellcodes zur weiteren Verarbeitung angepasst werden.

3 „Given a Turing-complete template programming language, programmers are tempted to add logic directly where they need it in the template instead of having the data model do the logic and passing in the boolean result, thereby, decoupling the view from the model.“

Diese Anpassung betrifft zwei Aspekte:

1. Falls in der Synthesephase eine andere Sicht auf die Datenstrukturen aus der Analyse gewünscht wird, ist eine Abbildung der einen auf die andere Struktur nötig.

2. Falls Implementations- und Synthesesprache auf unterschiedlichen Plattformen ausgeführt werden, ist der Einsatz einer Kompatibilitätsschicht erforderlich.

Beide Probleme lassen sich prinzipiell durch den Einsatz von Metamodellen behandeln. Ein *Modell* ist im Kontext dieser Arbeit ein symbolisches Abbild eines Aspektes der Wirklichkeit. Ein Objekt ist *konform* zu einem Modell, wenn es die durch das Modell beschriebenen Aspekte einschließlich deren Relationen und Beschränkungen erfüllt. Zu einem Modell konforme Objekte werden als *Instanzen* des Modells bezeichnet. Ein *Metamodell* ist ein Modell, dessen Gegenstand ein Modell ist, also ein Modell der Komponenten eines Modells.

Der metamodellbasierte Ansatz nutzt ein Metamodell zur Beschreibung der Datenstrukturen von Modellen zur Repräsentation der Semantik von Quellcode. Idealerweise verständigen sich Analyse- und Synthesephase auf eine gemeinsame Darstellung durch Konformität der Parameter- bzw. Rückgabetypen zum gleichen Metamodell und laufen außerdem auf der gleichen Plattform. In diesem Fall sind zusätzliche Transformationsschritte überflüssig, andernfalls werden:

- im ersten Fall jeweils ein Metamodell für Quell- und Zielformat sowie eine formale Spezifikation der Transformation,

- im zweiten Fall ein Metamodell mit Serialisierungs- und Deserialisierungsroutinen in die Binärformate der Quell- und Zielsprache benötigt.

Da in der Analysephase für Übersetzer traditionell domänenspezifische Sprachen zum Einsatz kommen und in der Synthesephase ein ähnlicher Ansatz verfolgt wird, kann ein Werkzeughersteller prinzipiell beide Phasen bedienen und so automatisch sicherstellen, dass alle beteiligten Werkzeuge Daten in einem zum gemeinsamen Metamodell konformen Format austauschen.

Ein Beispiel für diesen ganzheitlichen Ansatz ist das Sprachpaar *Xtext* und *Xtend*. Beide Sprachen werden unter dem Schirm der Eclipse-Foundation entwickelt und nutzen die Infrastruktur der Sprache *Java* bei der

gemeinsamen Interaktion sowie das Eclipse Modeling Framework (EMF) für die Definition des gemeinsamen Metamodells. Prinzipiell nimmt *Xtext* die Rolle des Parsergenerators und *Xtend* die Rolle der Synthesesprache ein, wobei *Xtext* unter Einsatz von *ANTLR*[10] *Java*-Code generiert. Im Vergleich zu traditionellen Parsergeneratoren unterstützt *Xtext* keine Annotation der Grammatikregeln mit Aktionscode, stattdessen werden die syntaktischen Informationen aus der Grammatik mit den semantischen aus dem Metamodell durch gemeinsame Bezeichner miteinander assoziiert. Im Metamodell tauchen diese Bezeichner als Attribute bzw. Membervariablen der Datenstrukturen auf, in der Grammatik werden damit Terme in Grammatikregeln ausgezeichnet. Für auf diese Weise markierte Terme generiert *Xtext* Code für die Befüllung der durch das Metamodell vorgegebenen Strukturen, der während des Parsens an entsprechender Stelle ausgeführt wird. Das entspricht einer Kanonisierung der Rolle des Aktionscodes in der traditionellen Compilerentwicklung.

Das Metamodell, einschließlich des abstrakten Syntaxbaumes, kann entweder automatisch aus der annotierten Grammatik abgeleitet oder durch den Entwickler vorgegeben werden. Instanzen des Metamodells lassen sich mithilfe der durch *Xtext* generierten Schnittstellen in Form von abstrakten Basisklassen unter Verwendung von *Xtend* verarbeiten.

Listing 6 und 7 zeigen ein Beispielprojekt für die Übersetzung eines *ASN.1*-Fragmentes, das Enumerationen unterstützt, in *C*-Code. Das Metamodell wurde für dieses Beispiel automatisch durch *Xtext* aus der Grammatik abgeleitet und ist daher nicht mit angegeben. Listing 6 zeigt die Vermischung der Grammatik mit Bezeichnern, die im Metamodell Attributen des abstrakten Syntaxbaumes entsprechen. In Listing 7 wird der auf diese Weise gewonnene Baum für die Codegenerierung unter Verwendung von Codeschablonen abgelaufen und serialisiert.

Der Ansatz bietet folgende Vorteile:

- Das Einführen eines Codegenerators wird auf die Implementation einer abstrakten Codegeneratorklasse abgebildet und unterstützt damit den Einsatz des Entwurfmusters aus Abbildung 3 auf Seite 18.

- Das Wissen über Struktur und Funktion des abstrakten Syntaxbaumes und der Symboltabelle ermöglicht unter anderem die automatische Ableitung fortschrittlicher Editoren und hierarchischer Debugger.

```
1  // Module definition (§13.1)
2  ModuleDefinition:
3      module=TYPEREF "DEFINITIONS" "::=" "BEGIN" body=ModuleBody "END";
4  ModuleBody returns AssignmentList:
5      AssignmentList;
6  AssignmentList:
7      {AssignmentList} assignment+=Assignment*;
8  Assignment:
9      TypeAssignment;
10 TypeAssignment:
11     name=TYPEREF "::=" definition=Type;
12 Type returns EnumeratedType:
13     EnumeratedType;
14
15 // Notation for the enumerated type (§20.1)
16 EnumeratedType:
17     "ENUMERATED" "{" body=Enumerations "}";
18 Enumerations:
19     values+=RootEnumeration ("," values+=ExtensionMarker ("," 
   ↪   values+=AdditionalEnumeration)?)?;
20 RootEnumeration returns Enumeration:
21     Enumeration;
22 AdditionalEnumeration returns Enumeration:
23     Enumeration;
24 ExtensionMarker:
25     {ExtensionMarker} "...";
26 Enumeration:
27     list+=EnumerationItem ("," list+=EnumerationItem)*;
28 EnumerationItem returns NamedNumber:
29     NamedNumber;
30 NamedNumber:
31     name=IDENTIFIER ("(" value=NUMBER ")")?;
32
33 // Whitespace (§3.8.97), Type reference (§12.2), Identifier (§12.3), Number (§12.8)
34 terminal WS: (' '|'\t'|'\r'|'\n')+;
35 terminal TYPEREF: ('A'..'Z') ('a'..'z'|'A'..'Z'|'-'|'0'..'9')*;
36 terminal IDENTIFIER: ('a'..'z') ('a'..'z'|'A'..'Z'|'-'|'0'..'9')*;
37 terminal NUMBER returns ecore::EInt: ('1'..'9')('0'..'9')*;
```

Listing 6: *Xtext-Grammatik für ein Sprachfragment aus X.680 [16] mit Enumerationen. Die Kommentare referenzieren entsprechende Abschnitte. Aus dieser Grammatik erzeugt Xtext sowohl das Metamodell als auch Java-Code zur Initialisierung von dessen Instanzen. Bezeichner auf der linken Seite einer Produktion werden im Metamodell zu Klassen, deren Attribute mittels Zuweisung im Körper referenziert und durch Xtext im erzeugten Aktionscode mit den konkreten Werten aus Quelldateien belegt werden. Terminalsymbole werden durch das Schlüsselwort* `terminal` *eingeleitet und erlauben reguläre Ausdrücke im Produktionskörper.*

```
1  class Asn1Generator implements IGenerator {
2      override doGenerate(Resource resource, IFileSystemAccess fsa) {
3          for (e: resource.allContents.toIterable.filter(TypeAssignment)) {
4              fsa.generateFile(e.name + ".h", generate(e));
5          }
6      }
7      // prepare a list of reserved words
8      val keywords = #{
9          "auto", "break", "case", "char", "const", "continue", "default", "do",
       ↪  "double", "else", "enum", "extern", "float", "for", "goto", "if", "inline", "int",
       ↪  "long", "register", "restrict", "return", "short", "signed", "sizeof", "static",
       ↪  "struct", "switch", "typedef", "union", "unsigned", "void", "volatile", "while"
10     };
11     // define a function that ensures the validity of an identifier
12     def mkid(String str) {
13         var mask = str.replaceAll("[^a-zA-Z0-9]+", "_");
14
15         if (keywords.contains(mask))
16             mask = mask.toFirstUpper();
17
18         return mask;
19     }
20     // generate the enumeration
21     def generate(TypeAssignment expr) '''
22         typedef enum «mkid(expr.name)» {
23             «FOR it: expr.definition.body.values»
24                 «IF it instanceof Enumeration»
25                     «FOR it: (it as Enumeration).list»
26                         «mkid(it.name)» = «it.value»,
27                     «ENDFOR»
28                 «ELSEIF it instanceof ExtensionMarker»
29                     /*
30                      * Enumeration is extensible
31                      */
32                 «ENDIF»
33             «ENDFOR»
34         } «mkid("e_" + expr.name)»;
35     '''
36 }
```

Listing 7: *Codegenerator für die Xtext-Grammatik aus Listing 6 in Xtend. Der Einstiegspunkt befindet sich in der virtuellen Methode `doGenerate()` in Zeile 2, der eine Instanz des Metamodells übergeben wird. Im konkreten Fall erfolgt die Verarbeitung durch Iteration über alle Knoten vom Typ `TypeAssignment` (Zeile 3), um mithilfe der Funktion `generate()` für jede davon eine Header-Datei zu generieren (Zeile 4). Die Funktionen `mkid()` und `generate()` sind analog zu denen aus Listing 5.*

Diskussion

Der Kern dieses Ansatzes ist der Einsatz eines Metamodells zur Beschreibung des abstrakten Syntaxbaumes und der Symboltabelle. Grundsätzlich bildet jede abstrakte Definition von Datenstrukturen ein Metamodell, einschließlich derer in der Implementationssprache (z. B. in Form von *C*-Headerdateien mit Typdefinitionen), allerdings ist deren Syntax nicht standardisiert, weshalb ich diese Form als *implizites* Metamodell bezeichne. Im Vergleich dazu nutzt ein *explizites* Metamodell eine domänenspezifische Beschreibung, die sich ausschließlich auf Strukturaspekte konzentriert.

Es ist denkbar, die Instanzen eines expliziten Metamodells von außen über eine fest definierte Schnittstelle unabhängig von der Plattform erkundbar zu machen, wobei der Übergang zwischen verschiedenen Plattformen und Programmiersprachen durch Codegeneratoren, Serialisierungs- und Deserialisierungsroutinen realisierbar wäre. Damit könnte das Metamodell in heterogenen Umgebungen als Kompatibilitätsschicht dienen. Gegen diesen Ansatz spricht, dass existierende Compiler in der Praxis kein explizites Metamodell einsetzen, weshalb dessen Neuentwicklung notwendig wäre. Dieses müsste mithilfe eines Codegenerators auf die Implementationssprache abgebildet und dann in der Analyse- und Synthesephase eingesetzt werden. Der Aufwand dafür ist vergleichbar mit der Neuentwicklung des Compilers.

3. Glue

3.1. Vorüberlegungen

Die Analyse aus dem vorangegangenen Abschnitt zeigt, dass die Lösung mit dem besten Kosten-Nutzen-Verhältnis für den Compilerentwickler aus der Kategorie der domänenspezifischen Sprachen kommt. Der metamodellbasierte Ansatz erlaubt zwar die Ableitung weiterer Werkzeuge, erfordert allerdings im Gegenzug die Entwicklung eines expliziten Metamodells und greift dadurch in alle Phasen der Compilerentwicklung ein. Im Vergleich dazu lässt sich ein Interpreter unabhängig von anderen Programmkomponenten verwenden.

Für Compiler, die in der Java Virtual Machine ausgeführt werden, lässt sich *Xtend* einsetzen, allerdings gibt es meinen Recherchen zufolge keine äquivalente Variante für die *C*-Sprachfamilie, obwohl viele Übersetzer traditionell Sprachen daraus einsetzen. Die nächstbeste Option ist in diesem Fall eine interpretierte *general-purpose* Sprache mit einer Schnittstelle für die Implementationssprache, ergänzt durch eine Template Engine in Form einer Bibliothek, allerdings sind beim Import der Datenstrukturen statt einer nun zwei Schnittstellen zu konfigurieren. Daraus resultierende Nachteile sind:

- Da die Template Engine als Bibliothek bereitgestellt wird, kann innerhalb von Codeschablonen nicht auf den umliegenden Variablenkontext zugegriffen werden. Variablen müssen daher explizit importiert werden.
- Die Synthesesprache ist möglicherweise nicht optimal auf den Import komplexer Datenstrukturen vorbereitet und stellt nur unhandliche Schnittstellen bereit, die die Rekonstruktion zu importierender Daten mit eingebauten Strukturen erfordern.
- Die Template Engine enthält eine eigene Skriptsprache, die zur Laufzeit interpretiert wird, man verwendet also eine interpretierte Sprache, um zur Laufzeit eine Sprache zu interpretieren. Das kann einen negativen Effekt auf die Laufzeitperformanz der Codegenerierungskomponente haben. Dieser Punkt ist für die Praxis nicht wesentlich und ist nur aus Gründen der Vollständigkeit aufgelistet.

Um diesen Nachteilen vorzubeugen, habe ich eine Skriptsprache mit dem Namen *Glue* entwickelt, die auf die Beschreibung der Synthese von Zielsprachencode abzielt. Der Name „*Glue*“ soll, analog zur Fähigkeit von Klebstoff,

unterschiedliche Komponenten eines Gegenstands miteinander zu verbinden, den Fokus auf die Verbindung der Analyse- mit der Synthesephase eines Übersetzers richten.

3.2. Anforderungen

Das Ziel meiner Synthesesprache ist eine möglichst elegante und einfache Beschreibung des Zielcodes, bei kleinstmöglichem Integrationsaufwand für eine bestehende Compilerarchitektur. Dafür habe ich eine Liste von Anforderungen zusammengestellt:

1. Die Sprache soll eine Template Engine beinhalten, deren Ausgabe automatisch eingerückt wird.

   In manchen Zielsprachen ist die Einrückung von Code ein Bestandteil der Grammatik.[4]

2. Der Import von externen Datenstrukturen soll ausschließlich die Definition derjenigen atomaren Operationen erfordern, die am Ende im Synthesecode verwendet werden. Die Navigation importierter Strukturen soll ohne die Übersetzung aller Elemente in eingebaute *Glue*-Datentypen möglich sein.

   Der abstrakte Syntaxbaum aus der Analysephase des Compilers muss in die Sprache importiert werden. Eine minimalistische Schnittstelle hilft, den Aufwand dafür gering zu halten.

3. Syntaktisches Rauschen ist so weit wie möglich zu vermeiden. Damit ist Code gemeint, der nicht unmittelbar zur Lösung der Aufgabe des Programmes beiträgt. Darunter fallen:

   a) die Auseinandersetzung mit dynamischer Speicherverwaltung,

   b) die Handhabung von Variablentypen,

   c) die Behandlung plattformabhängiger Limitationen,

   d) der Test auf Laufzeitfehler.

   Jede neue Zielsprache erfordert neue Skripte, daher ist es hilfreich, sich in diesen auf die Beschreibung wesentlicher Elemente zu konzentrieren.

4 Ein Beispiel dafür ist die Sprache *Python*, in der die Einrückung von Code zur Blockbildung von Anweisungen verwendet wird.

4. Der Interpreter soll in Form einer freistehenden und plattformunabhängigen Bibliothek ausgeliefert werden.

   Abhängigkeiten in der Codegenerierungskomponente führen zu Einschränkungen in der Plattform potentieller Anwendungen (vgl. Xtend).

5. Die Sprachsyntax soll Minimalität, Orthogonalität und Vertrautheit vor Effizienz, Innovation und Optimierungspotential stellen.

   Eine einfach zu erlernende Sprache hat den Vorteil auch für externe Anwender (etwa die Benutzer des Compilers) verständlich zu sein. Laufzeitperformanz ist kein Primärziel, da sich die berechnungsintensiven Anteile eines Compilers in der Analysephase mit Lexer, Parser und Optimierer befinden, und so relativ viel Spielraum für die Codegenerierung lassen.

3.3. Eingesetzte Werkzeuge

Grundsätzlich bietet sich bei der Neuentwicklung eines Interpreters der metamodellbasierte Ansatz aus Abschnitt 2.6 an, da neben dem Interpreter weitere hilfreiche Werkzeuge wie Debugger und Editor abgeleitet werden können. Dagegen spricht die Bindung an die *Java*-Plattform und die damit einhergehenden Probleme beim Importieren der Datenstrukturen über Plattformgrenzen hinweg. Da ein vergleichbarer Ansatz noch nicht für *C* existiert, habe ich mich bei der Implementation des Interpreters auf traditionelle Werkzeuge zur Sprachentwicklung beschränkt. Verwendet werden:

- der Lexergenerator *flex* in der Version 2.5.39, ein Werkzeug für die Erzeugung von Programmen zur Mustererkennung auf Texten,
- der Parsergenerator *bison* in der Version 3.0.3, ein Werkzeug, das eine annotierte, kontextfreie Grammatik in einen deterministischen LR-Parser unter Verwendung von LALR(1)-Parsertabellen umwandelt,
- der *C*-Compiler *clang* in der Apple-Version 6.0 (basierend auf Version 3.5) und die *C*-Standardbibliothek.

Der Lexergenerator erzeugt Code zur Erkennung der *Glue*-Lexeme, der Parsergenerator erzeugt Code zur Verarbeitung der kontextfreien *Glue*-Grammatik und der *C*-Compiler erzeugt für diese Artefakte gemeinsam mit unterstützendem Code eine statische Bibliothek, die andere Projekte einbinden können. Es bestehen keine weiteren Abhängigkeiten, der Interpreter erfüllt demnach das Kriterium der Unabhängigkeit (4).

3.4. Herausforderungen

In diesem Abschnitt diskutiere ich eine Auswahl wichtiger Probleme, die bei der Implementation des Interpreters für *Glue* eine Rolle gespielt haben. Dazu schildere ich jeweils kurz die Aufgabe, charakterisiere die Lösungsansätze und schließe mit einer Abwägung, die in meiner gewählten Lösungsvariante mündet. Aspekte werden konzeptionell diskutiert und Sprachdetails befinden sich in Anhang A.

3.4.1. Architektur

Die Konstruktion eines Interpreters erlaubt einige Freiheiten bei der Wahl der Architektur, die sich in vier grundlegende Varianten mit unterschiedlichen Merkmalen einteilen lassen. Die Varianten unterscheiden sich in zwei Aspekten: ob ein abstrakter Syntaxbaum konstruiert wird und ob die Ausführung eine virtuelle Maschine benötigt. Eine *virtuelle Maschine* ist ein Programm, das die Ausführung eines echten oder hypothetischen Rechnersystemes simuliert. Die Programmierung einer virtuellen Maschine erfolgt über Maschinencode, der als *Bytecode* bezeichnet wird. Zur Ausführungszeit wird dieser Bytecode durch die virtuelle Maschine interpretiert. Mögliche Architekturformen sind:

1. direkte Ausführung

   Der Quellcode wird beim Parsen schrittweise ausgeführt.

2. direkte Übersetzung in Bytecode

   Der Quellcode wird beim Parsen in Bytecode übersetzt und dann in einer virtuellen Maschine ausgeführt.

3. rekursive Ausführung des abstrakten Syntaxbaumes

   Für den Quellcode wird beim Parsen ein abstrakter Syntaxbaum konstruiert und dieser dann rekursiv ausgeführt.

4. rekursive Übersetzung des abstrakten Syntaxbaumes in Bytecode

   Für den Quellcode wird beim Parsen ein abstrakter Syntaxbaum konstruiert und rekursiv Bytecode erzeugt, welcher dann in einer virtuellen Maschine ausgeführt wird.

Ich habe mich für die dritte Variante, rekursive Ausführung des abstrakten Syntaxbaumes, entschieden und möchte diese Entscheidung nachfolgend diskutieren.

Direkte Ausführung Im augenscheinlich einfachsten Architekturmodell werden Programmanweisungen während des Parsens direkt ausgeführt. Obwohl dieser Ansatz für einfache Sprachen sinnvoll ist und wenig zusätzliche Unterstützung durch Datenstrukturen und Methoden benötigt, ergeben sich Probleme bei Sprunganweisungen, insbesondere bei bedingten Sprüngen und Schleifen, da die Eingabe für den Parser effektiv nicht dem Text in der Quelldatei entspricht. Ein Lösungsansatz für dieses Problem ist die Konstruktion eines rekursiven Interpreters, der sich für bedingte Sprünge auf Teilen der Eingabe selbst ausführt. Daraus entsteht der Nachteil, dass die Eingabe mehrmals eingelesen wird, was negative Auswirkungen auf das Laufzeitverhalten haben kann.

Direkte Übersetzung in Bytecode Die Probleme mit Sprüngen bei direkter Ausführung lassen sich auch durch die Entkopplung von Analyse- und Ausführungseinheit lösen. Eine Möglichkeit dafür besteht in der Generierung von Bytecode für eine virtuelle Maschine mit geeignetem Befehlssatz. Bedingte Sprünge benötigen in diesem Modell keine rekursiven Parserläufe, sondern können durch die virtuelle Maschine mittels Anpassung des Instruktionszeigers behandelt werden. Nachteilhaft sind der erhöhte Aufwand für Entwurf, Implementation und Wartung der virtuellen Maschine sowie die Schwierigkeit guten Bytecode zu erzeugen, da während des Parsens Kontextinformationen fehlen.

Rekursive Ausführung Eine Alternative zum Einsatz einer virtuellen Maschine bietet die Konstruktion eines abstrakten Syntaxbaumes in der Analysephase, gefolgt von dessen rekursiver Ausführung. Sprunganweisungen lassen sich in diesem Modell mittels rekursiver Rufe der Ausführungseinheit behandeln, die den abstrakten Syntaxbaum systematisch abläuft. Auch hier erhöht sich die Komplexität der Architektur um Datenstrukturen für die Konstruktion und Iteration des abstrakten Syntaxbaumes. Problematisch ist weiterhin die direkte Kopplung zwischen Programmanweisung und Knoten im Syntaxbaum, was Spracherweiterungen nur simultan mit der Einführung neuer Knotenarten ermöglicht. Beim Einsatz einer virtuellen Maschine besteht dieses Problem nicht, da sich neue Anweisungen mithilfe des bereits bestehenden Befehlssatzes ausdrücken lassen, allerdings ist der Aufwand für Entwurf und Implementation höher.

Rekursive Übersetzung in Bytecode Die Kombination der beiden vorigen Ansätze liefert eine Lösung, in der der abstrakte Syntaxbaum in der

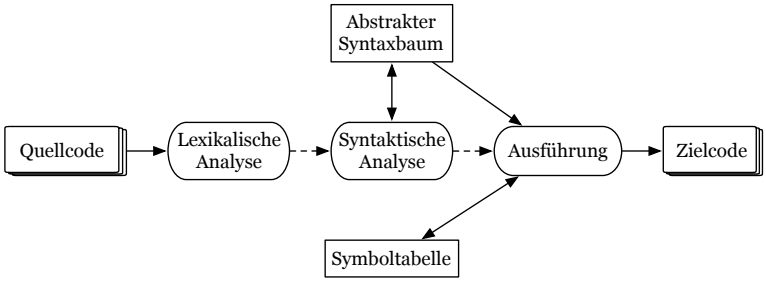


Abbildung 4: *Programm- und Datenfluss der relevanten Phasen und Strukturen bei der Abarbeitung von Quellcode durch den Glue-Interpreter. Die Semantik der Markierung entspricht der aus Abbildung 1 auf Seite 10. Quellcode wird zunächst während der lexikalischen Analyse in Lexeme zerlegt, mit denen dann in der syntaktischen Analyse unter Einsatz der Grammatik ein abstrakter Syntaxbaum konstruiert wird. Dieser dient in der Ausführungsphase als Eingabe und wird rekursiv abgearbeitet, wobei Variablennamen über die Symboltabelle mit Werten assoziiert werden. Die Ausgabe bei der Verarbeitung des Quellcodes entspricht dem Zielcode.*

Analysephase konstruiert, dann in einer zusätzlichen Codegenerierungsphase in Bytecode überführt und schließlich in der virtuellen Maschine ausgeführt wird. Sprunganweisungen lassen sich durch Anpassung des Instruktionszeigers behandeln und im Vergleich zur direkten Übersetzung in Bytecode enthält der abstrakte Syntaxbaum während der Codesynthese den genauen Programmkontext (was registerbasierte Befehlssätze ermöglicht). Die Komplexität der Architektur erhöht sich sowohl um Datenstrukturen und Methoden für den abstrakten Syntaxbaum, als auch um solche für die virtuelle Maschine, was die Lösung zur aufwändigsten macht. Auf der anderen Seite resultiert das Wissen über den genauen Programmkontext in der Synthesephase in größeren Freiheiten beim Entwurf des Befehlssatzes und entkoppelt Syntax und Interpretation, was Spracherweiterungen ohne Anpassung der virtuellen Maschine zulässt.

Resultierende Architektur

Die rekursive Ausführung des Syntaxbaumes ergibt im Rahmen dieser Arbeit den besten Kompromiss bezüglich Aufwand für Entwurf und Implementation auf der einen, Wartung und Erweiterung auf der anderen Seite. Abbildung 4 zeigt die resultierende Architektur für den *Glue*-Interpreter. Für die lexikalische und syntaktische Analyse werden *flex* und *bison* verwendet, wobei in *bison* der abstrakte Syntaxbaum konstruiert und partiell

optimiert wird. Die Optimierung beschränkt sich dabei auf die Faltung von Zeichenketten-, Zahl-, Tupel- und Listenkonstanten sowie die statische Auswertung konstanter Bedingungen. Die semantische Analyse findet zur Laufzeit statt, weshalb nur dort Zugriff auf die Symboltabelle benötigt wird. Die Symboltabelle selbst ist konzeptionell ein Stapel von Abbildungen (*Stack of Maps*), mit der Sichtbarkeitsbereiche wie in *C* geschichtet werden können und die während der Programmabarbeitung die aktuelle Sicht auf Variablen widerspiegelt.

3.4.2. Schnittstellen

In der Abarbeitungsphase manipuliert der Interpreter Daten über eine abstrakte Schnittstelle. Abbildung 5 zeigt ein *UML*-Klassendiagramm, das neben dieser Schnittstelle die Basisklassen der Objekte und Variablentypen darstellt. Im folgenden Abschnitt diskutiere ich anhand dieser Abbildung das Objekt- und Typsystem in *Glue* und erläutere, welche Schritte notwendig sind, um neue Variablentypen einzuführen. Der Compilerentwickler verwendet die Schnittstellen, um dem Zielsprachenentwickler den abstrakten Syntaxbaum zur Verfügung zu stellen.

GlueObject
Alle durch den Interpreter manipulierbaren Objekte sind *GlueObject*-Instanzen[5] und bieten über das Attribut `vptr` Zugriff auf die in *GlueTypeInfo* hinterlegten Typinformationen[6]. Das Attribut `ref` wird vom Interpreter zur Realisierung einer referenzbasierten, dynamischen Speicherverwaltung verwendet: jede Referenz auf ein *GlueObject* erhöht den Referenzzähler und jede Variable, die aus ihrem Sichtbarkeitsbereich läuft, verringert diesen. *Glue*-Objekte werden freigegeben, wenn keine weiteren Referenzen auf sie verweisen. Um zirkuläre Strukturen zu vermeiden, werden schwache Referenzen (*weak pointer*) eingesetzt. *Glue*-Objekte sind üblicherweise nicht die Besitzer des *GlueTypeInfo*-Objektes, auf das ihr `vptr` verweist, sondern teilen sich den schreibgeschützten Zugriff darauf mit anderen, gleichartigen Objekten. Der Besitzer eines *GlueTypeInfo*-Objektes nimmt die Rolle des Datentyps der Objekte, die darauf verweisen, ein.

5 Da der Interpreter in *C* geschrieben ist, entspricht das erste Attribut einer abgeleiteten Klasse in der Praxis einer Instanz der Basisklasse, so dass Zeiger auf Instanzen der einen auch Zeiger auf Instanzen der anderen Klasse sind.

6 Das schließt in der *C*-Implementation Zeiger auf alle als virtuell markierten Methoden ein. Die Rolle von *GlueTypeInfo* ist vergleichbar mit der von `std::type_info` aus der *C++*-Standardbibliothek.

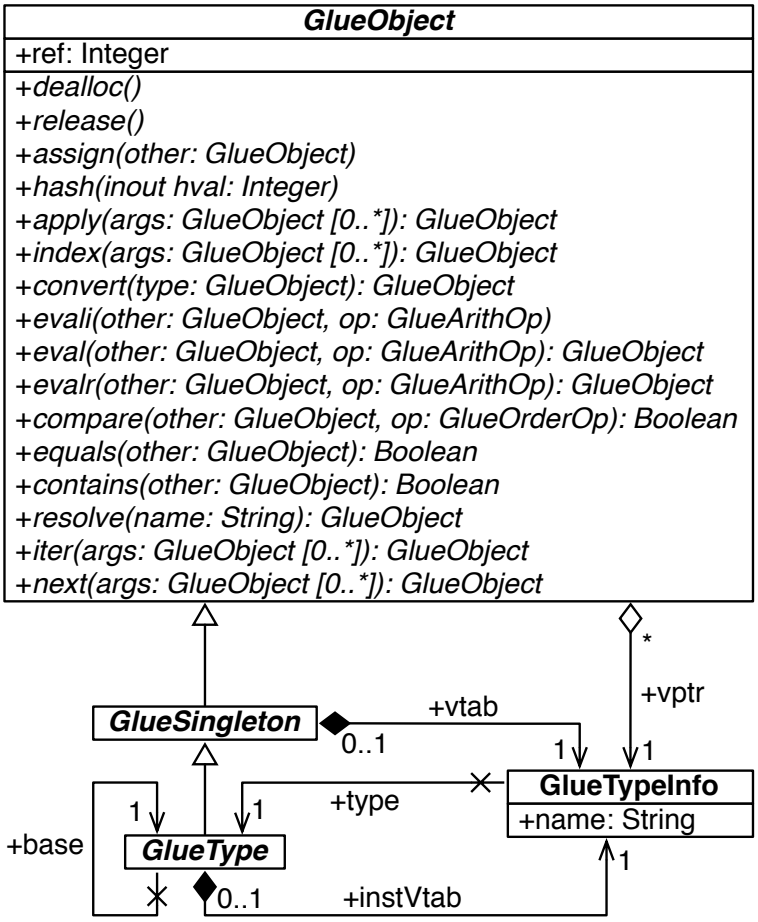


Abbildung 5: *UML-Klassendiagramm für durch den Glue-Interpreter manipulierbare Objekte und damit verbundene Klassen. Abgebildet sind die Klasse* GlueObject*, die als Basisklasse aller Glue-Objekte fungiert, die abgeleitete Klasse* GlueSingleton *sowie deren Ableitung* GlueType*. Instanzen der Klasse* GlueObject *besitzen einen Referenzzähler* `ref` *und verweisen über das Attribut* `vptr` *auf ein* GlueTypeInfo*-Objekt mit Informationen bezüglich ihres Glue-Typs. Die Klasse* GlueSingleton *besitzt mit* `vtab` *ihr eigenes* GlueTypeInfo*-Objekt und dient als Basisklasse für Glue-Typen mit nur einer Instanz. Davon abgeleitet ist die Klasse* GlueType*, welche als Basisklasse der Variablentypen in Glue dient. Sie beinhaltet mit* `instVtab` *das* GlueTypeInfo*-Objekt der von ihr produzierten Instanzen sowie eine Referenz* `base` *auf ihre Glue-Basisklasse. Die Attribute der Klasse* GlueTypeInfo *sind der Name* `name` *des Glue-Typs und eine Referenz* `type` *auf das entsprechende* GlueType*-Objekt.*

Da abstrakte Klassen nicht durch den *C*-Compiler unterstützt werden, sind die virtuellen Methoden als Funktionszeiger in der Klasse *GlueTypeInfo* hinterlegt und werden in dessen Abschnitt diskutiert.

GlueSingleton
Glue-Typen mit nur einer Instanz, sogenannte *Singletons*, werden als *GlueSingleton*-Objekte modelliert. Sie sind über das Attribut `vtab` Besitzer ihres eigenen *GlueTypeInfo*-Objektes und können über diese Verbindung ihre Schnittstelle frei definieren. Ein wichtiger Vertreter dieser Kategorie ist die Menge der Datentypen.

GlueType
Konzeptionell werden Instanzen eines Datentyps in *Glue* durch das Objekt, dem die entsprechende *GlueTypeInfo*-Schnittstelle gehört, konstruiert – *Glue*-Typen sind also Fabriken, die Instanzen produzieren. Da es von jedem *Glue*-Typ nur eine Instanz geben sollte, ist die Klasse *GlueType* eine Ableitung von *GlueSingleton*. Sie enthält das *GlueTypeInfo*-Objekt, auf das das `vptr`-Attribut der von ihr konstruierten Instanzen zeigt sowie eine Referenz auf ihre Superklasse, die vom Interpreter zur Unterstützung eines einfachen Vererbungsschemas verwendet wird.

GlueTypeInfo
Die Klasse *GlueTypeInfo* kapselt die Schnittstelle für *Glue*-Objekte, welche darüber ihr Verhalten definieren. Sie beinhaltet den Namen und eine Referenz auf den *Glue*-Typ sowie Funktionszeiger, die der Interpreter in der Abarbeitungsphase rufen kann. Unterstützte Methoden fallen in eine von zwei Kategorien – Speichermanagement und Operatoren – die ich im folgenden kurz vorstelle.

Speichermanagement In diese Kategorie fallen `dealloc()` zur Speicherfreigabe und `release()` zur Finalisierung der Attribute einer Instanz. Ohne diese Unterscheidung wäre die Möglichkeit zur Komposition von *Glue*-Objekten eingeschränkt, da die Speicherfreigabe der Instanz nicht auf ein übergeordnetes Objekt delegierbar wäre.

Operatoren Diese Kategorie beinhaltet die Definition aller durch *Glue* unterstützten Operatoren. Das sind: Zuweisungen (via `assign()`), die Berechnung eines Hashwertes (via `hash()`), den Ruf wie bei einer Funktion (via `apply()`), die Indizierung wie bei einem Feld (via `index()`), die Konvertierung in einen anderen Datentyp (via `convert()`), die Auswertung

eines arithmetischen Operators (via `evali()`, `eval()` und `evalr()`), den Vergleich (via `compare()`), den Test auf Gleichheit (via `equals()`), den Test auf Mitgliedschaft in einer Kollektion (via `contains()`), die Rückgabe eines Iterators (via `iter()`) sowie, im Falle eines Iterators, die Wahl des nächsten Elementes im Container (via `next()`). Die arithmetischen Operatoren sind mit `evali()`, `eval()` und `evalr()` in verschiedenen Varianten vertreten: `eval()` ist die Basisform, die alle unären und binären Operatoren implementiert und ein neues *GlueObject* als Ergebnis zurückgibt. Im Vergleich dazu modifiziert `evali()` das gerufene Objekt direkt. Die Methode `evalr()` wird ausschließlich für binäre Operatoren am zweiten Parameter mit vertauschten Operanden gerufen und erlaubt deren Überladung von der rechten Seite. Die Methode `resolve()` löst Attributszugriffe auf und macht damit innere Strukturen navigierbar.

Neudefinition von Variablentypen

Jeder Compiler, der *Glue* als Codegenerierungskomponente einsetzt, muss zumindest die Datenstruktur des abstrakten Syntaxbaumes aus der Analysephase importieren. Dabei müssen grundsätzlich nur die Teile der Schnittstelle aus *GlueTypeInfo* implementiert werden, die am Ende im Skript unterstützt werden sollen – für alle undefinierten Methoden, die verwendet werden, kommt es zu einem Laufzeitfehler, falls deren Funktion nicht durch andere Methoden simuliert werden kann.

Konzeptionell müssen *Glue*-Typen die `apply()`-Methode implementieren, um neue Objekte zu instanzieren. Diese Methode ist für die Bereitstellung von Speicher und die Initialisierung der neukonstruierten Instanz zuständig. Danach kann der *Glue*-Typ über die Symboltabelle mit einem Namen assoziiert werden und ist durch den Benutzer der Skriptsprache unter diesem Namen referenzierbar.

3.4.3. Syntax

Die Notation legt den Rahmen der Interaktion zwischen Entwickler und Interpreter fest und spielt daher eine wichtige Rolle bei die Umsetzung von Anforderung 5 auf Seite 37. Um das Erlernen der Sprache, die Formulierung und das Lesen von Programmcode zu unterstützen, habe ich mich beim Entwurf der Syntax an drei Prinzipien orientiert: *Minimalität* in der Wahl der Ausdrucksmöglichkeiten, *Orthogonalität* bei der Unterstützung von Konzepten und *Vertrautheit* der Notation. Das Ergebnis ist eine imperative Sprache, die sich syntaktisch an der Sprache *C* orientiert, ohne

Konzept	Anmerkung
Ganze Zahlen	immer vorzeichenbehaftet
Zeichenketten	Strings und String Templates
Kollektionen	Listen und Tupel als Literale
unäre Operatoren	wie in *C*, keine Dereferenzierung
binäre Operatoren	wie in *C*, zusätzlich Potenzierung
ternäre Operatoren	Semantik wie in *C*, syntaktische Differenzen
Vergleiche	wie in *C*, zusätzlich Test auf Mitgliedschaft
Zuweisungen	wie in *C*, zusätzlich Potenzierung
Variablendeklarationen	Typinferenz, Referenz- und Wertesemantik
Funktionsdeklarationen	parametrisierte Variablen, keine Überladung
bedingte Sprünge	*if* wie in *C*, kein *switch-case*
unbedingte Sprünge	*break* und *continue*, keine Label, kein *goto*
Schleifen	*while* und *do-while* wie in *C*, *for-each* wie in *Java*

Tabelle 3: *Überblick über durch Glue unterstützte Konzepte.*

explizite Typdeklarationen für Variablen auskommt und eine Template Engine integriert. Tabelle 3 gibt einen kurzen Überblick über die unterstützten Konzepte, Anhang A enthält eine vollständige Syntaxspezifikation.

Kommentare
Glue-Kommentare verwenden die Kommentarsyntax von *C*++.

Literale
Neben den Zahlen- und Stringliteralen wie in *C*, erlaubt *Glue* die direkte Notation von Listen in Form einer kommaseparierten Variablensequenz zwischen eckigen Klammern. Ebenfalls unterstützt wird die Notation von Tupeln – Listen fixer Länge – als kommaseparierte Sequenz zwischen runden Klammern. Nicht unterstützt werden Fließkommazahlen, da ihre Einführung Probleme mit der Präzision auf unterschiedlichen Plattformen mit sich bringen würde. Stringliterale können in Hochkommas oder Anführungszeichen eingeschlossen werden und unterstützen die benannten Escapesequenzen wie in *C*, nicht jedoch in der Oktal- oder Hexadezimalschreibweise (`'\nnn'` oder `'\xnn'`), die den ASCII-Code direkt angibt. Der Grund besteht im Anspruch auf Plattformunabhängigkeit der Sprache, aufgrund derer die konkrete Kodierung von Zeichenketten vom Benutzer verborgen ist.

Ausdrücke
Neben den bekannten unären und binären Operatoren aus *C* unterstützt *Glue* zusätzlich die Potenzierung mit der Notation `**`. Der ternäre *C*-Operator (`<Bedingung> ? <Dann> : <Ansonsten>`) folgt mit (`<Dann> if <Bedingung> else <Ansonsten>`) einer leicht veränderten Syntax, hat aber eine äquivalente Semantik: `<Dann>` wird genau dann ausgewertet, wenn `<Bedingung>` zu `true` evaluiert, andernfalls wird `<Ansonsten>` ausgewertet. Der Grund für diese Änderung besteht darin, dass der Doppelpunkt bereits als binärer Operator zum Test auf Mitgliedschaft eingesetzt wird und deshalb an dieser Stelle mehrdeutig wäre. Der Dereferenzierungsoperator `*` aus *C* wird in *Glue* sowohl zum Entpacken von Kollektionen verwendet als auch zur Markierung variabler Argumentlisten in Funktionssignaturen.

Zusätzlich zu den aus *C* bekannten Vergleichsoperatoren (`>`, `<`, `>=`, `<=`, `==` und `!=`) unterstützt *Glue* die Notation von Tests auf Mitgliedschaft in Kollektionen via `:` und `!:`. Vergleiche werden notiert wie in *C*, unterscheiden sich allerdings in der Semantik bei Vergleichsketten wie etwa `(a < i < b)`. Diese Bedingung ist in *Glue* genau dann erfüllt, wenn `(a < i)` und `(i < b)` ist, also wenn i im offenen Intervall zwischen a und b liegt. In *C* ist diese Kette genau dann wahr, wenn `(a < i)` zu einem Wert evaluiert, deren (implizite) Konvertierung in eine Zahl kleiner als `b` ist. Die *Glue*-Semantik ermöglicht eine intuitive Formulierung von Tests auf Mitgliedschaft in Intervallen.

Anweisungen
Aus *C* übernommen wurden die *while* und *do-while*-Schleifen, die *if*-Anweisung sowie die *for-each*-Schleife aus *Java*. Das Verhalten der *for*-Schleife aus *C* kann mit einer *while*-Schleife problemlos repliziert werden und ist aus Minimalitätsgründen deshalb nicht vertreten. Die gleiche Begründung gilt für die fehlende Unterstützung der *switch-case*-Fallunterscheidung, die sich aus geschachtelten *if-else-if*-Anweisungen reproduzieren lässt.

Variablendeklarationen wird das Schlüsselwort `new` vorangestellt und erfordern eine sofortige Zuweisung, die zur Ableitung ihres Datentyps verwendet wird. Zuweisungen lassen sich optional mit allen binären Operatoren wie in *C* augmentieren.

Template Engine

Ein Auszeichnungsmerkmal der Sprache ist die eingebaute Template Engine. Prinzipiell lassen sich Zeichenketten und Variablen auch mit den

```
1  $(
2      <body>
3          <h1>$title</h1>
4          $for (page: pages)
5              <h2>${page.title}</h2>
6              <ol>
7                  $for (element: page.elements)
8                      <li>$element</li>
9                  $$
10             </ol>
11         $$
12     </body>
13 )
```

Listing 8: *Beispiel für ein String Template in Glue. Die Schablone ist geklammert mit* `$(` *in Zeile 1 und* `)` *in Zeile 13. In Zeile 3 wird die Variable* `title` *aus dem umliegenden Sichtbarkeitsbereich in den Ausgabestrom eingefügt. Die Zeilen 5 bis 10 werden für alle Elemente der Kollektion* `pages` *wiederholt, ebenso Zeile 8 für die Elemente aus* `page.elements`*. In Zeile 5 wird* `page.title` *ausgegeben. Die grau markierten Anteile werden automatisch entfernt, um die korrekte Einrückung zu erhalten. Ebenfalls entfernt werden alle Zeilen mit Kontrollstrukturen.*

gewöhnlichen Stringkonkatenationsoperatoren kombinieren, allerdings bietet die Notation als String Template einige Vorteile:

1. Die Vermischung von Zeichenketten und Variablen erfolgt ohne explizite Konkatenation und Konvertierung.

2. Es sind keine Escapesequenzen für Einrückung und Zeilenende nötig.

3. Die Einrückung der Ausgabe wird aus der Schablone abgeleitet.

Listing 8 zeigt ein Beispiel für ein String Template in *Glue*. Schablonen sind in `$(` und `)` oder `$[` und `]` eingeschlossen, wobei die dynamischen Anteile durch ein vorangestelltes Dollarsymbol ausgezeichnet und speziell verarbeitet werden:

- ▷ Bezeichner werden aufgelöst, in eine Zeichenkette konvertiert, eingerückt und in den Ausgabestrom eingefügt,

- ▷ Unterschablonen lassen sich mit `$()` und `$[]` einbinden und können beliebig geschachtelt werden,

- ▷ Codeblöcke können mit geschweiften Klammern geöffnet werden und erlauben sowohl Ausdrücke als auch Anweisungen; für letztere wird das Resultat der `return`-Anweisung an entsprechender Stelle eingefügt,

- ▷ Kontrollstrukturen (`$if`, `$for`, `$while` und `$else`)[7] lassen sich *inline* notieren und werden mit `$$` abgeschlossen.

Alle aktiven Sonderzeichen (`$`, `(`, `)`, `[`, `]` und gegebenenfalls `\`) können mit `\` maskiert werden. Balancierte Klammerpaare müssen nicht maskiert werden. Die beiden Klammerarten bei der Markierung der Schablonengrenzen haben unterschiedliche Voreinstellungen bezüglich der Behandlung der Sonderzeichen:

1. eckige Klammern müssen im Schablonenkontext von `$()` nicht ausbalanciert sein, ebenso runde Klammern in dem von `$[]`,

2. das Dollarsymbol wird im Schablonenkontext von `$[]` nicht interpretiert; die Maskierung *aktiviert* dort dessen Semantik.

Automatische Einrückung
Die *Einrückung* einer Zeile entspricht der Sequenz von nicht-sichtbaren Zeichen an deren Anfang. Die Template Engine in *Glue* unterstützt die sprachagnostische Einrückung mittels Extrapolation. Dafür wird der statische Text in Schablonen unter Anwendung der folgenden Regeln vorbehandelt:

1. Unmittelbar bei Betreten einer Schablone werden alle Leerzeilen entfernt und die Einrückung der ganzen Schablone um die Einrückung der ersten, nicht-leeren Zeile verringert.

2. Die Einrückung von Unterschablonen wird um die angepasste Einrückung der Startzeile in der umliegenden Schablone erhöht.

3. Zeilen mit Kontrollstrukturen (`$for`, `$if`, etc.) werden entfernt, falls die Zeile ansonsten keine sichtbaren Zeichen beinhaltet. Kontrollstrukturen produzieren bei der Ausführung selbst keine Ausgabe.

4. Der Körper von Kontrollstrukturen wird wie eine Unterschablone behandelt.

7 Die *do-while*-Schleife wird für String Templates nicht unterstützt, da Schleifen schachtelbar sind und das *while*-Ende syntaktisch ununterscheidbar vom Kopf der *while*-Schleife ist.

Die Einrückung der Expansion von Codeblöcken wird um die Einrückung der entsprechenden Zeile erhöht, falls ein Rückgabewert vorliegt, ansonsten wird die Zeile gelöscht, wenn sie andernfalls leer wäre. Das ist die Laufzeitvariante der Regeln (2) und (3).

3.4.4. Arithmetik

Ganzzahlarithmetik ist ein potentielles Hindernis beim Schreiben plattformunabhängiger Programme: die Wortbreite der Zahlentypen ist möglicherweise nicht standardisiert (oder nur mit Minimalanforderungen belegt), die Darstellung negativer Zahlen kann zwischen unterschiedlichen Plattformen variieren und als Korollar das Verhalten bei Überläufen undefiniert sein. Damit ist es prinzipiell möglich, mithilfe einer Codegenerierungskomponente Zielcode abzuleiten, der sich von Plattform zu Plattform unterscheidet, mal semantisch korrekt ist, mal Laufzeitfehler erzeugt und mal in legalen, inkorrekten Programmen mündet. Schlimmer noch: durch die Undefiniertheit von Überläufen ist der Test auf einen Überlauf *nach* dem Fakt ebenfalls undefiniert – eine schwere Bürde für den Zielsprachenentwickler.

Der aktuelle *C*-Standard [7] schreibt in § 6.2.6.2-2 minimale Wortbreiten für eingebaute Zahlentypen vor und fordert für die Darstellung negativer Zahlen eine von drei möglichen Formen: Negation, Einerkomplement oder Zweierkomplement. Für den Überlauf vorzeichenbehafteter Zahlen ist das Verhalten durch § 6.5-5 explizit *undefiniert* – auf einer Stufe mit der Dereferenzierung des Null-Zeigers. Alle geschilderten Probleme treffen also auf *Glue* zu. Um diesem Problem zu begegnen, kann einer der folgenden Wege beschritten werden:

1. Ganzzahlen können prinzipiell mit unendlicher Präzision ausgestattet werden, was Überläufe ausschließt. Dieser Ansatz resultiert in sogenannter *Langzahlarithmetik*, bei der die maximale Breite einer Zahl nur durch den verfügbaren Arbeitsspeicher beschränkt ist.

2. Ganzzahlen können, statt überzulaufen, ihren maximalen bzw. minimalen Wert annehmen. Dieser Ansatz resultiert in sogenannter *Sättigungsarithmetik*, in der die Wortbreite zwar durch die Plattform diktiert wird, Überläufe jedoch nicht in undefiniertem Verhalten münden und durch den Benutzer getestet werden können.

Beide Ansätze haben Vor- und Nachteile:

- Langzahlarithmetik produziert auf jeder Plattform mit ausreichend viel Arbeitsspeicher die gleichen Resultate und andernfalls einen

Laufzeitfehler, hat allerdings höhere Anforderungen an die Implementation und ist weniger effizient in der Ausführung.

- ▷ Sättigungsarithmetik produziert konsistentes, wohldefiniertes Verhalten, ist mit relativ wenig Aufwand zu implementieren und bis auf eine Konstante so schnell wie naïve Arithmetik, erfordert allerdings manuelle Tests durch den Anwender, um korrekten Zielcode zu garantieren.

Mit Blick auf den Umfang dieser Arbeit habe ich mich für die Umsetzung der zweiten Variante, Sättigungsarithmetik, entschieden. Arithmetische Operationen, die überlaufen, produzieren den größten bzw. kleinsten Wert. Dieser muss auf allen Plattformen, unabhängig von der tatsächlichen Wortbreite, gleich sein. Da meine Motivation die Konstruktion eines *ASN.1*-Compilers und der Ganzzahltyp dort unbegrenzt ist, habe ich die Domäne des *Glue*-Ganzzahlentyps um drei Spezialwerte erweitert:

1. Die Zahl *unendlich* (in *Glue* `inf`) ist größer als alle anderen Zahlen, sich selbst ausgenommen.
2. Die Zahl *minus-unendlich* (in *Glue* `-inf`) ist kleiner als alle anderen Zahlen, sich selbst ausgenommen.
3. Die Zahl *NaN* (*Not a Number*, in *Glue* `nan`) ist ungeordnet und ungleich zu sich selbst.

Die größte Zahl *unendlich* ist das Ergebnis eines positiven Überlaufes, die kleinste Zahl *minus-unendlich* das eines negativen Überlaufes. *Glue* unterstützt eine eingeschränkte Form der Arithmetik mit diesen Werten, wobei sie stellvertretend für eine beliebige, bestimmt-divergierende Folge mit dem Grenzwert $\pm\infty$ stehen. Falls das Ergebnis einer arithmetischen Operation undefiniert oder abhängig von der gewählten Folge ist, wird als Resultat *NaN* produziert, etwa bei der Division zweier Unendlichkeiten. Arithmetik unter Beteiligung von *NaN* produziert weitere *NaN*, falls das Ergebnis von diesem Wert abhängt. Auf diese Weise werden Fehler in Berechnungsbäumen zur Wurzel propagiert.

Der Nutzer kann auf diese Werte testen, wenn das Ergebnis einer Rechnung in Zielcode übertragen werden soll.

3.5. Beispiel

In diesem Abschnitt wird *Glue* beispielhaft zur Generierung von *C*-Strukturen für ein *ASN.1*-Sprachfragment verwendet, das die *ASN.1* Sequenz-, Listen-

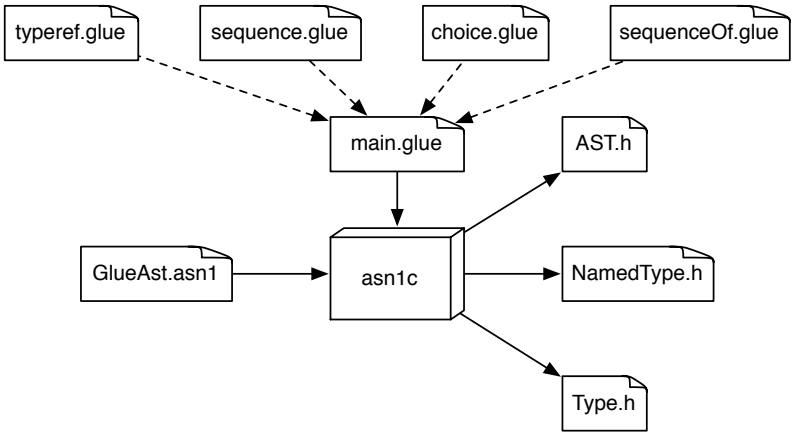


Abbildung 6: *Abarbeitungsdiagramm des erweiterten ASN.1-Beispieles. Der ASN.1-Compiler* `asn1c` *ist als Box, Dateien sind als Rechtecke mit gefalteter Ecke dargestellt. Pfeile auf eine Box zeigen Lesezugriff, Pfeile ausgehend davon Schreibzugriff an. Gestrichelte Pfeile zwischen zwei Dateien zeigen die Referenzierung der Datei mit dem ausgehenden Pfeil durch die Datei mit dem eingehenden Pfeil an.*

und Auswahltypen unterstützt. Die erhöhte Ausdrucksmächtigkeit erlaubt nun die rekursive Schachtelung, Einbettung und Referenzierung von Datentypen, was das Beispiel aus Listing 9 im Vergleich mit Listing 1 auf Seite 16 aufgrund erhöhter Komplexität aussagekräftiger macht.

3.5.1. Analyse

Die *ASN.1*-Spezifikation aus Listing 9 wird mithilfe einer angepassten Version des *ASN.1*-Compilers `asn1c` von Walkins [13] eingelesen und verarbeitet. Die Codegenerierungskomponente des ursprünglichen `asn1c` wurde dafür durch eine Komponente mit einem *Glue*-Interpreter ersetzt, den diese zur Interpretation eines *Glue*-Skriptes an einem vordefinierten Dateipfad verwendet. Dem dort hinterlegten Programm wird die Instanz des abstrakten Syntaxbaumes der konkreten Spezifikation in Form der globalen Variablen `asn1` zugänglich gemacht. Die Beispielspezifikation fungiert dabei sowohl als Beispieleingabe als auch als Strukturdefinition der Variablen `asn1`, wobei der Blick auf die interne Struktur des abstrakten Syntaxbaumes mithilfe der in Abschnitt 3.4.2 diskutierten Schnittstellen für den Interpreter aufbereitet wurde. Abbildung 6 enthält eine Darstellung aller beteiligten Dateien und Programme und Listing 10 enthält den abstrakten Syntaxbaum der Spezifikation in *ASN.1*-Wertenotation.

```
1  GlueAst DEFINITIONS AUTOMATIC TAGS ::= BEGIN
2
3      AST ::= SEQUENCE OF SEQUENCE {
4          module VisibleString,
5          member SEQUENCE OF NamedType
6      }
7
8      NamedType ::= SEQUENCE {
9          name VisibleString,
10         type Type
11     }
12
13     Type ::= CHOICE {
14         typeref VisibleString,
15         choice SEQUENCE OF NamedType,
16         sequence SEQUENCE OF NamedType,
17         sequenceOf Type
18     }
19
20 END
```

Listing 9: *Erweitertes ASN.1-Beispiel unter Einsatz von Sequenz-, Listen- und Auswahltypen. Die Spezifikation definiert das Modul* GlueAst *mit den drei benannten Typen* AST*,* NamedType *und* Type*.* AST *ist der Datentyp eines abstrakten Syntaxbaumes und enthält eine Liste von Modulen, die jeweils einen Namen (`module`, Zeile 4) und eine Liste benannter Typen (`member`, Zeile 5) haben. Benannte Typen sind Tupel von Namen (`name`, Zeile 9) und anonymen Typen (`type`, Zeile 10) und werden durch* NamedType *repräsentiert. Anonyme Typen sind die als* Type *bezeichnete Auswahl zwischen Typreferenz (`typeref`, Zeile 14), Auswahltyp (`choice`, Zeile 15), Sequenztyp (`sequence`, Zeile 16) und Listentyp (`sequenceOf`, Zeile 17).*

```
asn1 AST ::= {
    { module "GlueAst", member {
        { name "AST", type sequenceOf: sequence: {
            { name "module", type typeref: "VisibleString" },
            { name "member", type sequenceOf: typeref: "NamedType" }
        }},
        { name "NamedType", type sequence: {
            { name "name", type typeref: "VisibleString" },
            { name "type", type typeref: "Type" }
        }},
        { name "Type", type choice: {
            { name "typeref", type typeref: "VisibleString" },
            { name "choice", type sequenceOf: typeref: "NamedType" },
            { name "sequence", type sequenceOf: typeref: "NamedType" },
            { name "sequenceOf", type typeref: "Type" }
        }}
    }}
}
```

Listing 10: *Abstrakter Syntaxbaum der Spezifikation aus Listing 9 in ASN.1-Wertenotation. Die Struktur ist im Format der Spezifikation, die sie kodiert.*

Die konkrete Wahl des Auswahltyps wird als Zeichenkette im Attribut `which` gespeichert und der Inhalt ist jeweils im Attribut dieses Namens hinterlegt. Um in dieser Struktur etwa das Attribut *sequence* aus dem Datentyp *Type* auszuwählen, muss in *Glue* `asn1[0].member[2].type.choice[2]` aufgelöst werden. Das Attribut `choice` existiert in `asn1[0].member[2].type`, da `asn1[0].member[2].type.which` die Zeichenkette „choice“ enthält.

Folgende Probleme treten bei der Implementation der Codesynthese auf:

1. Die Möglichkeit der direkten Einbettung von Strukturen erfordert die Unterscheidung zwischen benannten und unbenannten Typen.

2. Strukturen, welche andere referenzieren, müssen diese deklarieren.

3. Strukturen, die sich selbst direkt oder indirekt referenzieren, müssen diese Situation erkennen und sicherstellen, dass sie sich nicht selbst enthalten.

4. Unzulässige Bezeichner müssen durch zulässige ersetzt werden.

5. Auswahl-, Sequenz- und Listentypen sind keine nativen Typen in *C* und müssen daher als Kombination solcher dargestellt werden.

3.5.2. Lösung

Die prinzipielle Vorgehensweise bei der Generierung des Codes besteht im systematischen Ablauf des abstrakten Syntaxbaumes, wobei für jeden benannten Typ eine *C*-Headerdatei generiert wird. Dabei werden die angesprochenen Probleme wie folgt gelöst:

1. Da sich in *ASN.1* alle benannten Typen auf Modulebene befinden, wird die Struktur in diesem Fall mit einem Namen versehen, andernfalls ist sie anonym.

2. Das Programm führt für jeden benannten Typ Buch über die Menge aller enthaltenen Teilstrukturen, die für die Erzeugung von `#include`-Direktiven verwendet wird.

3. Um den Fall der Selbstinklusion zu testen, werden den benannten Typen alle bereits berechneten Mengen enthaltener Teilstrukturen zugeordnet und, im Falle der Referenzierung, deren reflexiv-transitive Enthaltenbarkeitshülle nach dem Supertyp durchsucht. Ist dieser enthalten, darf er den entsprechenden Typ nicht direkt referenzieren, sondern muss ihn vorwärts deklarieren und über einen Zeiger einbinden. Es ist nicht nötig, beidseitig auf Mitgliedschaft zu testen, weil eine der beiden Seiten die andere durchaus enthalten darf, daher ist es zulässig die erwähnte Abbildung von benannten Typen auf Teilstrukturen in der Generierungsphase schrittweise zu berechnen.

4. Unzulässige Bezeichner enthalten entweder unzulässige Zeichen (wie etwa den Bindestrich) oder entsprechen reservierten Bezeichnern. Im ersten Fall werden die unzulässigen Zeichen mit einem Unterstrich maskiert, im zweiten wird der erste Buchstabe groß geschrieben. Da Schlüsselworte in *C* keine Großbuchstaben enthalten, ist der Bezeichner danach zulässig. Nicht berücksichtigt werden semantische Fehler, die durch die Umbenennung durch Kollision mit anderen Bezeichnern entstehen können.

5. Der Auswahltyp wird als Struktur mit Enumeration und Vereinigung dargestellt, die jeweils Auswahl und Inhalt enthalten. Der Sequenztyp wird als gewöhnliche Struktur synthetisiert und der Listentyp enthält Kapazität, Länge und einen getypten Zeiger auf dynamisch-allozierten Speicher mit dem Inhalt.

Ich habe den Synthesecode aus Gründen der Übersichtlichkeit auf verschiedene Dateien verteilt, die aufeinander verweisen und so indirekt rekursiv sind. Verwendete Anteile der *Glue*-Sprachbibliothek sind:

- der Mengentyp `Set`, in dem jedes Element höchstens ein Mal vorkommt,
- der Abbildungstyp `Map`, mit dem beliebige Objekte mit anderen Objekten assoziiert werden können,
- der Datentyp `Path`, der zum Lesen, Schreiben und Erstellen von Dateien und Ordnern genutzt wird sowie
- die Funktion `eval()`, mit der Zeichenketten und Dateien als *Glue*-Code ausgeführt werden können.

Folgende Skriptdateien sind abgebildet:

- Listing 11 auf Seite 58 enthält den Einstiegspunkt des *ASN.1*-Compilers in die Codegenerierung mit Hilfsstrukturen und Funktionen, die während der Synthese genutzt werden. Konkret enthalten ist die Menge der *C*-Schlüsselworte und die Funktion `mkid()`, mit der eine Zeichenkette in einen gültigen *C*-Bezeichner umwandelt werden kann sowie die Abbildung `containment` und die Funktion `contains()`, die zur Suche in der reflexiv-transitiven Enthaltenbarkeitshülle eines Typs nach einem anderen Typ genutzt werden.
- Listing 12 folgt direkt auf das vorherige Listing und enthält die Hauptschleife. In dieser wird über jedes Modul und alle benannten Typen iteriert und für jeden Typ eine *C*-Headerdatei angelegt. Diese enthält eine Liste von `#include`-Direktiven für alle enthaltenen Teilstrukturen, eine Liste von Vorwärtsdeklarationen indirekt referenzierter Typen und eine Typdefinition, die durch Ausführung des entsprechenden Skriptes erzeugt wird.
- Listing 13 wird für Typreferenzen ausgeführt und erzeugt jeweils entweder eine `#include`-Direktive und eine Kopie des referenzierten Typs oder im Falle rekursiver Inklusion eine Vorwärtsdeklaration und Referenzierung mittels Zeiger.
- Listing 14 generiert Code für *ASN.1*-Sequenzen in Form einer *C*-Struktur mit rekursiv erzeugten Unterstrukturen. Die Struktur ist benannt, falls der Typ dem oberen Typen entspricht und andernfalls anonym.

- Listing 15 erzeugt den Auswahltyp als Struktur mit einer Enumeration zur Kodierung der Wahl sowie einer Vereinigung zur Kodierung des Inhalts. Da Enumerationswerte in *C* stets globale Bezeichner verwenden, wird diesen der Name des Haupttyps vorangestellt. Dieses Schema führt zu Namenskollisionen, falls es Gemeinsamkeiten bei den Namen der Attribute zweier eingebetteter Auswahlinstanzen im gleichen Typ gibt.

- Listing 16 enthält schließlich den Code zur Erzeugung des Listentyps, der als Struktur aus Kapazität, Länge und dynamisch angelegtem Speicher für die Listenelemente besteht.

3.5.3. Diskussion

Listing 17, 18 und 19 zeigen die drei erzeugten Headerdateien für das oben vorgestellte *Glue*-Programm mit der Spezifikation aus Listing 9 als Eingabe. Die Ausgabe setzt implizit voraus, dass der Datentyp `VisibleString` im Ausgabeordner in der Datei "VisibleString.h" deklariert ist. Dieser Datentyp wäre in einem echten Compiler ein Teil der Laufzeitbibliothek und müsste gegebenenfalls in den Ausgabeordner kopiert werden. Bei der Ausgabe für `Type` aus Listing 19 fällt auf, dass die zweifache Indirektion der `choice`- und `sequence`-Attribute vermieden werden kann. Der Synthesecode berücksichtigt beim Listentyp nicht, dass `list` bereits als Zeiger erzeugt und deshalb schon indirekt referenziert wird. Wünschenswert ist ebenfalls die Synthese von Initialisierungs-, Freigabe-, Kodierungs- und Dekodierungsroutinen.

Die genannten Fehler und Unzulänglichkeiten in der Ausgabe lassen sich direkt mit der Skriptdatei des entsprechenden Typs korrelieren, was einen inkrementellen Stil bei der Entwicklung von Codegeneratoren fördert.

3.6. Zusammenfassung

Mein Lösungsvorschlag erfüllt viele der Anforderungen aus Abschnitt 3.2 auf Seite 36, insbesondere beinhaltet *Glue* eine Template Engine mit automatischer Einrückung (1), erlaubt den Import externer Datenstrukturen in die Sprache, ohne eine vollständige Transformation in eingebaute Datentypen zu benötigen (2), automatisiert die meisten Verwaltungsaufgaben durch den Einsatz von automatischer, dynamischer Speicherverwaltung, Typinferenz und einer plattformunabhängigen Ausführungsumgebung mit unbeschränkten Ganzzahlen (3), wird als autonome *C*-Bibliothek ausgeliefert (4) und orientiert sich syntaktisch an der *C*-Sprachfamilie (5).

Es fehlen Konzepte für die strukturierte Behandlung von Laufzeitfehlern. Ausnahmeobjekte werden gegebenenfalls bei Übersetzung oder Ausführung von *Glue*-Skripten produziert, lassen sich allerdings weder behandeln noch gezielt im Rahmen eines Skripts erzeugen. Sie führen zum sofortigen Abbruch des *Glue*-Programms und können zu Diagnosezwecken ausgegeben werden. Ebenfalls ohne Unterstützung sind dynamisch erstellte Strukturen mit Attributen. Es gibt zwar die Kollektionstypen *Tupel* und *Liste*, jedoch keine abstrakten Datentypen.

Problematisch für die Erweiterbarkeit ist die enge Kopplung von Interpretation und abstraktem Syntaxbaum. Jedes neue Sprachelement benötigt einen neuen Knoten, dessen Semantik direkt durch *C*-Code zu implementieren ist. Aus Gründen der Nachvollziehbarkeit wäre es günstiger, atomare Operationen zu identifizieren und durch eine virtuelle Maschine in Form von Bytecode interpretieren zu lassen. Neue Sprachelemente ließen sich dann mithilfe bereits definierter Operationen ausdrücken, was zu erhöhter Transparenz der Implementation und infolgedessen zu vereinfachter Wartung führen würde. Positive Nebeneffekte wären eine erhöhte Effizienz bei der Übersetzung bereits bekannter Programme durch Caching und Serialisierung in Objektdateien sowie ein erweitertes Optimierungspotential.

```
1  new keywords = Set(
2      "auto", "break", "case", "char", "const", "continue", "default", "do", "double",
   ↪   "else", "enum", "extern", "float", "for", "goto", "if", "inline", "int", "long", "register",
   ↪   "restrict", "return", "short", "signed", "sizeof", "static", "struct", "switch",
   ↪   "typedef", "union", "unsigned", "void", "volatile", "while"
3  );
4
5  new mkid(&str) = {
6      new mask = str.mask("^a-zA-Z0-9", "_");
7
8      if (mask: keywords)
9          mask = mask[0].upper + mask[1,-1];
10
11     return mask;
12 }
13
14 new containment = Map();
15
16 new contains(&super, &sub) = {
17     if (super == sub)
18         return true;
19
20     for (it: containment[super])
21         if (contains(it, sub))
22             return true;
23
24     return false;
25 }
```

Listing 11: *Einstiegspunkt für die Codegenerierung mit Hilfsroutinen. Die Menge der Schlüsselworte in Zeile 1 wird in der Funktion `mkid()` in Zeile 5 zur Umwandlung von Zeichenketten in gültige C-Bezeichner verwendet. Die Abbildung `containment` aus Zeile 14 assoziiert Typnamen mit der Menge enthaltener Typen und wird im Laufe des Programms schrittweise berechnet. Zusammen mit der Funktion `contains()` aus Zeile 16 kann damit auf Mitgliedschaft von einem in einem anderen Typ getestet werden, indem die reflexiv-transitive Hülle von letzterem durchsucht wird.*

```
26 for (module: asn1)
27     for (expr: module) {
28         new includes = Set();
29         new forwards = Set();
30         new decl = {
31             new &type = expr.type;
32             return eval(Path(type.which + ".glue"));
33         }
34
35         // create, write and commit the header file
36         Path(expr.name + ".h").write($(
37             #ifndef ${mkid(expr.name.upper)}_H_INCLUDED
38             #define ${mkid(expr.name.upper)}_H_INCLUDED
39
40             $for (include: includes)
41                 #include "$include.h"
42             $$
43
44             $for (forward: forwards)
45                 struct ${mkid(forward)}_s;
46             $$
47
48             typedef $decl ${mkid(expr.name)};
49
50             #endif      /* ${mkid(expr.name.upper)}_H_INCLUDED */
51         )).commit();
52
53         // store the set of physically included sub-structures
54         containment[expr.name] = includes;
55     }
```

Listing 12: *Hauptschleife des Codegenerators. In diesem Abschnitt wird über alle Module aus dem abstrakten Syntaxbaum und darin jeweils über alle benannten Typen iteriert. Für jeden Typ legt das Programm Variablen für die Menge der referenzierten Datentypen an, wobei dabei zwischen direkt enthaltenen und indirekt referenzierten Typen unterschieden wird (Zeile 28 und 29). In Zeile 30–33 wird zur Erzeugung der Datenstrukturen das Glue-Skript für den entsprechenden Typ ausgeführt und erstmalig der Knoten `type` gesetzt. Dieser wird in den Unterskripten erwartet und entspricht der aktuellen Position im Baum. Schließlich werden die Ergebnisse in eine C-Headerdatei geschrieben (Zeile 36) und die Menge der enthaltenen Typen in der `containment`-Abbildung gespeichert (Zeile 54).*

```
1  // check if the referenced type contains the top-level type
2  if (contains(type.typeref, expr.name)) {
3          // don't require forward declarations for the top-level type
4          if (type.typeref != expr.name)
5                  forwards.insert(type.typeref);
6
7          return $(struct ${mkid(type.typeref)}_s*);
8  }
9
10 // generate an include directive and physically include the type
11 includes.insert(type.typeref);
12 return mkid(type.typeref);
```

Listing 13: *Die Schablone für Typreferenzen „typeref.glue“. Zunächst prüft das Skript, ob eine direkte Inklusion zulässig ist, indem mittels `contains()` aus Listing 11 ermittelt wird, ob sich durch die Inklusion eine zyklische Abhängigkeit ergeben würde (Zeile 2). Für indirekt-rekursive Typen wird in diesem Fall eine Vorwärtsdeklaration erzeugt (Zeile 5) und darauf über einen Zeiger verwiesen (Zeile 7). Andernfalls wird die Typreferenz in die Menge der enthaltenen Typen mit aufgenommen und direkt eingebunden (Zeile 11 und 12).*

```
1  new members = [];
2
3  for (it: type.sequence) {
4          new &type = it.type;
5          members.append($(${eval(Path(type.which + ".glue"))} ${mkid(it.name)}));
6  }
7
8  return $(
9          $if (type == expr.type)
10         struct ${mkid(expr.name)}_s {
11         $else
12         struct {
13         $$
14                 $for (member: members)
15                         $member;
16                 $$
17         }
18 );
```

Listing 14: *Die Schablone für Sequenzen „sequence.glue“. Für jedes Element aus der Sequenz wird ein Attribut des entsprechenden Typs generiert (Zeile 5) und in einer C-`struct` gekapselt zurückgegeben (Zeile 8). Für den Top-Level-Typ wird die Struktur benannt (Zeile 10), andernfalls ist sie unbenannt (Zeile 12).*

```
1  new members = [];
2
3  for (it: type.choice) {
4      new &type = it.type;
5      members.append($(${eval(Path(type.which + ".glue"))} ${mkid(it.name)}));
6  }
7
8  return $(
9      $if (type == expr.type)
10     struct ${mkid(expr.name)}_s {
11     $else
12     struct {
13     $$
14         enum {
15             $for (it: self.choice)
16                 ${expr.name.upper}_${it.name},
17             $$
18         } present;
19
20         union {
21             $for (member: members)
22                 $member;
23             $$
24         } choice;
25     }
26 );
```

Listing 15: *Die Schablone für den Auswahltyp „choice.glue“. Analog zu Listing 14 wird für jede Alternative ein Attribut des entsprechenden Typs generiert (Zeile 5) und als C-`union` gepackt (Zeile 20). Zur Kodierung der Wahl zwischen den Alternativen wird eine Enumeration erzeugt (Zeile 14–18) und Instanzen der beiden Typen mit einer C-`struct` kombiniert. Diese ist im Falle des Top-Level-Typs benannt (Zeile 10), ansonsten unbenannt (Zeile 12).*

```
1  new decl = {
2          new &type = type.sequenceOf;
3          return eval(Path(type.which + ".glue"));
4  }
5
6  return $(
7          $if (type == expr.type)
8          struct ${mkid(expr.name)}_s {
9          $else
10         struct {
11         $$
12                 unsigned int cap, len;
13                 $decl* list;
14         }
15 );
```

Listing 16: *Die Schablone für den Listentyp „sequenceOf.glue“. In Zeile 3 wird die Deklaration des Elementtyps erzeugt und dann zusammen mit Attributen für Kapazität und Länge als dynamisches Array mit einer C-`struct` gekapselt (Zeile 6–15). Diese ist im Falle des Top-Level-Typs benannt (Zeile 8), ansonsten unbenannt (Zeile 10).*

```
1  #ifndef AST_H_INCLUDED
2  #define AST_H_INCLUDED
3
4  #include "VisibleString.h"
5  #include "NamedType.h"
6
7
8  typedef struct AST_s {
9          unsigned int cap, len;
10         struct {
11                 VisibleString module;
12                 struct {
13                         unsigned int cap, len;
14                         NamedType* list;
15                 } member;
16         }* list;
17 } AST;
18
19 #endif       /* AST_H_INCLUDED */
```

Listing 17: *Generierte Struktur für den abstrakten Datentyp `AST`. Die Ausgabe wurde durch das Glue-Programm in die Datei "AST.h" geschrieben.*

```
1  #ifndef NAMEDTYPE_H_INCLUDED
2  #define NAMEDTYPE_H_INCLUDED
3
4  #include "VisibleString.h"
5  #include "Type.h"
6
7
8  typedef struct NamedType_s {
9          VisibleString name;
10         Type type;
11 } NamedType;
12
13 #endif       /* NAMEDTYPE_H_INCLUDED */
```

Listing 18: *Generierte Struktur für den abstrakten Datentyp* `NamedType`*. Die Ausgabe wurde durch das Glue-Programm in die Datei "NamedType.h" geschrieben.*

```
#ifndef TYPE_H_INCLUDED
#define TYPE_H_INCLUDED

#include "VisibleString.h"

struct NamedType_s;

typedef struct Type_s {
	enum {
		TYPE_TYPEREF,
		TYPE_CHOICE,
		TYPE_SEQUENCE,
		TYPE_SEQUENCEOF,
	} present;

	union {
		VisibleString typeref;
		struct {
			unsigned int cap, len;
			struct NamedType_s** list;
		} choice;
		struct {
			unsigned int cap, len;
			struct NamedType_s** list;
		} sequence;
		struct Type_s* sequenceOf;
	} choice;
} Type;

#endif	/* TYPE_H_INCLUDED */
```

Listing 19: *Generierte Struktur für den abstrakten Datentyp Type. Die Ausgabe wurde durch das Glue-Programm in die Datei "Type.h" geschrieben.*

4. Fazit

Im Rahmen dieser Arbeit wurden verschiedene Ansätze bei der Entwicklung eines sprachunabhängigen Codegenerierungsmodules für die Erweiterung bestehender Compiler untersucht. Dabei stellte sich heraus, dass Lösungen mit wünschenswerten Eigenschaften existieren, allerdings noch nicht sprachübergreifend verbreitet sind. Aus diesem Grund habe ich die Sprache *Glue* entwickelt, deren Entwurf aus den Stärken und Schwächen bestehender Ansätze lernt und die als Sprache für die Entwicklung von Codegenerierungskomponenten in Frage kommt.

Die Implementation dieser Sprache ist nicht ohne Makel:

- Notation und Ausführung sind durch die direkte, rekursive Interpretation des abstrakten Syntaxbaumes eng miteinander gekoppelt, was Spracherweiterungen aufwändig macht und das Verständnis erschwert.
- Die Implementation unterstützt keine Langzahlarithmetik, sondern lediglich die simplere Sättigungsarithmetik, bei der der Benutzer selbst auf Überläufe testen muss.
- Abstrakte Datentypen und Ausnahmen werden lediglich über die *C*-Schnittstelle unterstützt, jedoch nicht innerhalb der Sprache, was dem Zielsprachenentwickler wichtige Ausdrucksmittel verwehrt.

Meine nächsten Bemühungen konzentrieren sich daher auf:

- Implementation einer virtuellen Maschine mit atomaren Befehlssatz und dazugehörigem Codegenerator,
- Einsatz einer plattformunabhängigen Bibliothek für Langzahlarithmetik, etwa die *GNU Multiple Precision Arithmetic Library* [6],
- Erweiterung der Sprache um Records (in Form von Tupeln mit benannten Attributen), Klassen und strukturierter Ausnahmebehandlung.

Mit diesen Verbesserungen ist mein Ziel die Konstruktion einer generischen Codegenerierungskomponente für den `asn1c` von Walkins [13] mit einer Syntheseeinheit für die Zielsprache C^{++}.

A. Die Sprache *Glue*

A.1. Syntax und Semantik

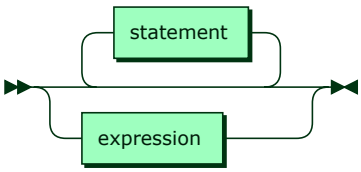
Der folgende Abschnitt enthält die Grammatik für *Glue* notiert in EBNF und visualisiert mit Syntaxdiagrammen. Bei der Notation in EBNF richte ich mich nach der des W3C für *XML*[24], da diese reguläre Ausdrücke mit einbezieht und damit auch die Lexeme beschrieben werden können.

Syntaxdiagramme sind grafische Aufbereitungen der textuellen Notation: jedes Nicht-Terminal hat ein entsprechendes Diagramm mit einem Eintritts- und einem Austrittspunkt. Verbindungslinien markieren erlaubte Pfade zwischen diesen Punkten und durchlaufen Terminale und Nicht-Terminale, die möglichen Eingaben entsprechen. Nicht-Terminale werden durch eckige, Stringliterale durch abgerundete und Zeichenklassen durch abgewinkelte Boxen dargestellt. Die dargestellten Diagramme wurden mithilfe des *Railroad Diagram Generator* von Rademacher [12] erzeugt.

Überblick

Glue-Programme ignorieren grundsätzlich Leerzeichen in Ausdrücken und Anweisungen. Ein Programm besteht entweder aus einem einzelnen Ausdruck oder einer (potentiell leeren) Sequenz von Anweisungen. Einzelne Ausdrücke werden akzeptiert, damit Codeschablonen direkt in eingefügten Dateien angegeben werden können, ohne ein `return`-Statement zu benötigen. Ihre Syntax ist:

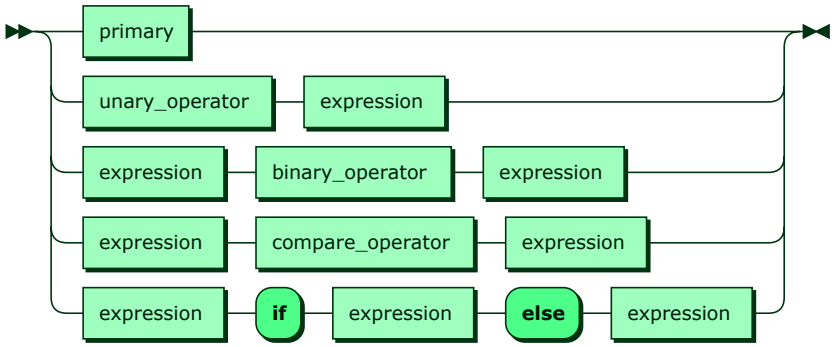
```
▷ program ::= expression | statement*
```



Ausdrücke

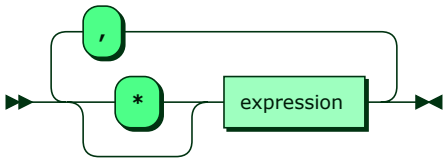
Ausdrücke sind Literale oder das Resultat von Operatoren, angewendet auf Ausdrücke. Die am engsten gebundenen Ausdrücke werden als `primary` bezeichnet und beinhalten sowohl Literale als auch den Index-, Auflösungs- und Rufoperator.

```
expression ::= primary
             | unary_operator expression
             | expression binary_operator expression
             | expression compare_operator expression
             | expression "if" expression "else" expression
```



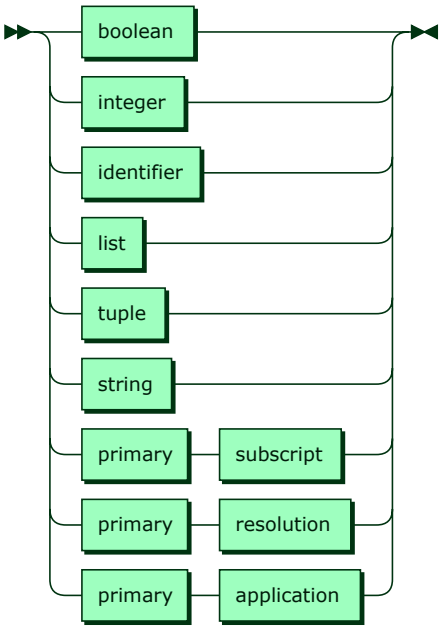
Ausdrücke kommen häufig als kommaseparierte Sequenz vor und lassen sich in dieser Form mit dem optionalen Sternoperator entpacken. Für Kollektionen (das sind alle Ausdrücke, deren `iter()`-Methode definiert ist) läuft der Sternoperator über alle Elemente und fügt diese nacheinander an der entsprechenden Position in die Sequenz ein.

```
expression_list ::= "*"? expression ("," "*"? expression)*
```



Die Gruppe der am engsten gebundenen Operanden wird `primary` genannt und beinhaltet Wahrheitswerte, Ganzzahlen, Bezeichner, Listen, Tupel und Zeichenketten sowie das Resultat von Index-, Auflösungs- und Rufoperationen auf diesen Operanden.

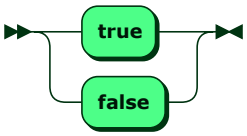
```
primary ::= boolean
          | integer
          | identifier
          | list
          | tuple
          | string
          | primary subscript
          | primary resolution
          | primary application
```



Literale

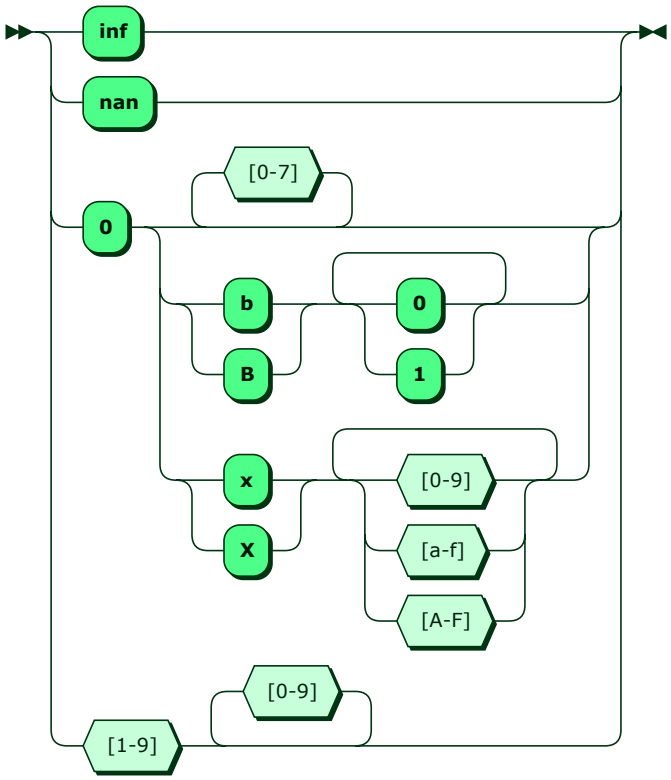
Wahrheitswerte sind entweder `true` oder `false`. Beide können frei mit Ganzzahlausdrücken gemischt werden und sind in diesem Kontext äquivalent zu 1 für `true` und 0 für `false`. Alle Datentypen können eine Konvertierung nach *Boolean* definieren, welche in Bedingungen implizit ausgeführt wird. Eingebaute Zahlen, Zeichenketten und Kollektionen sind `false`, wenn sie 0 oder `nan` bzw. leer sind und andernfalls `true`.

```
▷ boolean ::= "true" | "false"
```



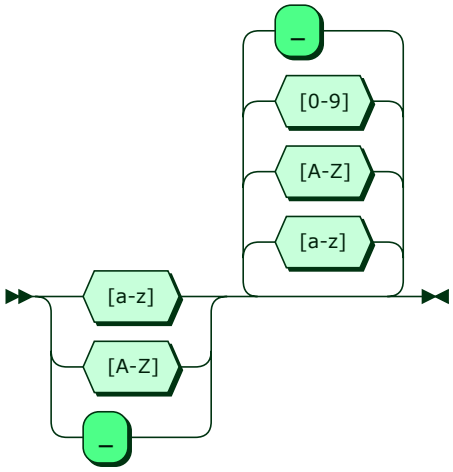
Der Ganzzahltyp unterstützt unendliche und undefinierte Werte sowie endliche Zahlen in Binär-, Hexadezimal-, Oktal- und Dezimaldarstellung. Der Basistyp für die Darstellung ist `intmax_t` und läuft für Zahlen mit zu vielen nicht-trivialen Ziffern nach $\pm\infty$ über.

```
▷ integer ::= "inf" | "nan"
            | "0" ([bB] [01]+ | [xX] [0-9a-fA-F]+ | [0-7]+)?
            | [1-9][0-9]*
```



Bezeichner haben beliebige Länge und berücksichtigen Groß- und Kleinschreibung.

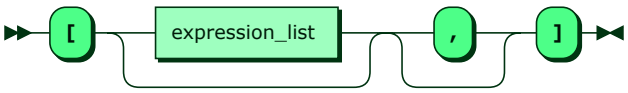
```
identifier ::= [a-zA-Z_][a-zA-Z0-9_]*
```



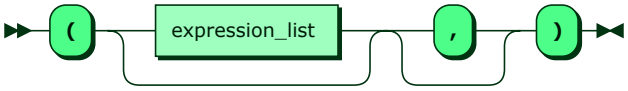
Listen und Tupel sind kommaseparierte Sequenzen von Ausdrücken, die durch eckige bzw. runde Klammern gekapselt sind. Der wichtigste Unter-

schied zwischen diesen beiden Kollektionstypen ist ein semantischer: Listen sind konzeptionell homogene Sequenzen variabler Länge (die Homogenität wird allerdings nicht erzwungen) und Tupel sind inhomogene Sequenzen fester Länge (Vektoren). Die auf den Kollektionen definierten Operatoren reflektieren diesen Unterschied: Listen können verlängert werden, die Länge von Tupeln ist fix.

```
list ::= "[" expression_list? ","? "]"
```



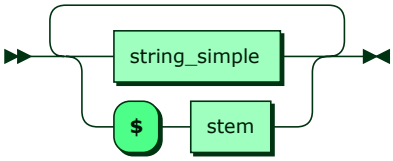

```
tuple ::= "(" expression_list? ","? ")"
```



Zeichenketten

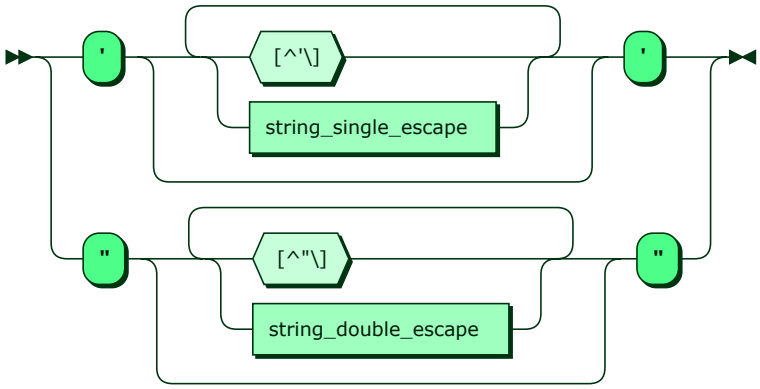
Strings werden in zwei Varianten unterstützt: klassische Sequenzen von Buchstaben und Mischungen von Ausdrücken und Zeichenketten. Strings sind atomar und im Gegensatz etwa zu *C* keine Felder von `char`. Sie unterstützen viele Listenmethoden, können jedoch nicht weiter zerlegt werden. Es ist nicht möglich, auf die unterliegende Repräsentation einzelner Zeichen (etwa in Form eines ASCII-Codes) zuzugreifen. Nebeneinanderliegende Stringliterale werden automatisch konkateniert und wie ein einzelnes Literal behandelt.

```
string ::= (string_simple | "$" stem)+
```



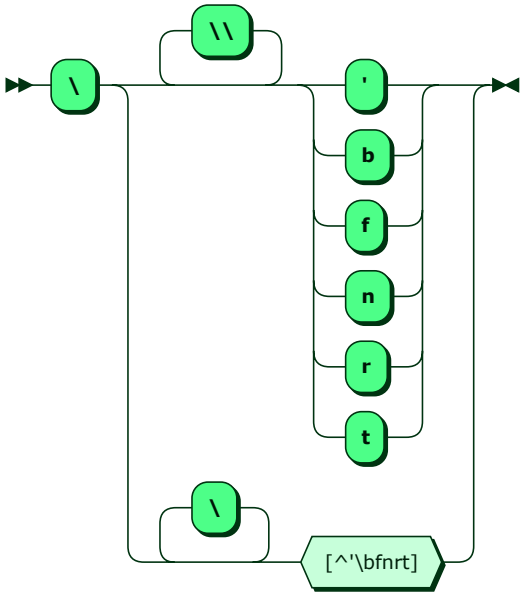
Die klassische Variante erlaubt die Formulierung von Zeichenketten, die durch Anführungszeichen oder Hochkommas eingeschlossen sind. Beide Notationen verhalten sich völlig analog und erlauben lediglich die Annehmlichkeit in hochkommaseparierten Strings Anführungszeichen nicht maskieren zu müssen und umgekehrt. Strings können mehrere Zeilen umfassen, wobei das Zeilenende Teil der Zeichenkette ist.

```
string_simple ::= "'" ([^'\] | string_single_escape)* "'"
                | '"' ([^"\] | string_double_escape)* '"'
```

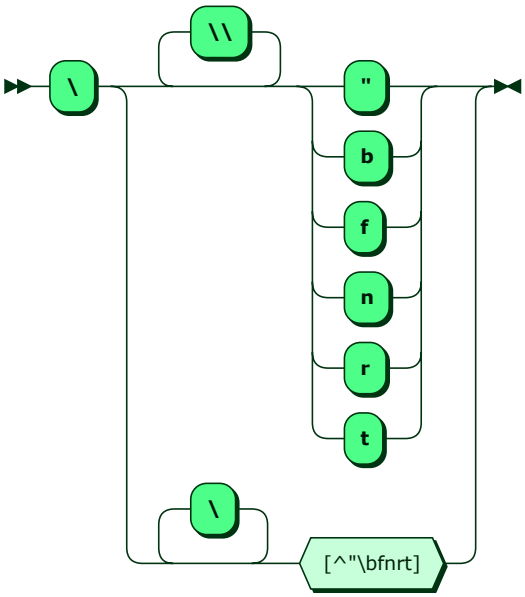


Die Maskierung unterscheidet sich semantisch von der in *C*-Stringliteralen: es werden nur Zeichen maskiert, die andernfalls eine semantische Bedeutung hätten oder ausgewählte Steuerzeichen sind. Es ist offen, ob dieses Verhalten in zukünftigen Versionen so beibehalten wird, da es potentiell Verwirrung stiftet.

```
string_single_escape ::= "\" ("\\"* ['bfnrt] | "\"* [^'\bfnrt])
```




```
string_double_escape ::= "\" ("\\"* ["bfnrt] | "\"* [^"\bfnrt])
```



Template Engine

Schablonen unterscheiden sich von gewöhnlichen Zeichenketten in drei wesentlichen Aspekten:

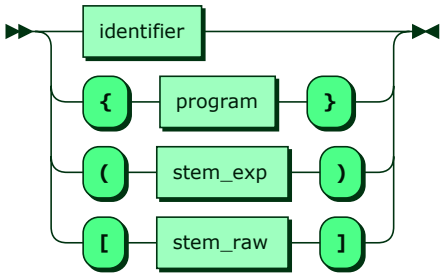
1. Sie unterstützen keine Escapesequenzen für Steuerzeichen und Zeilenenden.

2. Text wird automatisch, basierend auf der Einrückung der ersten Zeile, eingerückt.

3. Dynamische Anteile werden durch ein Dollarzeichen markiert, ausgeführt, implizit in eine Zeichenkette konvertiert und eingerückt an die entsprechende Stelle eingefügt.

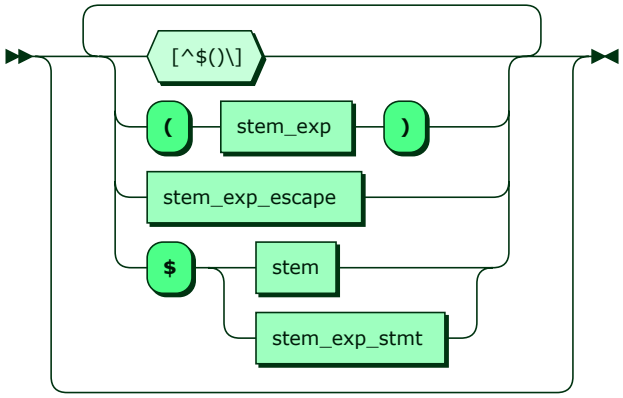
String Templates kommen in zwei Varianten vor: expandierte Schablonen (`stem_exp`) und unbehandelte Schablonen (`stem_raw`), wobei der Hauptunterschied zwischen diesen Modi die Behandlung des Dollarsymbols ist: für expandierte Schablonen ist es standardmäßig aktiviert (und kann maskiert werden) und für unbehandelte Schablonen ist es standardmäßig deaktiviert (und wird durch Maskierung aktiviert). Für Zielsprachen, die dem Dollar ebenfalls eine Sonderrolle zuweisen, kann der Einsatz der unbehandelten Schablonen hilfreich sein.

```
stem ::= identifier
       | "{" program "}"
       | "(" stem_exp ")"
       | "[" stem_raw "]"
```

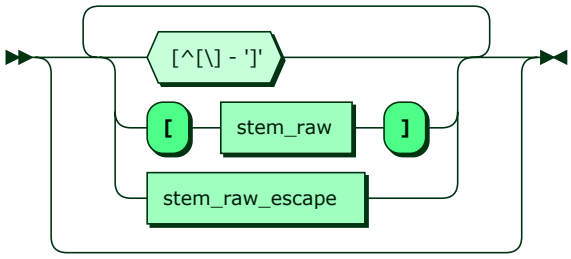


Expandierte Schablonen müssen die runden Klammern in statischem Text nicht maskieren, solange diese ausbalanciert sind. Der Status der eckigen Klammern ist in diesem Kontext irrelevant. Analoges gilt für eckige Klammern im Kontext der unbehandelten Schablonen.

```
stem_exp ::= ([^$()\] | "(" stem_exp ")" | stem_exp_escape | "$"
          ↪  (stem | stem_exp_stmt))*
```

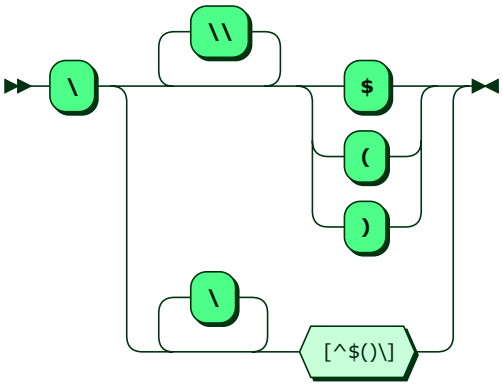



```
stem_raw ::= (([^[\] - ']') | "[" stem_raw "]" | stem_raw_escape)*
```

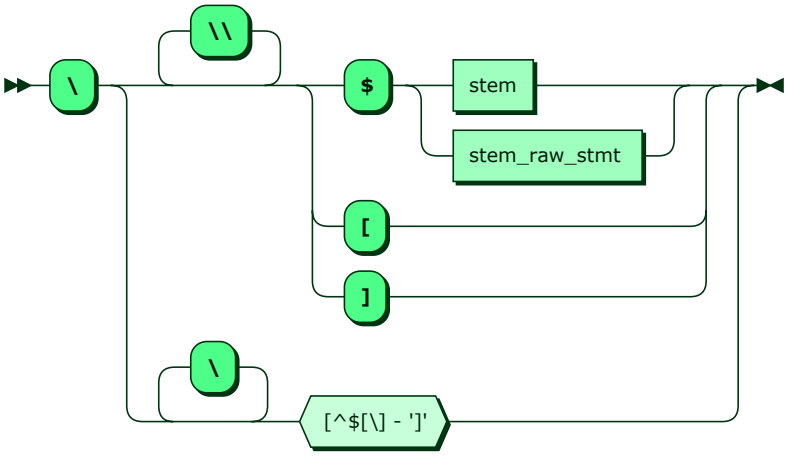


Es müssen grundsätzlich nur die Zeichen maskiert werden, die andernfalls eine semantische Bedeutung hätten, ansonsten ist der Backslash ein gewöhnliches Zeichen ohne besondere Bedeutung. Das minimiert die Anzahl der aktiv zu maskierenden Zeichen zugunsten der Lesbarkeit.

```
stem_exp_escape ::= "\" ("\\"* [$()] | "\"* [^$()\])
```




```
stem_raw_escape ::= "\" ("\\"* ("$" (stem|stem_raw_stmt)|'['|']')
                    ↪  | "\"* ([^$[\] - ']'))
```

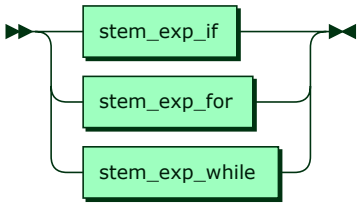


String Templates unterstützen einige eingebettete Kontrollstrukturen. Darunter fallen Bedingungen, Aufzählungen und Schleifen, die alle den `else`-Fall unterstützen. Für Bedingungen wird dieser genau dann ausgeführt, wenn die Bedingung zu `false` evaluiert, für Aufzählungen und Schleifen, wenn der Körper keine Iteration mit `break` abgebrochen hat. Dieses Verhalten ist nützlich, wenn im Körper eine Eigenschaft des aktuellen Aufzählungs- oder Schleifenelementes getestet wird und die Alternative nur bei Nichtfinden eines geeigneten Kandidaten ausgegeben werden

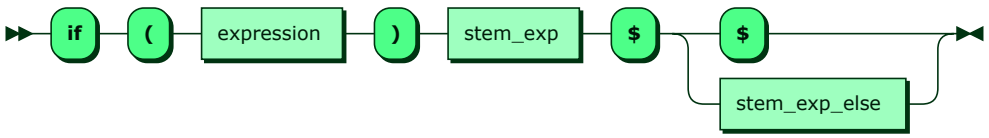
soll. Aufzählungs- und Schleifentypen können mit `break` und `continue` abgebrochen werden.

Da die Kontrollstrukturen für die beiden Expansionsmodi völlig analog sind, habe ich an dieser Stelle nur die Syntaxdiagramme für expandierende Schablonen abgebildet.

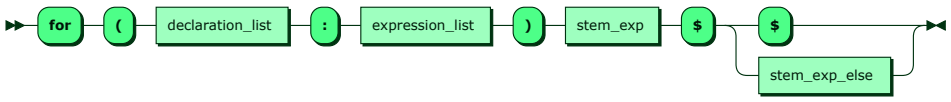
```
stem_exp_stmt ::= stem_exp_if | stem_exp_for | stem_exp_while
stem_raw_stmt ::= stem_raw_if | stem_raw_for | stem_raw_while
```




```
stem_exp_if ::=
        "if" "(" expression ")" stem_exp "$" ("$" | stem_exp_else)
stem_raw_if ::=
        "if" "(" expression ")" stem_raw "$" ("$" | stem_raw_else)
```



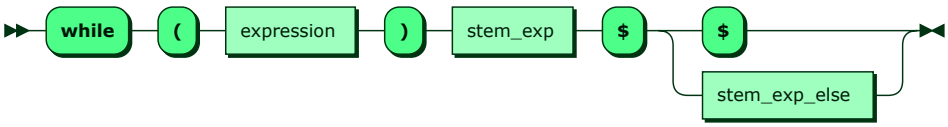

```
stem_exp_for ::=
        "for" "(" parameter_list ":" expression_list ")" stem_exp
    ↪   "$" ("$" | stem_exp_else)
stem_raw_for ::=
        "for" "(" parameter_list ":" expression_list ")" stem_raw
    ↪   "$" ("$" | stem_raw_else)
```



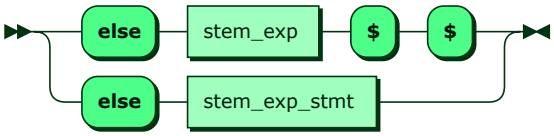

```
stem_exp_while ::=
        "while" "(" expression ")" stem_exp "$" ("$" |
    ↪   stem_exp_else)
stem_raw_while ::=
        "while" "(" expression ")" stem_raw "$" ("$" |
    ↪   stem_raw_else)
```

Operator	Assoziativität	Beschreibung
`.`, `[]`, `()`	→	Auflösung, Index, Ruf
`**`	←	Exponentiation
`~`, `!`, `+`, `-`		unäre Operatoren
`*`, `/`, `%`	→	Multiplikation, Division, Rest
`+`, `-`	→	Addition, Subtraktion
`<<`, `>>`	→	Schiebeoperatoren
`&`	→	bitweises UND
`^`	→	bitweises exklusives ODER
`|`	→	bitweises ODER
`>=`, `<=`, `>`, `<`, `==`, `!=`, `:`, `!:`	→	Vergleich und Mitgliedschaft
`&&`	→	logisches UND
`||`	→	logisches ODER
`if`, `else`	←	bedingte Auswertung

Tabelle 4: *Tabelle mit Operatorpräzedenzen und Assoziativitäten. Die Operatoren sind von vorrangiger zu nachrangiger Präzedenz aufgelistet. Die Richtung der Assoziativität wird durch einen Pfeil ausgedrückt und für unäre Operatoren weggelassen.*




```
stem_exp_else ::= "else" stem_exp "$" "$"
                | "else" stem_exp_stmt
stem_raw_else ::= "else" stem_raw "$" "$"
                | "else" stem_raw_stmt
```



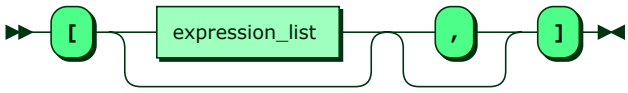
Operatoren

In Tabelle 4 zeigt die Liste aller Operatoren nach Präzedenz geordnet mit deren Assoziativität. Vergleichsoperatoren $\prec$ können verkettet werden, wobei $a \prec b \prec c$ genau dann zu `true` evaluiert, wenn $a \prec b$ und $b \prec c$ ist. Zuweisungen lassen sich nicht verketten und haben daher weder Präzedenz

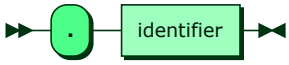
noch Assoziativität. Für die arithmetisch-logischen Operatoren habe ich die Syntaxdiagramme aus Platzgründen ausgelassen.

```
unary_operator ::=
    "+"|"-"|"~"|"!"
binary_operator ::=
    "**"|"+"|"-"|"*"|"/"|"%"|"<<"|">>"|"&"|"^"|"|"|"&&"|"||"
compare_operator ::=
    "<="|">="|"<"|">"|"=="|"!="|":"|"!:"
assign_operator ::=
    "="|"+="|"-="|"*="|"/="|"%="|"**="|"<<="|">>="|"&="|"^="|"|="
```

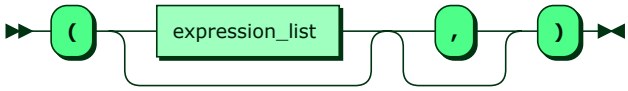
```
subscript ::= "[" expression_list? ","? "]"
```




```
resolution ::= "." identifier
```



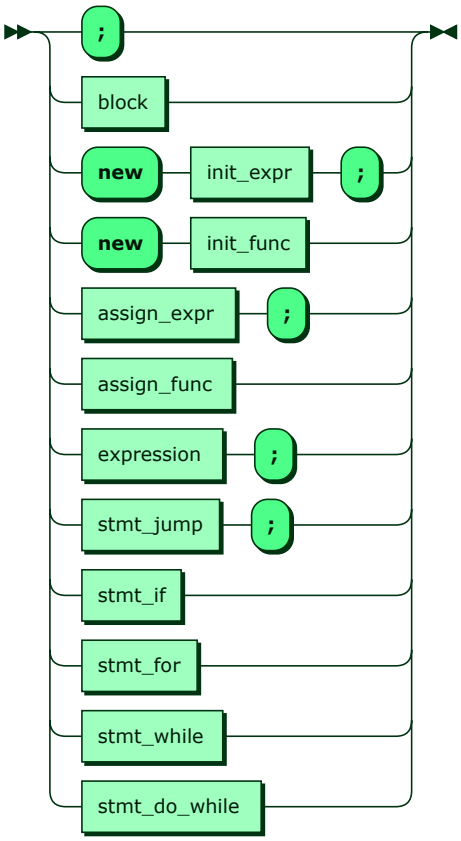

```
application ::= "(" expression_list? ","? ")"
```



Anweisungen

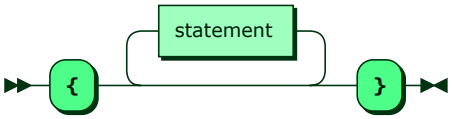
Anweisungen werden gewöhnlich durch ein Semikolon abgeschlossen und Rückgabewerte werden verworfen. Ihre genaue Syntax ist:

```
statement ::= ";"
            | block
            | "new" init_expr ";"
            | "new" init_func
            | assign_expr ";"
            | assign_func
            | expression ";"
            | stmt_jump ";"
            | stmt_if
            | stmt_for
            | stmt_while
            | stmt_do_while
```



Codeblöcke bieten eine Möglichkeit, Anweisungen zu gruppieren und bilden Sichtbarkeitsbereiche für Bezeichner. Neu deklarierte Bezeichner können solche aus höheren Sichtbarkeitsbereichen überdecken und werden nach der Abarbeitung des Blocks automatisch abgeräumt. Bezeichner sind Aliase von Werten und teilen sich den Zugriff darauf mit allen anderen Aliasen des entsprechenden Wertes. Das Entfernen von Bezeichnern korrespondiert nur dann mit der Freigabe eines Wertes, wenn dieser keine weiteren Aliase hat.

```
▷ block ::= "{" statement* "}"
```

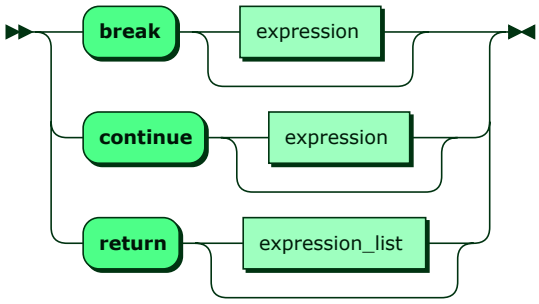


Sprunganweisungen

Unbedingte Sprunganweisungen können verwendet werden, um Schleifen und Funktion zu verlassen. Dabei dürfen `break` und `continue` niemals

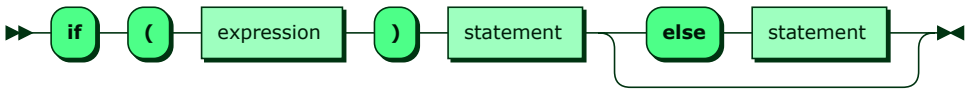
den Funktionskontext verlassen, allerdings eine oder mehrere Schleifen. Die Anzahl der zu verlassenden Schleifen muss ein konstanter, positiver Zahlenausdruck größer als 0 sein, da das Sprungziel statisch berechnet wird. Die `return`-Anweisung verlässt den Kontext der aktuellen Funktion und kann optional einen oder mehrere Rückgabewerte zurückgeben. Im Falle mehrerer Rückgabewerte werden diese in einer Listenkollektion gesammelt.

```
▷ stmt_jump ::= "break" expression?
              | "continue" expression?
              | "return" expression_list?
```



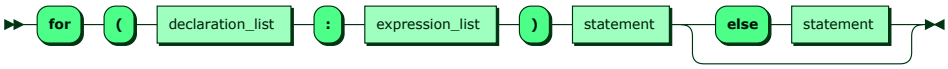
Bedingte Sprunganweisungen sind Bedingungen, Aufzählungen und Schleifen, und unterstützen den optionalen `else`-Fall. Für die if-Anweisung wird die Alternative für negative Bedingungen ausgeführt, im Falle der Schleifen wird sie genau dann ausgeführt, wenn die Schleife nicht mittels `break` verlassen wurde. Dieses Verhalten ist nützlich für Filter, die in einer Sequenz nach einem Wert mit bestimmten Eigenschaften suchen und bei Nichtfinden alternative Wege versuchen möchten.

```
▷ stmt_if ::= "if" "(" expression ")" statement ("else" statement)?
```



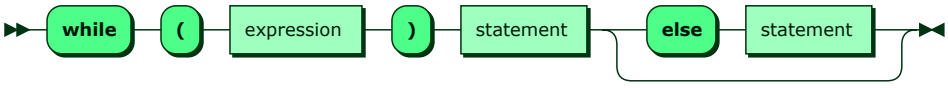
Aufzählungen deklarieren im Schleifenkopf implizit alle Laufvariablen und entfernen diese bei Verlassen des Schleifenkörpers. Sie unterstützen auch die Abarbeitung heterogener Sequenzen, d. h. der Typ der Laufvariablen kann sich zwischen zwei Iterationen unterscheiden.

```
▷ stmt_for ::= "for" "(" declaration_list ":" expression_list ")"
             ↪  statement ("else" statement)?
```



Die `while`-Schleife testet vor dem Betreten des Schleifenkörpers die Bedingung im Schleifenkopf und führt diesen nur für Bedingungen aus, die zu `true` evaluieren. Ansonsten wird der `else`-Fall ausgeführt, falls er existiert.

```
stmt_while ::=
        "while" "(" expression ")" statement ("else" statement)?
```



Die `do-while`-Schleife führt den Schleifenkörper aus, bevor die Bedingung getestet wird. Solange die Bedingung wahr ist, wird der Körper weiter ausgeführt, ansonsten wird die Kontrolle an den `else`-Fall übertragen, falls er existiert.

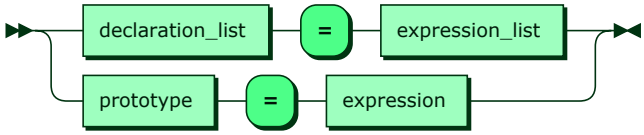
```
stmt_do_while ::=
        "do" statement "while" "(" expression ")" ("else"
    ↪   statement)?
```

Variableninitialisierung

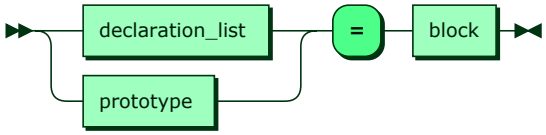
Bezeichner sind konzeptionell Aliase für Variablenwerte, daher stellt die Initialisierung eine Assoziation zwischen einem Wert und einem Namen her. Nichtsdestotrotz hat *Glue* standardmäßig Wertesemantik, d. h. Variablenwerte werden während der Initialisierung berechnet, gegebenenfalls kopiert und dann mit dem Bezeichner assoziiert. Falls ein Bezeichner mit dem Referenzierungsoperator markiert ist, wird der Wert nicht kopiert, sondern lediglich assoziiert. Falls der zu assoziierende Name bereits Teil des aktuellen Sichtbarkeitsbereiches ist, wird die alte durch die neue Assoziation ersetzt.

Es ist möglich, mehreren Variablen eine Sequenz von Ausdrücken oder eine Kollektion zuzuweisen. In letzterem Falle wird diese entpackt und deren Elemente zur Initialisierung der Variablen von links nach rechts verwendet. Falls mehr Elemente Teil der Kollektion bzw. Sequenz sind, als Variablen zur Verfügung stehen, wird eine Ausnahme ausgelöst, die zum Abbruch des Skriptes führt.

```
init_expr ::= declaration_list "=" expression_list
            | prototype "=" expression
```

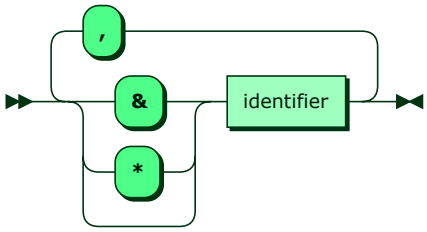



```
init_func ::= (declaration_list | prototype) "=" block
```

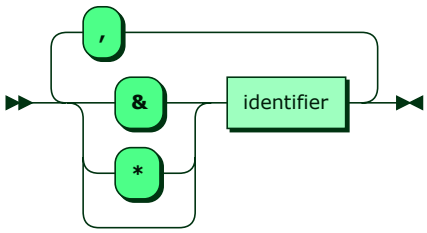


In jeder Deklarations- oder Parameterliste darf ein Bezeichner mit dem Sternoperator markiert werden. Dieser erhält für entpackte Kollektionen genau die Werte in Form einer Liste, die nicht in die anderen Bezeichner gepasst haben. Der ausgezeichnete Bezeichner darf an einer beliebigen Position der Liste stehen und bekommt genau den Ausschnitt, den eine entsprechende Anzahl von Bezeichnern an dieser Stelle erhalten hätte.

```
declaration_list ::= [&*]? identifier ("," [&*]? identifier)*
```

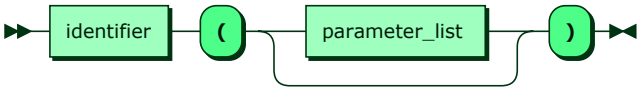



```
parameter_list ::= [&*]? identifier ("," [&*]? identifier)*
```



Funktionen sind Bezeichner mit Parametern. Der Ausdruck auf der rechten Seite der Initialisierung wird nicht direkt ausgeführt, sondern mit dem Bezeichner assoziiert. Erst die Anwendung des Ruf-Operators mit der korrekten Anzahl an Argumenten führt zur Abarbeitung.

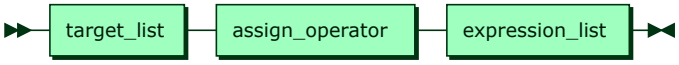
```
prototype ::= identifier "(" parameter_list? ")"
```



Variablenzuweisung

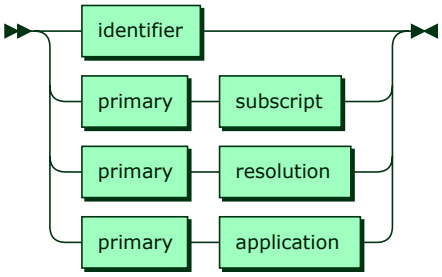
Im Gegensatz zur Initialisierung ist der Variablentyp bei der Zuweisung fixiert und kann nicht verändert werden. Es ist weiterhin zulässig, mehreren Variablen gleichzeitig Werte zuzuweisen, allerdings führt eine unterschiedliche Anzahl von Quell- und Zieltermen zu einer Ausnahme, die den Abbruch der Interpretation bewirkt. Die Verwendung des Sternoperators auf der linken Seite ist nicht erlaubt.

```
assign_expr ::= target_list assign_operator expression_list
```

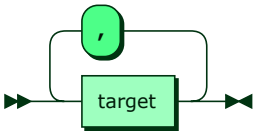



```
assign_func ::= target_list assign_operator block
```


```
target ::= identifier
         | primary subscript
         | primary resolution
         | primary application
```




```
target_list ::= target ("," target)*
```



A.2. Quellcode und Dokumentation

Alle in dieser Arbeit diskutierten Projekte liegen in digitalem Format auf der unten eingebundenen CD vor.

Das sind:

1. die Quellen der Beispiele aus Abschnitt 2 auf Seite 15,
2. der *Glue*-Interpreter als freistehendes Projekt,
3. der modifizierte `asn1c` von Walkins [13], in dem die Codegenerierungskomponente durch den *Glue*-Interpreter ersetzt wurde,
4. eine englischsprachige Dokumentation des von mir geschriebenen Codes im *HTML*-Format,
5. diese Diplomarbeit als PDF-Datei.

Nomenklatur

Abstraktionsgrad	Distanz relativ zu direktem, imperativ ausgeführten Maschinencode bezüglich des Auslassens von Details bei der Beschreibung von Algorithmen
Alphabet	beliebige, endliche Menge von Symbolen
ASN.1	*Abstract Syntax Notation One*; sowohl Sprachstandard zur Beschreibung von Datenstrukturen als auch Sammlung regelbasierter Verfahren mit denen sich diese Strukturen serialisieren und deserialisieren lassen
AST	*Abstract Syntax Tree*; gerichteter, schleifenfreier Graph, der die eingelesenen Quelldateien ohne Verweise auf die konkrete Syntax der Quellsprache repräsentiert; unter Umständen mit weiteren Informationen angereichert
BSD	*Berkeley Software Distribution*; Software unter dieser Lizenz darf frei verwendet, verändert und kopiert werden; einzige Bedingung ist die Aufnahme eines Copyright-Vermerks in abgeleiteten Programmen
Bytecode	Maschinencode einer virtuellen Maschine
Chomsky-Grammatik	besteht aus einem Alphabet von Nichtterminalsymbolen N, einem Alphabet von Terminalsymbolen T, einer Menge von Produktionen P der Form $\alpha \rightarrow \beta$ und einem Startsymbol S
Chomsky-Hierarchie	Kategorisierung von Chomsky-Grammatiken nach Einschränkungen in der Form der Produktionen
Compiler	Übersetzer einer abstrakten Quellsprache in eine weniger abstrakte Zielsprache

DSL	*Domain Specific Language*; formale Sprache, die für eine Klasse von Problemen konstruiert wurde und deren Eignung für generelle Probleme deshalb eingeschränkt ist
EPL	*Erlang Public License*; Software unter dieser Lizenz darf frei verwendet, verändert und kopiert werden, Veränderungen müssen jedoch dokumentiert und ebenfalls unter dieser Lizenz veröffentlicht werden; abgeleitete Programme unterliegen keinen Einschränkungen
formale Sprache	die Menge aller syntaktisch erlaubten Ausdrücke bezüglich einer Chomsky-Grammatik
Hochsprache	formale Sprache, die von konkretem Maschinencode abstrahiert
Implementationssprache	Programmiersprache, in der ein Übersetzer selbst implementiert ist
Interpreter	Programm, das eine Hochsprache in Maschinencode übersetzt und direkt ausführt
Maschinencode	Programmiersprache, deren Instruktionen direkt durch den Prozessor einer Maschine ausgeführt werden können
Metamodell	Modell, das Informationen über eine Klasse von Modellen beinhaltet
Modell	symbolisches Abbild eines Aspektes der Wirklichkeit
Netzwerkprotokoll	Menge von Nachrichten und Regeln, die festlegen, unter welchen Umständen Akteure welche Nachrichten im Netzwerk austauschen
Quellsprache	Eingabesprache für einen Übersetzer

Schablone	Abfolge von statischem Text, Platzhaltern und bedingten Verweisen auf weitere Schablonen, die durch eine Template Engine in eine reine Zeichenkette konvertiert werden kann
Schablonendefinitionssprache	Sprache zur Formulierung von Schablonen
Statisches Wissen	Zusammenhänge, die entweder universell gelten oder sich aus der konkreten Konfiguration von Quellcode ergeben
Symboltabelle	Datenstruktur, welche alle Bezeichner aus der Quelldatei zusammen mit deren Attributen enthält
Synthesesprache	turingmächtige, domänenspezifische Sprache zur Codegenerierung, die eine Template Engine beinhaltet
Template Engine	Programm zur Auswertung von Schablonen
virtuelle Maschine	Programm, das die Ausführung eines echten oder hypothetischen Rechnersystemes simuliert
Zielsprache	Ausgabesprache eines Übersetzers
Übersetzer	Programm, das eine formale Sprache in eine andere formale Sprache übersetzt, ohne die Semantik zu verändern; Generalisierung eines Compilers hinsichtlich des Abstraktionsgefälles von Quell- und Zielsprache

Literatur

[1] CHOMSKY, Noam: Three models for the description of language. In: *IRE Transactions on Information Theory* 2 (1956), September, Nr. 3, S. 113–124. http://dx.doi.org/10.1109/TIT.1956.1056813. – DOI 10.1109/TIT.1956.1056813. – ISSN 0096–1000

[2] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995. – ISBN 0–201–63361–2

[3] In: **[2]**, S. 325–330

[4] In: **[2]**, S. 107–116

[5] GARLAN, David ; SHAW, Mary: An Introduction to Software Architecture. (1994)

[6] GNU: *The GNU Multiple Precision Arithmetic Library*. https://gmplib.org/, Abruf: 18. Mai 2015. – Software

[7] ISO/IEC: C. 2011 (9899:2011). – Internationaler Standard

[8] ITU-T: *Tools*. http://www.itu.int/en/ITU-T/asn1/Pages/Tools.aspx, Abruf: 12. Januar 2015. – Software

[9] LARMOUTH, John: *ASN.1 Complete*. OSS Nokalva, 2000

[10] PARR, Terence J.: *ANTLR: Another Tool for Language Recognition*. www.antlr.org. Version: February 2014, Abruf: 13. Januar 2015. – Software

[11] PARR, Terence J.: Enforcing Strict Model-view Separation in Template Engines. In: *Proceedings of the 13th International Conference on World Wide Web*. New York, NY, USA : ACM, 2004 (WWW '04). – ISBN 1–58113–844–X, 224–233

[12] RADEMACHER, Gunther: *Railroad Diagram Generator*. http://bottlecaps.de/rr/ui, Abruf: 2. Juni 2015. – Webseite

[13] WALKINS, Lev: *asn1c: The ASN.1 Compiler*. http://lionet.info/asn1c/. Version: September 2014, Abruf: 13. Januar 2015. – Software

[14] WEBER, Dorian: *Integration von ASN.1 in SDL-RT*. 2013

[15] ITU-T: ASN.1: Specification of Abstract Syntax Notation One. 1988 (X.208). – Internationaler Standard. – Zurückgezogen am 30. Oktober 2002, da abgelöst durch die ITU-T Empfehlungen [16, 17, 18, 19].

[16] ITU-T: ASN.1: Specification of basic notation. 2008 (X.680). – Internationaler Standard

[17] ITU-T: ASN.1: Information object specification. 2008 (X.681). – Internationaler Standard

[18] ITU-T: ASN.1: Constraint specification. 2008 (X.682). – Internationaler Standard

[19] ITU-T: ASN.1: Parameterization of ASN.1 specifications. 2008 (X.683). – Internationaler Standard

[20] ITU-T: ASN.1: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). 2008 (X.690). – Internationaler Standard

[21] ITU-T: ASN.1: Specification of Packed Encoding Rules (PER). 2008 (X.691). – Internationaler Standard

[22] ITU-T: ASN.1: Specification of XML Encoding Rules (XER). 2008 (X.693). – Internationaler Standard

[23] ITU-T: ASN.1: Specification of Octet Encoding Rules (OER). 2014 (X.696). – Internationaler Standard

[24] W3C: Extensible Markup Language (XML) 1.0 (Fifth Edition). 2008. – Internationaler Standard

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 8. Juni 2015 .....

Statement of authorship

I hereby proclaim my sole authorship of this thesis and that I have not submitted it for examination elsewhere. All sources, including internet sources, that have been copied or reproduced, especially sources for text, graphics, tables and pictures, have been correctly cited. I understand that the violation of these principles can lead to a legal process due to fraud or the attempt of fraud.

Berlin, June 8, 2015 .....