Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik
Department of Computer Science

Abschlussarbeit

im Bachelor-Studiengang Computer Science

Entwicklung einer universellen Sprache zur Verarbeitung von Relationen

von
Wilke Schwiedop

Betreuer:	Prof. Dr. Martin Eric Müller
Zweitbetreuer:	Prof. Dr. Alexander Asteroth
Eingereicht am:	January 29, 2015

Eidestattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbst angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher keiner Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Unterschrift Sankt Augustin, den

Inhaltsverzeichnis

1 Einleitung

1.1 Problemstellung

In zahlreichen Bereichen der Mathematik und der Informatik spielen Relationen eine bedeutende Rolle. Die Untersuchung von Relationen und ihren Eigenschaften ist daher ein wichtiges Themengebiet. Die Untersuchung von Hand ist jedoch aufwendig und fehleranfällig. Daher ist eine computergestützte Untersuchung von Relationen wünschenswert.

Zu diesem Zweck wurde von Prof. Müller 2011 das Rela-X Projekt ins Leben gerufen, welches seit dem kontinuierlich von Studenten der Hochschule Bonn Rhein-Sieg weiterentwickelt wird. Rela-X dient dabei als Dachprojekt für verschiedene Teilprojekte, die sich alle mit verschiedenen Aspekten der Arbeit mit Relationen befassen. Kern des Projektes ist vor allem die Bibliothek RelaFix. RelaFix wurde 2012 von Peter Berger im Rahmem einer Bachelorarbeit erstellt. [Ber12] In der Bibliothek sind eine Vielzahl von Datentypen sowie Basistests und Operationen zur Untersuchung und Arbeit mit Relationen implementiert. Diese Bibliothek wurde seit dem im Rahmen des Bachelor-Projektes bereits verschlankt und weiter modularisiert.

Ursprünglich beinhaltete RelaFix zudem eine Domain Specific Language mit der Mengen und Relationen definiert und einige Funktionen von RelaFix interaktiv verwendet werden konnten. Diese Domain Specific Language war jedoch stark eingeschränkt, daher wurde beschlossen ihre Entwicklung einzustellen.

Da jedoch nach wie vor Bedarf nach einer Möglichkeit besteht, auf die von RelaFix bereitgestellten Funktionen aus einer interaktiven Umgebung heraus zuzugreifen, soll im Rahmen dieser Bachelorarbeit eine Programmiersprache entwickelt werden, die sowohl eine kommandozeilenbasierte Interaktion wie auch die Implementierung komplexerer Algorithmen auf Relationen erlaubt.

1.2 Ziele

Anders als die ursprünglich in RelaFix enthaltene Sprache soll sich die neu entwickelte Sprache nicht auf die Verarbeitung von Relationen beschränken. Zwar soll sie ebenfalls für die Arbeit mit Relationen besonders geeignet sein, jedoch sollen auch Elemente bekannter universell einsetzbare Programmiersprachen integriert werden. Dadurch wird es möglich, weitergehende Eigenschaften einer Relation, beispielsweise die Kardinalität der Domäne einer Relation, abzufragen und den ausgelesenen Wert weiterzuverarbeiten.

Um die Verarbeitung der oben erwähnten Eigenschaften zu ermöglichen, müssen neben dem Datentyp für die zu betrachtenden Relationen zusätzliche primitive Datentypen Teil der Sprachdefinition werden. Die primitiven Datentypen sind zum jetzigen Zeitpunkt int, float und String. Sie entsprechen den Datentypen aus anderen universellen Programmiersprachen wie z.B. Java oder C++.

Dabei ist vor allem auf Kompatibilität zur aktuellen RelaFix-Bibliothek sowie deren paralleler Weiterentwicklung durch weitere Projektmitarbeiter zu achten.

1.3 Mathematische und technische Begriffe

1.3.1 Regulärer Ausdruck / Typ-3 Grammatik

Ein regulärer Ausdruck ist ein Muster für Wörter einer regulären Sprache. Eine Typ-3 Grammatik gibt Regeln an, mit denen die Wörter einer regulären Sprache erzeugt werden können. Reguläre Ausdrücke stellen also ein beschreibendes Konzept, Typ-3 Grammatiken ein erzeugendes Konzept für reguläre Sprachen dar.

Die Bezeichnug "Typ-3" beschreibt dabei die unterste und damit ausdrucksschwächste Stufe der Chomsky-Hierarchie für formale Sprachen.

Dennoch stellen sowohl reguläre Ausdrücke als auch Typ-3 Grammatiken wichtige Hilfsmittel in der Praktischen und in der Angewandten Informatik dar, etwa im Compilerbau und in der Softwaretechnik. So werden reguläre Ausdrücke z.B. zur formalen Beschreibung von zulässigen Eingaben für Dialogmasken verwendet. Es gibt Werkzeuge, die aus der formalen Beschreibung in Form regulärer Ausdrücke automatisch eine Software zur Überprüfung der syntaktischen Korrektheit der Benutzereingaben, sogenannte Scanner, generieren.

Reguläre Ausdrücke sind stets über einem Alphabet Σ definiert und es gelten folgende Regeln.

- Das leere Wort ϵ ist ein regulärer Ausdruck
- Jedes Zeichen aus dem Alphabet ist ein regulärer Ausdruck.
- Jeder Ausdruck, der mit Hilfe der drei Operatoren *Konkatenation*, *Selektion* und *Wiederholung* gebildet werden kann, ist ein regulärer Ausdruck.

Regeln einer regulären Grammatik haben die Form

$$X \to aY$$

$$X \to a$$

$$X \to \epsilon$$

mit $X, Y \in N$ und $a \in \Sigma$. Das bedeutet, dass a ein Terminalsymbol, also ein Zeichen aus unserem Alphabet ist. X und Y sind Nichtterminale, also andere Regeln. Vgl. [VW06]

1.3.2 Kontextfreie Sprache / Typ-2 Grammatik

Wie in Kapitel 1.3.1 beschrieben, steht eine Sprache vom Typ-3 auf der untersten und damit ausdrucksschwächsten Stufe der Chomsky-Hierarchie für formale Sprachen. Es gibt also Sprachen - sogar strukturell sehr einfache Sprachen - die nicht mit regulären Ausdrücken beschrieben und nicht mit Typ-3 Grammatiken erzeugt werden können.

Die nächst höhere Stufe in der Chomsky-Hierarchie sind die Sprachen vom Typ-2. Diese die Klasse der regulären Sprachen umfassenden Sprachen, werden auch als kontextfreie Sprachen bezeichnet. Diese Klasse hat in der Praktischen Informatik eine sehr große Bedeutung. Fast alle verfügbaren Programmiersprachen wie z.B. FORTRAN, COBOL, PL/I, PASCAL, C und Java sind mit Hilfe kontextfreier Konzepte definiert und implementiert.

Zusätzlich zu den Regeln der regulären Grammatiken erlaubt eine kontextfreie Grammatik Regeln der Form

$$X \to aYb$$

mit $X, Y \in N$ und $a, b \in \Sigma$.

Es dürfen in einer Regel also mehrere Terminal- und Nichtterminalsymbole erzeugt werden. Durch diese unscheinbare Änderung wird es möglich, verschachtelte Ausdrücke zu erstellen. Dies ist für Programmiersprachen von entscheidender Bedeutung, mit Typ-3 Grammatiken jedoch nicht möglich. Vgl. [VW06]

1.3.3 Domain Specific Language

Eine Domain Specific Language ist eine formale Sprache, die besonders für ein bestimmtes Anwendungsgebiet entwickelt wurde.

Bekannte DSL sind beispielsweise *CSS* zur Definition der Darstellung einer Webseite, *make* zur Implementierung eines Build Systems für Softwareprojekte und *SQL* zur Steuerung und Abfrage von Datenbanken.

Durch diese Spezialisierung auf ein bestimmtes Anwendungsgebiet ist es möglich, Aufgabenstellungen aus diesem Gebiet schnell und in kürzerer Form umzusetzen, als dies in universellen Programmiersprachen möglich wäre. Andererseits eignen sich solche domänenspezifischen Sprachen häufig nur eingeschränkt oder auch garnicht zur Umsetzung von Aufgaben ausserhalb der Domäne. Daher ist schon beim Entwurf einer DSL besonders darauf zu achten, welche Ziele mit der Entwicklung der Sprache verfolgt werden sollen.

2 Analyse

In Kapitel 1.2 wurden bereits die Ziele genannt, die mit der Entwicklung von RLang verfolgt werden sollen. Im folgenden Kapitel werden diese Anforderungen konkretisiert.

Grundsätzlich soll mit RLang eine interpretierte Sprache entwickelt werden, d.h. der eingelesene Quelltext soll sofort ausgeführt werden. Eine Übersetzung in Maschinensprache oder Bytecode bzw. die Erzeugung einer ausführbaren Datei soll nicht geschehen.

Bei RLang handelt es sich um eine Neuentwicklung in einem umfangreichen Themengebiet. Da der Zeitrahmen eine Bachelorarbeit begrenzt ist, wird für RLang ein iteratives Entwicklungsmodell gewählt. Dadurch wird sichergestellt, dass auch bei unerwartet auftretenden Problemen in der Entwicklung, ein funktionsfähiges Teilergebnis vorliegt.

Im ersten Schritt soll RLang einfache arithmetische Operationen auf Ganz- und Fließkommazahlen ausführen können. Zur Darstellung der Operanden und Ergebnisse werden die Datentypen *int* (Ganzzahlen) und $float$ (Fließkommazahlen) benötigt. Die auf diesen Datentypen ausführbaren Operationen sind PLUS, MINUS, TIMES, POWER, DIVIDE und MODULO (nur Ganzzahlen).

Im zweiten Schritt wird RLang um Vergleichsoperationen zwischen Zahlenwerten und Operatoren aus der booleschen Algebra erweitert. Dafür wird ein weiterer Datentyp *bool* für Wahrheitswerte benötigt. Die Operatoren aus der booleschen Algebra sind NOT, AND, OR, XOR, die Vergleichsoperatoren sind EQUAL, LESS, LESSEQUAL, GREATER und GREATEREQUAL.

Die so ermittelten Zwischenergebnisse der Operationen sollen im dritten Schritt in Form von Variablen zwischengespeichert werden können. In folgenden Operationen sollen diese Variablen anstelle von Konstanten verwendet werden können.

Im vierten Schritt soll RLang um Kontrollstrukturen erweitert werden die es ermöglichen, abhängig von ermittelten Ergebnissen verschiedene Ausführungspfade abzulaufen. Dazu wird RLang um die bedingte Ausführung (IF-THEN-ELSE) und eine Schleife (WHILE) erweitert.

Im fünften Schritt sollen Codefragmente wiederverwendbar gemacht werden. Dies wird über Funktionen realisiert. Diese können mit Parametern aufgerufen werden und können zudem einen Rückgabewert haben. Um Funktionen sinnvoll verwenden zu können, muss RLang zudem über *Sichtbarkeitsbereiche* für Variablen verfügen. Diese auch als *Scopes* bezeichneten Bereiche stellen sicher, dass Variablen innerhalb einer Funktion frei verwendet werden können, ohne mit ausserhalb der Funktion definierten Variablen in Konflikt zu geraten. Da in RLang nicht jede Funktion zwangsweise einen Rückgabewert haben muss, wird der Datentyp *void* zur Sprachdefinitien hinzugefügt.

Durch die im fünften Schritt eingeführte Syntax zum Aufruf von Funktionen, kann RLang im sechsten Schritt um vordefinierte Funktionen erweitert werden. Diese können so auch als Brücke zwischen RLang und RelaFix dienen, in dem die eingebauten Funktionen RelaFix Funktionen aufrufen um etwa Relationen zu erzeugen oder zu verarbeiten. Die Ausgabenfunktionen können ebenfalls mit Hilfe dieser eingebauten Funktionen realisiert werden. Jedoch ist es sicherlich wünschenswert nicht nur Daten ausgeben zu können, sondern auch beliebige Texte. Um diese in RLang verarbeiten zu können wird in diesem Schritt der Datentyp *String* der Sprachdefinition hinzugefügt.

3 Entwurf

Nachdem in Kapitel 2 die funktionalen Anforderungen an RLang spezifiziert wurden, sollen im vorliegenden Kapitel die Grundlagen zur Umsetzung dieser Anforderungen beschrieben werden. Zunächst wird erklärt, wie das Einlesen des Programmtextes (Parsing) implementiert werden soll. Anschließend wird beschrieben wie die zur Umsetzung der Anforderungen nötigen Sprachkonstrukte in Form von grammatikalischen Regeln abgebildet werden. Zuletzt wird erläutert wie die eingelesenen Informationen dargestellt und verarbeiten werden sollen.

3.1 Parsing

Im grundsätzlichen Aufbau sind sich die meisten Interpreter äußerst ähnlich. So muss auch der RLang Interpreter zunächst den Text mit den RLang Ausdrücken aus einer Datei oder der Benutzereingabe einlesen. Ein Computerprogramm liest einen solchen Text jedoch zeichenweise ein, was die Verarbeitung erschwert. An dieser Stelle setzt der Lexer an. Aufgabe des Lexers ist die Zerlegung des Textes in sinnvolle Einheiten. Er identifiziert Schlüsselwörter, Zahlen- und Zeichenkonstanten und sonstige in der Programmiersprache enthaltenen Elemente. Diese Elemente werden auch als Tokens bezeichnet. Übertragen auf die natürliche Sprache ist die Aufgabe des Lexers die einzelnen gelesenen Buchstaben zu Worten der Sprache zu gruppieren.

Darauf aufbauend arbeitet der Parser. Dem Parser liegt die Sprachdefinition in Form einer Grammatik vor. Der Parser erhält vom Lexer die Tokensequenz des eingelesenen Textes und überprüft, ob diese grammatikalisch korrekt ist.

Üblicherweise möchte man mit einem Parser jedoch mehr als nur die grammatikalische Korrektheit eines Textes überprüfen. Daher können die grammatikalischen Regeln des Parsers mit Aktionen verknüpft werden, die ausgeführt werden, sobald die Tokensequenz einer Regel entspricht.

Die oben beschriebenen Arbeitsschritte sind für fast alle Interpreter und Compiler notwendig. Es ist daher nicht verwunderlich, dass für diese Aufgaben eine Vielzahl an spezialisierten Hilfsmitteln existiert, die Entwickler bei der Erstellung von Lexern und Parsern unterstützen. Diese Hilfsmittel werden auch als Parsergeneratoren bezeichnet.

Eines der wichtigsten Ziele bei der Entwicklung von Softwarekomponenten ist die Wiederverwendbarkeit. Durch die Verwendung verschiedener bewährter und frei verfügbarer Programme und Bibliotheken wird die Entwicklung nicht nur vereinfacht sondern auch beschleunigt. Darüber hinaus wird das entwickelte Programm robuster, da diese eingebundenen Teile bereits ausgiebig im Praxiseinsatz erprobt und getestet wurden. Die freie Verfügbarkeit der Komponenten stellt zudem sicher, dass jeder interessierte Entwickler die Arbeit an diesem Projekt fortführen kann. Daher wird auch bei RLang der Parser mit Hilfe eines Parsergenerators erstellt, da es bei der Umsetzung eines Projektes selten sinnvoll ist, alle Komponenten eines Programmes von Grund auf neu zu entwickeln.

Für RLang stehen nun verschiedene Parsergeneratoren zur Auswahl. Im Rahmen der ersten Implementierung von RelaFix wurde von Peter Berger bereits eine Analyse mehrerer Parsergeneratoren durchgeführt [Ber12, Kapitel 2.3]. In dieser Auswertung wurde der Bison [FSF99] Parser für geeignet befunden und zur Umzetzung des Projektes gewählt. Da technisch keine Gründe gegen die Verwendung von Bison sprechen, kommt auch bei RLang Bison zur Generierung

seines Parsers zum Einsatz. Durch die Wahl des Parsergenerators steht zudem fest, dass RLang in C implementiert wird, da dies von Bison vorausgesetzt wird. Dadurch fügt sich RLang zudem sehr gut in das Rela-X Projektumfeld ein.

3.2 Sprachentwurf

In Kapitel 2 wurden bereits die funktionalen Anforderungen an RLang spezifiziert. Dies sagt jedoch weder etwas über das zugrundeliegende Sprachparadigma, noch über das schlussendliche Aussehen der Sprache aus. Bei Programmiersprachen ist es jedoch durchaus üblich, sich an bestehenden Programmiersprachen zu orientieren oder bewährte Konzepte aus einer anderen Sprache zu übernehmen. Dadurch kommt es zur Entstehung sogenannter Sprachfamilien, deren Syntax sich gleicht und die nur in bestimmten Aspekten voneinander abweichen. Dies hat auch für Entwickler Vorteile, da der Umstieg von einer zur anderen Sprache wenig Aufwand bedeutet da viele Konstrukte bereits bekannt sind. RLang wird sich daher ebenfalls an bestehenden Sprachen orientieren. Eine der bekanntesten aktuellen Programmiersprachen ist C. C wurde ursprünglich in den 1970er Jahren entwickelt. Dennoch wird C auch heute noch in vielen Softwareprojekten eingesetzt und die C-Familie zählt mit Programmiersprachen wie C++, C# und Java zu den größten existierenden Sprachfamilien überhaupt. Aus diesem Grund eignet sich die C-Syntax hervorragend als Ausgangspunkt für RLang.

3.2.1 Grammatik

Nachdem geklärt ist welchem sprachlichen Vorbild RLang folgen soll, werden in diesem Kapitel die einzelnen Sprachelemente der Reihe nach erläutert. Die Darstellung der Sprachelemente erfolgt in einer Variante der Backus-Naur Form. (siehe 1.3.2)

```
statements      : statement
                | statements statement
                ;
```

statements ist das Startsymbol des RLang Parsers, d.h. beim Einlesen des Programmtextest beginnt der Parser die schrittweise Auflösung der grammatikalischen Regeln mit diesem Symbol.

statements ist rekursiv definiert. Im konkreten Fall bedeutet dies, dass ein RLang Programm aus einem oder mehreren *statement*s besteht. Diese Liste von *statement*s wird von RLang schrittweise von Anfang bis Ende ausgeführt.

```
statement       : block
                | declaration
                | if
                | while
                | expr ';'
                | 'return' expr ';'
                ;
```

statement beschreibt eine einzelne Anweisung aus der RLang-Sprache.

```
block           : '{' statements '}'
                ;
```

Ein *block* ist eine einzelne Anweisung, die *statements* beinhaltet. Dadurch ist es möglich Anweisungen zu verschachteln bzw. zu gruppieren. Diese Gruppierung von Anweisungen ist vor allem für Funktionen, Verzweigungen und Schleifen von Bedeutung. Ein *block* bildet zudem immer einen eigenen Sichtbarkeitsbereich.

```
declaration     : vardecl '=' expr ';'
                | vardecl ';'
                | function
                ;
```

Eine *declaration* kann sowohl zur Deklaration einer Variablen als auch zur Deklaration einer Funktion aufgelöst werden. Die Variablendeklaration hat den Spezialfall, dass sie in einer einzelnen Anweisung deklariert und mit einem Ausdruck (*expr*) initialisiert wird.

```
if              : 'if' expr statement 'else' statement
                | 'if' expr statement
                ;
```

Die *if*-Anweisung ist die Verzweigung. *expr* wird ausgewertet. Ist der Wert des Ausdrucks der Wahrheitswert 'wahr', wird das erste angegebene *statement* ausgeführt. Ist ein 'else'-Teil angegeben und der Wert des Ausdrucks 'falsch', wird das zweite *statement* ausgeführt. Bei *expr* muss es sich offensichtlich um einen booleschen Ausdruck handeln

```
while           : 'while' expr statement
                ;
```

while beschreibt eine Schleife. Wie schon bei der Verzweigung wird zunächst *expr* ausgewertet. Ist *expr* 'wahr', wird *statement* ausgeführt. Anders als bei der Verzweigung wiederholt sich der Prozess des Auswertens von *expr* und des Ausführens von *statement* jedoch, bis die Auswertung von *expr* 'falsch' ergibt.

```
function        : identifier identifier
                  '(' function_args ')' block
                ;

function_args   : /* empty */
                | vardecl
                | function_args ',' vardecl
                ;
```

function stellt eine Funktionsdefinition dar, wie sie in C verwendet wird. Der erste *identifier* beschreibt den Rückgabetyp der Funktion, also z.B. *int* oder *void*. Der zweite *identifier* ist der Name der Funktion. Dieser Name muss eindeutig sein, da die Funktion in folgenden Ausdrücken mit Hilfe dieses Namens aufgerufen werden kann. *function_args* beschreibt die durch Komma voneinander getrennten Argumente der Funktion.

```
vardecl         : identifier identifier
                ;
```

vardecl dient zur Deklaration von Variablen. Der erste *identifier* beschreibt den Typ der Variablen. Der zweite *identifier* beschreibt den Namen der Variablen. Wie schon bei der Funktionsdefinition muss auch bei der Variablendeklaration

der Name eindeutig sein, d.h. es darf im Sichtbarkeitsbereich der Deklaration keine Variable mit dem gleichen Namen deklariert worden sein.

```
expr            : '(' expr ')'
                | call_expr
                | assign_expr
                | identifier
                | '!' expr
                | '-' expr
                | expr boolean_op expr
                | expr arithmetic_comp expr
                | expr arithmetic_op expr
                | BOOLEAN
                | INTEGER
                | FLOAT
                | STRING
                ;
```

Die Regel *expr* ist eine der vielseitigsten Regeln in RLang. *expr* umfasst Funktionsaufrufe, Zuweisungen von Variablen, arithmetische und boolesche Ausdrücke sowie die Auflösung von Variablen. Auch Konstanten der Typen *bool*, *int*, *float* und *String* sind *expr*-Ausdrücke. *expr* hat daher üblicherweise einen Rückgabewert, der entweder direkt oder nach Auswertung des Ausdrucks vorliegt. Ausnahme dazu ist ein Funktionsaufruf einer Funktion mit dem Rückgabetyp *void*.

```
call_expr       : identifier '(' call_args ')'
                ;

call_args       : /* empty */
                | expr
                | call_args ',' expr
                ;
```

Die Regel *call_expr* beschreibt einen Funktionsaufruf. *identifier* ist der Name der Funktion. *call_args* ist eine durch Kommas getrennte Liste von Ausdrücken. Diese Ausdrücke werden ausgewertet und nach der Auswertung als Parameter an die Funktion übergeben.

```
assign_expr     : identifier := expr
                ;
```

Der *assign_expr* Ausdruck ist eine einfache Variablenzuweisung, bei der der Variablen *identifier* der Wert, der sich durch die Auswertung von *expr* ergibt, zugewiesen wird. Durch die Einbindung von *assign_expr* in die *expr*-Regel sind zudem Mehrfachzuweisungen der Form $a = b = c = ...$ möglich.

```
identifier      : IDENTIFIER
                ;
```

Die *identifier* Regel beschreibt Typ-, Variablen- und Funktionsnamen. Da diese in sehr vielen Regeln Anwendung finden, ist die Auswertung in eine eigene Regel ausgelagert. Da später grammatikalischen Regeln mit Aktionen verknüpft werden, ermöglicht es diese gesonderte Regel die Auswertung von Typ-, Variablen- und Funktionsnamen an einer zentralen Stelle zu erledigen.

```
boolean_op      : '=='
                | '!='
                | '&&'
                | '||'
                ;
```

Die booleschen Operatoren.

== beschreibt den EQUAL Operator.

!= beschreibt den NOTEQUAL (auch XOR) Operator.

&& beschreibt den booleschen AND Operator.

|| beschreibt den booleschen OR Operator.

```
arithmetic_comp : '<'
                | '<='
                | '=>'
                | '>'
                ;
```

Die Vergleichsoperatoren.

< beschreibt den LESS Operator.

<= beschreibt den LESSEQUAL Operator.

=> beschreibt den GREATEREQUAL Operator.

> beschreibt den GREATER Operator.

```
arithmetic_op   : '+'
                | '-'
                | '*'
                | '/'
                | '**'
                | '%'
                ;
```

Die arithmetischen Operatoren.

+ beschreibt den PLUS Operator.

− beschreibt den MINUS Operator.

∗ beschreibt den TIMES Operator.

/ beschreibt den DIVIDE Operator.

∗∗ beschreibt den POWER Operator.

% beschreibt den MODULO Operator.

3.2.2 Sprachelemente

Schlüsselworte	Operatoren	Sonderzeichen
if	==	{ }
else	! =	( )
while	&&	,
return	||	;
	<	
	<=	
	=>	
	>	
	+	
	−	
	*	
	/	
	**	
	%	

3.3 Entwurf der zentralen Datenstruktur

Im vorhergehenden Kapitel wurde die RLang Sprache in Form einer Grammatik formal spezifiziert. Mit Hilfe dieser Grammatik kann ein Parser wie Bison RLang Programme einlesen.

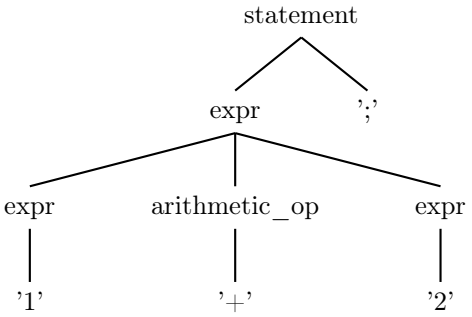
Bevor mit der Auswertung der RLang-Befehle begonnen werden kann, muss der Programmtext vollständig ausgewertet werden. Wie schon in Kapitel 3.1 beschrieben, lassen sich die grammatikalischen Regeln des Parsers mit Aktionen verknüpfen, die ausgeführt werden, wenn eine Regel auf einen Textabschnitt angewandt wird. Diese Aktionen direkt zur Umsetzung der Befehle zu verwenden eignet sich jedoch nur für sehr einfache Sprachen. Interpreter für solche Sprachen bezeichnet man auch als Syntax-directed Interpreter [Par09, P.24].

Selbst bei verhältnismäßig einfachen Sprachen wie RLang stößt man mit dieser Form der Auswertung auf einige Probleme. Beispielsweise kann eine Funktion nur noch dann ausgeführt werden, wenn die entsprechende Funktion bereits vom Parser eingelesen wurde. Dies führt dazu, dass Funktionen nicht mehr an beliebiger Stelle im Quelltext definiert werden können, sondern stets oberhalb der Stellen an denen sie verwendet wird. Es ist daher sinnvoll, zunächst den gesamten Quelltext einzulesen, bevor mit seiner Analyse und Auswertung begonnen wird. Den Quelltext mehrfach einzulesen wäre jedoch viel zu aufwendig. Wir brauchen daher eine Möglichkeit, den Quelltext in unserem Interpreter darzustellen, so dass die Semantik des Quelltextes nicht verändert wird.

Betrachtet man das Programm $1 + 2$; so besteht dieses Programm aus einem einzelnen *statement*, genau genommen einer *expr* gefolgt von einem ';' Token. Die *expr* wird zur 9. Alternative der *expr* Regel aufgelöst (siehe Kapitel 3.2.1). Sie besteht aus einer *expr*, einem *arithmetic_op* und einer weiteren *expr*.

Die erste *expr* besteht aus dem *int* Token '1'. Der *arithmetic_op* besteht aus dem Token '+'. Die zweite *expr* besteht aus dem Token '2'.

Diese Verschachtelung der verschiedenen grammatikalischen Regeln lässt sich sehr anschaulich als Baum darstellen:



Diese Darstellungsform wird auch als Parse- oder Syntaxbaum bezeichnet.

Prinzipiell eignen sich Bäume sehr gut um hierarchische Strukturen in einem Programm darzustellen. Der Syntaxbaum hat jedoch einige Schwächen. Zum einen beinhaltet der Syntaxbaum alle Symbole, die auch in der Sprache vorkommen. Dadurch ist der Syntaxbaum eng an die Grammatik der Sprache gebunden. Eine Änderung innerhalb der Grammatik würde also bedeuten, dass viele weitere Komponenten des Interpreters angepasst werden müssten, da sich ja auch der Syntaxbaum geändert hat. Eine losere Kopplung zwischen dem Parser und dem Interpreter wäre daher wünschenswert.

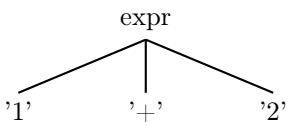
Zum Anderen ist es für eine Programmiersprache häufig sinnvoll, mehr als nur die absolut notwendigen Symbole zu verwenden. Vielfach wird ein Programm dadurch lesbarer oder es wird schlicht einfacher, den entwickelten Quelltext zu parsen. Für die Auswertung braucht der Interpreter jedoch nur die sematisch relevanten Teile eines Befehls. Um die Verarbeitung zu vereinfachen ist es also sinnvoll, auch nur diese Teile zu verwenden.

Zu diesem Zweck soll eine vom Syntaxbaum abstrahierte Datenstruktur geschaffen werden. Diese Datenstruktur soll:

- Minimal sein. In der Datenstruktur sollen nur die semantisch relevanten Elemente der Sprache abgebildet werden.
- Abstrakt sein. Durch die Datenstruktur soll eine lose Kopplung zwischen Parser und Interpreter ermöglicht werden.
- Korrekt sein. Die Semantik des eingelesenen Programms darf durch die Überführung in diese Datenstruktur nicht verändert werden.

Zur Veranschaulichung dient erneut der Syntaxbaum für $1 + 2;$
Zunächst können überflüssige Symbole aus dem Baum entfernt werden. Das ';' Token dient in RLang der Trennung von Statements. Das Symbol macht es für den menschlichen Betrachter einfacher, das Ende eines Statements zu erkennen, semantisch hat es jedoch keine Bedeutung und kann daher aus der Datenstruktur entfernt werden.

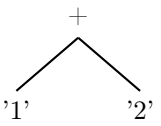
Danach können alle Knoten, die nur ein einzelnes Kind besitzen, durch eben diesen einen Kindknoten ersetzen werden. Durch die beiden Schritte ergibt sich zunächst folgender Baum:



Wie zu sehen ist, besteht dieser Baum nur noch aus einem *expr* Knoten mit drei Kindknoten.

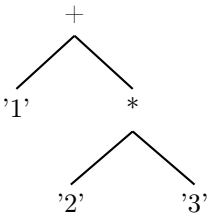
Um diesen Baum weiter zu vereinfachen, muss die Sematik der einzelnen Elemente betrachtet werden. Der oben genannte Ausdruck besteht aus zwei Zahlsymbolen und einem Infix-Operator, der diese beiden Zahlen verknüpft. Der *expr* Knoten selbst hat dabei jedoch keine Bedeutung. Er wird ersetzt durch den Operator, der die beiden Operanden des Ausdrucks verknüpft. Der Baum besteht nun aus 3 Knoten, von denen keiner entfernt werden kann, ohne die Semantik des Ausdrucks zu verändern. Der Baum ist damit minimal.

Um die Forderung nach Abstraktion zu erfüllen, muss nun lediglich das '+' Symbol, welches ja noch aus dem Syntaxbaum stammt und in RLang den Additionsoperator darstellt, durch einen abstrakten Additionsoperator + ersetzt werden. Dadurch entsteht schlussendlich der Baum

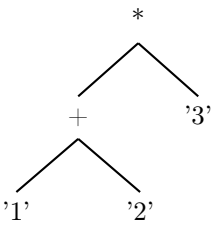


Offen bleibt damit nur noch die Frage nach der Korrektheit. Für diesen einfachen Fall ist die Frage schnell zu beantworten, da der Baum in jedem Fall nur eine Interpretation zulässt.

Ein etwas komplexeren Fall ist der Ausdruck $1 + 2 * 3;$. Hier ist die Operatorpräzedenz zu beachten. Es muss also zuerst $2 * 3$ berechnet werden, bevor das Ergebnis mit 1 addiert werden kann. Dies kann erreicht werden, indem die Berechnung von $2 * 3$ im Baum "nach unten" bewegt wird.



Der zweite Operand der Addition ist also keine Zahlenkonstante, sondern selber das Ergebnis einer Rechenoperation. Im Vergleich dazu müsste beim Ausdruck $(1 + 2) * 3;$ zunächst $1 + 2$ berechnet werden, bevor das Ergebnis mit 3 multipliziert wird. Der Baum für diesen Ausdruck wäre also



Durch diese Form der Darstellung lassen sich auch beliebig komplizierte Ausdrücke ohne die Verwendung zusätzlicher Hilfsmittel korrekt darstellen.

Diese Datenstruktur findet sich in vielen Sprachanwendungen in ähnlicher Form wieder. Sie wird als abstrakter Syntaxbaum oder im Englischen als abstract syntax tree, kurz AST, bezeichnet. [Par09, Kapitel 4.2]

3.4 Abstrakter Syntaxbaum

Im vorhergehenden Kapitel wurde gezeigt, wie ein Parserbaum schrittweise zu einem abtrakten Syntaxbaum umgeformt wird, um daraus eine geeignete Datenstruktur zur Darstellung des eingelesenen RLang Programms abzuleiten. In diesem Kapitel wird versucht, mit Hilfe dieser Datenstruktur geeignete Lösungeswege zur Umsetzung der Anforderungen an RLang zu finden.

Zunächst einmal existieren mehrere Möglichkeiten einen abstrakten Syntaxbaum (AST) zu implementieren: homogen, normalisiert heterogen und (irregulär) heterogen. [Par09, P.9-11] Diese unterscheiden sich in der Art und Weise wie ein Knoten aus dem AST und seine jeweiligen Kindknoten im Programm dargestellt werden.

Beim einem heterogenen AST erhält jeder Knoten eine eigene Datenstruktur in der die spezifischen Eigenschaften des Knoten abgebildet werden. So hätte beispielsweise der Knoten für eine Additionsoperation in der heterogenen Darstellung zwei Felder für seine beiden Operanden. Die explizite Benennung aller Eigenschaften und Kindknoten erlaubt eine sehr intuitive Verwendung des Knotens. Der Nachteil liegt darin, dass Operationen, die den gesamten Baum betreffen sollen, sehr aufwendig sind, da jeder Knoten einen Sonderfall darstellt. Die normalisiert heterogene Darstellung verringert dieses Problem indem die Kinder eines Knoten, anders als bei der heterogenen Form, nicht als einzelne benannte Felder geführt werden, sondern als einheitliche Liste. In der homogenen Form existiert nur eine einzige Datenstruktur, die für alle Knoten des Baumes verwendet wird. Diese Darstellung eignet sich vor allem für Sprachen, die keine komplexen Vererbungshierarchien erlauben.

Da zur Implementierung von RLang mit C bereits eine nicht-objektorientiere Programmiersprache gewählt wurde, fällt die Wahl auf die homogene Repräsentierung des AST.

Dabei wird jeder Knoten mit der gleichen Datenstruktur dargestellt, so dass zunächst eine Möglichkeit benötigt wird, die verschiedenen Knoten voneinander zu unterscheiden. Dies geschieht über einen eindeutigen Typ-Identifikator. Abhängig von dieser Id besitzt ein Knoten eine Anzahl von Kindknoten oder einen Wert. Der Wert ist dabei eine Zeichenkette, die unverarbeitet vom Parser übernommen wurde und abhängig von der Id verarbeitet werden muss.

N_BLOCK Ein N_BLOCK Knoten dient zur Abbildung einer Befehlssequenz. Der N_BLOCK Knoten hat einen oder mehrere Kindknoten, die schrittweise nacheinander ausgeführt werden.

N_IF Ein N_IF Knoten dient zur Abbildung des IF-THEN-ELSE Konstruktes. Der erste Kindknoten ist der auszuwertende Ausdruck, der zweite der THEN-Zweig des IF-THEN-ELSE Konstruktes. Der dritte Knoten ist optional und stellt den ELSE-Zweig dar.

N_WHILE Ein N_WHILE Knoten dient zur Abbildung der WHILE Schleife. Der erste Kindknoten ist der auszuwertende Ausdruck, der zweite der zu wiederholende Befehl.

N_DECLARATION Ein N_DECLARATION Knoten dient zur Abbildung einer Variablendeklaration. Der erste Kindknoten ist ein Identifier mit dem Typ der Variablen. Der zweite Kindknoten ist der Name der deklarierten Variablen.

N_FUNCTION Ein N_FUNCTION Knoten dient zur Abbildung einer Funktionsdefinition. Der erste Kindknoten ist ein Identifier mit dem Typ der Funktion. Der zweite Kindknoten ist der Name der deklarierten Funktion. Der dritte Kindknoten ist ein N_FUNCTIONARGS Knoten mit den Argumenten der Funktion. Der vierte Knoten ist ein N_BLOCK Knoten mit dem Funktionskörper.

N_FUNCTIONARGS Ein N_FUNCTIONARGS Knoten dient zur Abbildung der Argumente, die eine Funktion erwartet. Der N_FUNCTIONARGS Knoten hat keinen oder mehr Kindknoten vom Typ N_DECLARATION.

N_RETURN Ein N_RETURN Knoten dient zur Abbildung eines return-statements. Ein return-statement beendet den aktuellen Funktionsaufruf. Ein N_RETURN Knoten hat nur dann einen Kindknoten wenn die aufgerufene Funktion einen Rückgabewert hat.

N_CALL Ein N_CALL Knoten dient zur Abbildung eines Funktionsaufrufes. Der erste Kindknoten ist ein Identifier mit dem Namen der aufzurufenden Funktion, der zweite ein N_CALLARGS Knoten mit den Argumenten des Funktionsaufrufes.

N_CALLARGS Ein N_CALLARGS Knoten dient zur Abbildung der Argumente eines Funktionsaufrufes. Die Anzahl der Kindknoten hängt von der Anzahl der Argumente der Funktion ab.

N_ASSIGNMENT Ein N_ASSIGNMENT Knoten dient zur Abbildung einer Variablenzuweisung. Der erste Kindknoten ist ein Identifier mit dem Namen der Variablen, der zweite ist ein Ausdruck dessen Wert nach der Auswertung der Variablen zugewiesen wird.

N_IDENTIFIER Ein N_IDENTIFIER Knoten dient zur Abbildung eines Identifiers. Ein N_IDENTIFIER Knoten hat keine Kindknoten. Der Wert des Knotens ist der Identifier.

N_NEG Ein N_NEG Knoten dient der Abbildung einer arithmetischen Negation. Der N_NEG Knoten hat einen einzelnen Kindknoten, den zu negierenden arithmetischen Ausdruck.

N_NOT Ein N_NOT Knoten dient der Abbildung einer booleschen Negation. Der N_NOT Knoten hat einen einzelnen Kindknoten, den zu negierenden booleschen Ausdruck.

N_EQ Ein N_EQ Knoten dient zur Abbildung des Gleich-operators. Der Knoten hat zwei Kindknoten, welche die beiden Operanden darstellen.

N_NEQ Ein N_NEQ Knoten dient zur Abbildung des Ungleich-operators. Der Knoten hat zwei Kindknoten, welche die beiden Operanden darstellen.

N_AND Ein N_AND Knoten dient zur Abbildung des booleschen Und-operators. Der Knoten hat zwei Kindknoten, welche die beiden Operanden darstellen.

N_IOR Ein N_IOR Knoten dient zur Abbildung des booleschen Oder-operators. Der Knoten hat zwei Kindknoten, welche die beiden Operanden darstellen.

N_LT Ein N_LT Knoten dient zur Abbildung des Kleiner-operators. Der Knoten hat zwei Kindknoten, welche die beiden Operanden darstellen.

N_LE Ein N_LE Knoten dient zur Abbildung des Kleinergleich-operators. Der Knoten hat zwei Kindknoten, welche die beiden Operanden darstellen.

N_GE Ein N_GE Knoten dient zur Abbildung des Größergleich-operators. Der Knoten hat zwei Kindknoten, welche die beiden Operanden darstellen.

N_GT Ein N_GT Knoten dient zur Abbildung des Größer-operators. Der Knoten hat zwei Kindknoten, welche die beiden Operanden darstellen.

N_ADD Ein N_ADD Knoten dient zur Abbildung des Additions-operators. Der Knoten hat zwei Kindknoten, welche die beiden Operanden darstellen.

N_SUB Ein N_SUB Knoten dient zur Abbildung des Subtraktions-operators. Der Knoten hat zwei Kindknoten, welche die beiden Operanden darstellen.

N_MUL Ein N_MUL Knoten dient zur Abbildung des Multiplikations-operators. Der Knoten hat zwei Kindknoten, welche die beiden Operanden darstellen.

N_DIV Ein N_DIV Knoten dient zur Abbildung des Divisions-operators. Der erste Kindknoten ist der Dividend, der zweite der Divisor.

N_POW Ein N_POW Knoten dient zur Abbildung des Exponential-operators. Der erste Kindknoten ist die Basis, der zweite der Exponent.

N_MOD Ein N_MOD Knoten dient zur Abbildung des Modulo-operators.

N_BOOLEAN Ein N_BOOLEAN Knoten dient zur Abbildung eines booleschen Wahrheitswertes. Ein N_BOOLEAN Knoten hat keine Kindknoten. Der Wert des Knotens ist der boolesche Wahrheitswert.

N_INTEGER Ein N_INTEGER Knoten dient zur Abbildung einer Ganzzahlkonstanten. Ein N_BOOLEAN Knoten hat keine Kindknoten. Der Wert des Knotens ist die Ganzzahlkonstante.

N_FLOAT Ein N_FLOAT Knoten dient zur Abbildung einer Fließkommazahlkonstanten. Ein N_BOOLEAN Knoten hat keine Kindknoten. Der Wert des Knotens ist die Fließkommazahlkonstante.

N_STRING Ein N_STRING Knoten dient zur Abbildung einer Zeichenkette. Ein N_BOOLEAN Knoten hat keine Kindknoten. Der Wert des Knotens ist die Zeichenkette.

N_SET Ein N_SET Knoten dient zur Abbildung einer Menge.

N_R Ein N_R Knoten dient zur Abbildung einer Relation.

Auf Grundlage dieses Entwurfes kann mit der Implementierung begonnen werden.

4 Implementierung

Nachdem das Kapitel 3.4 die zentrale Datenstruktur für das Programm beschreibt, wird in diesem Kapitel die Implementierung der Sprache näher betrachtet.

Da entschieden wurde einen Tree-Based Interpreter [Par09, P.25] zu entwickeln, gestaltet sich die Implementierung des Parsers unter Verwendung des Bison Parsergenerators äußerst einfach. Er muss lediglich anhand der in Kapitel 3.2.1 definierten grammatikalischen Regeln einen Baum konstruieren, wie er in Kapitel 3.4 genauer beschrieben ist.

Dazu wird die folgende Datenstruktur definiert:

```
typedef struct {
    NodeClass       class;
    char            *value;
    List<AST>       children;

    Scope   *scope
    Symbol  *symbol;
    Type    *type;
} AST;
```

NodeClass beschreibt die Typ-Id aus Kapitel 3.4. Im Feld *value* wird, sofern vorhanden, der Wert des Knotens hinterlegt. Das letzte Feld, das für diesen Schritt benötigt wird, ist das Feld *children* in dem die Kinder des Knotens gespeichert sind [1]. Die übrigen Felder sind erst in späteren Schritten erforderlich. Mit dem Aufbau unseres AST kann nun begonnen werden.

Dazu ist zunächst ein Lexer notwendig. Zur Generierung des Lexers wird Flex genutzt. Es handelt sich dabei um ein Programm, das häufig in Verbindung mit Bison verwendet wird. Die Konfigurationsdatei von Flex besteht, ähnlich wie die des Parsers, aus Regeln, die mit Aktionen verknüpft werden können. Anders als beim Parser handelt es sich bei diesen Regeln jedoch nur um Reguläre Ausdrücke und nicht um eine Kontextfreie Grammatik.

Zur Veranschaulichung seien hier einige Regeln aus der Konfiguration dargestellt.

```
"+"             { return ADD; }
"-"             { return SUB; }
"*"             { return MUL; }
"/"             { return DIV; }

"**"            { return POW; }
"%"             { return MOD; }

"<"             { return LT; }
"<="            { return LE; }
">="            { return LE; }
">"             { return GT; }
```

[1] C besitzt keine native Listenstruktur und schon gar keine generischen Listen. Tatsächlich wird diese Liste in RLang mit Hilfe einer zusätzlichen **next* Referenz im AST struct als verkettete Liste umgesetzt. Diese Details wurden aus Gründen der Übersichtlichkeit und besseren Verständlichkeit weggelassen.

```
"{"             { return LBRACE; }
"}"             { return RBRACE; }
"("             { return LPAREN; }
")"             { return RPAREN; }
```

Auf der linken Seite findet sich der Reguläre Ausdruck, in den geschweiften Klammern die mit der Regel verbundene Aktion. Die erste Regel ist wie folgt zu verstehen: Wenn der Lexer ein '+' einliest, gibt er die Konstante 'ADD' an die aufrufende Funktion zurück. In unserem Fall ist die aufrufende Funktion der Parser.

Der Parser verwendet diese Konstanten anstelle der Literale aus der ursprünglichen Regeldefinition. Exemplarisch sei an dieser Stelle die *expr*-Regel aufgeführt.

```
expr : LPAREN expr RPAREN     { $$ = $2; }
| call_expr
| assign_expr
| identifier
| NOT expr                    { $$ = ast_new(N_NOT);
                                ast_append_child($$, $2);
                              }
| SUB expr                    { $$ = ast_new(N_NEG);
                                ast_append_child($$, $2);
                              }
| expr boolean_op expr        { $$ = $2;
                                ast_append_child_all($$, $1, $3);
                              }
| expr arithmetic_comp expr   { $$ = $2;
                                ast_append_child_all($$, $1, $3);
                              }
| expr arithmetic_op expr     { $$ = $2;
                                ast_append_child_all($$, $1, $3);
                              }
| BOOLEAN                     { $$ = ast_new(N_BOOLEAN);
                                $$->value = $1;
                              }
| INTEGER                     { $$ = ast_new(N_INTEGER);
                                $$->value = $1;
                              }
| FLOAT                       { $$ = ast_new(N_FLOAT);
                                $$->value = $1;
                              }
| STRING                      { $$ = ast_new(N_STRING);
                                $$->value = $1;
                              }
```

Neben den bereits bekannten Regeln finden sich hier auch die mit den Regeln verknüpften Aktionen. Diese sind wie schon beim Lexer in geschweiften Klammern dargestellt.

Die spezielle Variable $$ beschreibt den Wert, der von der jeweiligen grammatikalischen Regel erzeugt wird. In unserem Fall handelt es sich dabei um

einen Pointer auf den Knoten des AST.

Die spezielle Variable $\$n$ dient zur Referenzierung des n-ten Elementes einer Regel. Dies kann entweder ein Knoten sein, wie z.B. bei der $NOT\ expr$-Regel oder eine Zeichenkette, die bei den Regeln $BOOLEAN$, $INTEGER$, $FLOAT$ und $STRING$ dem Wert des Knoten zugewiesen wird.

Damit wurden alle Komponenten beschrieben, die zur Konstruktion des AST benötigt werden. Dem durch Bison erstellten Parser kann man nun eine Datei mit RLang-Befehlen übergeben und erhält ein Abbild dieses Programms als AST.

4.1 Analyseschritte

Bevor jedoch mit der Auswertung des Programms begonnen werden kann, ist das Programm genauer zu analysieren. Das Programm wird zwar durch den Parser auf syntaktische Korrektheit überprüft, jedoch gibt es einige weitere Dinge, die schon vor Ausführung des Programms zu überprüfen sind. Da das Programm zunächst in einen AST überführt wurde, kann dies in mehrenen, voneinander unabhängigen Schritten erfolgen. Dazu werden mehrere External Tree Visitor [Par09, P.13] verwendet, also Programmkomponenten, die den abstrakten Syntaxbaum traversieren und dabei bestimmte Aktionen ausführen.

4.1.1 Symbole und Sichtbarkeitsbereiche

Eine der Anforderungen aus Kapitel 2 war die Umsetzung von Variablen und Sichtbarkeitsbereichen. Da diese beiden eng miteinander Verknüpft sind, werden sie zusammen in einem eigenen Kapitel vorgestellt.

Betrachtet man zunächst folgendes Beispiel:

```
int i = 10;
while i > 0 {
    ...
    i := i - 1;
}
```

Hier wird zunächst eine Variable mit dem Wert 10 definiert und diese dann in einer Schleife bei jedem Durchlauf um 1 verringert. Dies wird wiederholt bis i den Wert 0 erreicht hat. Effektiv bedeutet dies, dass die restlichen Anweisungen im Schleifenkörper zehnmal ausgeführt werden. Damit das Beispiel funktioniert, muss das Programm aber verstehen, dass es sich bei dem i, das im Schleifenkörper um 1 verringert wird, dem i aus dem Schleifenkopf und dem i aus der Definition um das selbe i handelt.

Das Programm braucht also etwas, das es ihm erlaubt, eine Variable durch den Programmcode hindurch zu verfolgen. Dazu wird jeder Variablen ein sogenanntes Symbol zugewiesen, das in einer Symboltabelle hinterlegt ist. Im Verlauf dieses Kapitels wird gezeigt, dass diese Symbole nicht nur für Variablen verwendet werden können, sondern für alle Dinge, die in unserem Programm verfolgt werden müssen.

Bei der Entwicklung eines größeren Programms würde es mit fortschreitender Entwicklungsdauer zunehmend schwieriger, sinnvolle Bezeichner für die Variablen zu finden, da die Symboltabelle mit den bereits verwendeten Bezeichnern

immer weiter anwächst. Zur Lösung dieses Problems existieren zwei verschiedene Ansätze.

Der erste Ansatz behält die Symboltabelle unverändert bei und gibt den Entwicklern schlicht die Möglichkeit, Symbole durch eine Art *undefine*-Befehl wieder aus der Symboltabelle zu entfernen. Dieser Ansatz erfordert offensichtlich eine manuelle Verwaltung aller Variablen und ist daher aufwendig.

Der zweite Ansatz ist die Einführung von Sichtbarkeitsbereichen. Ein Symbol, dass innerhalb eines Sichtbarkeitsbereiches definiert wird, ist auch nur innerhalb dieses Bereiches sichtbar und verliert nach Verlassen des Bereiches seine Gültigkeit. Dies bedeutet, dass sich ein Bezeichner i auf zwei völlig verschiedene Symbole beziehen kann, je nachdem wo i im Programm vorkommt. Üblicherweise lassen sich Sichtbarkeitsbereiche zudem verschachteln. Wenn das Symbol zu einem Bezeichner gefunden werden soll, müssen also nicht nur eine, sondern möglicherweise mehrere Symboltabellen durchsucht werden.

Dies kann an einem Beispiel veranschaulicht werden, das im Folgenden zeilenweise analysiert wird:

```
1  int a := 10;
2  {
3      a := 5;
4      int a := 1;
5      int b := a + 1;
6      {
7          int c := b + 1;
8      }
9  }
```

1 Definition der Variable a mit dem Wert 10 im globalen Sichtbarkeitsbereich.

2 Erstellung eines neuen Sichtbarkeitsbereiches innerhalb des globalen Sichtbarkeitsbereiches.

3 Zuweisung des Wertes 5 zur Variablen a aus dem globalen Sichtbarkeitsbereich.

4 Definition einer neuen Variable a mit dem Wert 1. Diese *überlagert* die Variable a aus dem globalen Sichtbarkeitsbereich.

5 Definition der Variable b mit dem Wert $a + 1$. Da die Definition von a aus dem aktuellen Sichtbarkeitsbereich die Definition aus dem globalen Sichtbarkeitsbereich überlagert, wird b also der Wert 2 zugewiesen.

6 Erstellung eines neuen Sichtbarkeitsbereiches.

7 Definition der Variable c mit dem Wert $b+1$. Die Definition zu b befindet sich im übergeordnen Sichtbarkeitsbereich, c wird also der Wert 3 zugewiesen.

8 Ende des aktuellen Sichtbarkeitsbereiches. Die Variable c verliert ihre Gültigkeit.

9 Ende des aktuellen Sichtbarkeitsbereiches. Die Variable b verliert ihre Gültigkeit. Die Variable a wird nun nicht von einer anderen mehr überlagert, d.h. a hat der Wert 5.

Wie aus dem Beispiel entnommen werden kann, sind Symbole immer an einen Sichtbarkeitsbereich gebunden. Technisch betrachtet stellen die Sichtbarkeitsbereiche also einen Container für Symbole dar. Wenn also ein Bezeichner zu einem Symbol aufgelöst werden soll, muss zunächst herausgefunden werden, in welchem Sichtbarkeitsbereich er sich befindet. Sobald das bekannt ist, wird die Symboltabelle des aktuellen oder des übergeordneten Bereiches nach dem Bezeichner durchsucht. Dieser Schritt wird wiederholt bis entweder das Symbol gefunden wurde, oder kein übergeordneter Sichtbarkeitsbereich mehr existiert.

Damit gelangt man zu folgenden Datenstrukturen:

```
typedef struct {
    char *name;
    Type type;
} Symbol;
```

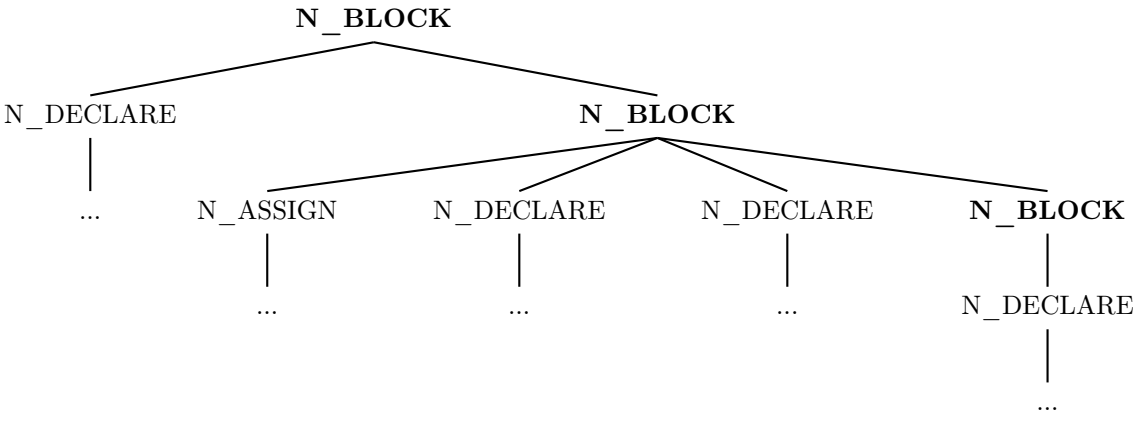
Das Symbol hat einen Namen, der dem Namen der Variablen oder der Funktion entspricht. Das Typ-Feld wird erst später verwendet.

```
typedef struct {
    Scope          *parent;
    List<Symbol>   symbols;
} Scope;
```

Der Sichtbarkeitsbereich ist wie oben beschrieben ein Container. Er besitzt eine Referenz auf den übergeordneten Sichtbarkeitsbereich und eine Liste mit den definierten Symbolen.

So bleibt also noch offen, wie herausgefunden werden kann, zu welchem Sichtbarkeitsbereich ein Knoten gehört. Die einfachste Möglichkeit besteht darin, in jedem Knoten zu markieren, welchem Sichtbarkeitsbereich er zugeordnet ist. Zu diesem Zweck befindet sich in der Datenstruktur für den AST das Feld *scope*.

Um nun jedem Knoten den passenden Sichtbarkeitsbereich zuzuweisen, werden die am Anfang von Kapitel 4.1 erwähnten External Tree Walker verwendet. In einen AST überführt würde das letzte Beispiel wie folgt aussehen:



Wie man sehen kann wird jeder Sichtbarkeitsbereich durch einen der hervorgehobenen N_BLOCK Knoten dargestellt. Der oberste Knoten ist dabei der globale Sichtbarkeitsbereich. Man traversiert nun den Baum und weisen jedem

Knoten den globalen Sichtbarkeitsbereich zu, bis man auf einen N_BLOCK Knoten stoßt.

Ist man bei einem N_BLOCK Knoten angelangt, wird ein neuer Sichtbarkeitsbereich erstellt und allen folgenden Knoten dieser Sichtbarkeitsbereich zugewiesen. Durch diesen einfachen Algorithmus erhält jeder Knoten den Sichtbarkeitsbereich, zu dem er gehört.

Nachdem nun von jedem Knoten ausgehend gesagt weren kann, wo mit der Suche nach einem Symbol begonnen werden soll, können nun die Symbole selbst festgelegt werden. Zu diesem Zweck wird erneut ein Tree Walker genutzt, der den AST mit den nötigen Informationen versieht. Da RLang noch keine Möglichkeit zur Definition eigener Typen hat, werden Symbole dafür im Interpreter festgelegt. Es bleiben also zwei verschiedenen Knoten, in denen Symbole erstellt werden: N_DECLARATION für Variablensymbole und N_FUNCTION für Funktionssymbole.

Bei der Erstellung dieser Symbole sind einige Kriterien zu überprüfen.

- Das Symbol muss einen zulässigen Namen haben. Beispielsweise darf eine Variable nicht den Namen eines Typs tragen.
- Im aktuellen Sichtbarkeitsbereich darf keine Definition mit gleichem Namen existieren. Die *Überlagerung* von Symbolen aus höheren Sichtbarkeitsbereichen ist jedoch zulässig.

Die so definierten Symbole werden in verschiedenen Knoten benötigt. Zunächst muss also für alle verwendeten Typen, Variablen und Funktionen überprüft werden, ob zu einem Bezeichner ein entsprechendes Symbol existiert. Darüber hinaus können an dieser Stelle noch folgende potenzielle Probleme erfasst werden:

- Eine Variable wird verwendet (gelesen) bevor für die Variable eine Zuweisung durchgeführt wurde.
- Eine Variable wird nie verwendet.
- Eine vom Nutzer definierte Funktion wird nie aufgerufen.

Wird ein solches Problem erkannt wird der Entwickler durch eine Warnung darauf hingewiesen.

Funktionen mit Rückgabewert sind ein Sonderfall an dem Symbole eingesetzt werden. Sie benötigen zwingend einen N_RETURN Knoten innerhalb des Funktionskörpers. Allen N_RETURN Knoten wird dazu das Symbol der Funktion zugewiesen. Sie ist auch später zur Überprüfung des des korrekten Rückgabetyps erforderlich.

Nach Abschluss dieser Schritte sind alle Informationen zu allen Symbolen, die im Programm nachverfolgt werden müssen, in den Knoten des AST hinterlegt.

4.1.2 Typen

Nachdem im vorigen Kapitel die Erstellung von Sichtbarkeitsbereichen und Symbolen beschrieben wurde, folgen in diesem Kapitel Erläuterungen zum Umgang mit Datentypen.

Aus den Anforderungen in Kapitel 2 kann man entnehmen, dass insgesamt sieben Typen benötigt werden.

T_BOOL beschreibt den Datentyp *bool* und dient zur Darsetllung von booleschen Wahrheitswerten.

T_INT beschreibt den Datentyp *int* und dient zur Darstellung positiver und negativer Ganzzahlen.

T_FLOAT beschreibt den Datentyp *float* und dient zur Darstellung von Fließkommazahlen.

T_STRING beschreibt den Datentyp *String* und dient zur Darstellung von Zeichenketten.

T_SET beschreibt den Datentyp *Set* und dient zur Darstellung von Mengen. Diese werden insbesondere zur Abbildung der Domänen einer Relation benötigt.

T_R beschreibt den Datentyp *R* und dient zur Darstellung von Relationen.

T_VOID ist streng genommen kein Datentyp. Eine Funktion mit dem Rückgabetyp *void* ist eine Funktion *ohne* Rückgabewert.

Für die Typzuweisung wird der AST traversiert und dabei jedem Knoten der passende Typ zugwiesen. Da nicht jeder Knoten auch einen Typ hat kann man sich dabei auf die Knoten beschränken, die der *expr* Regel untergeordnet sind.

- Den trivialen Knoten N_BOOL, N_INT, N_FLOAT, N_STRING, N_SET, N_R kann direkt ihr zugehöriger Datentyp zugewiesen werden.
- Bei den Knoten N_DECLARATION und N_FUNCTION, bei denen ein neues Symbol erzeugt wurde, wird der Typ zusätzlich dem *zugehörigen Symbol* zugewiesen.
- Knoten, die ein solches Symbol verwenden, z.B. N_IDENTIFIER und N_CALL, erhalten ihren Typ von ihrem zugehörigen Symbol.

Die weitaus schwierigste Situation entsteht bei den verschiedenen binären Operatoren. Beispielsweise werden bei der Operation $1 + 1$ zwei Ganzzahlen miteinander addiert, das Ergebnis ist also wieder eine Ganzzahl. Das bedeutet, dass beide Operanden *und* des gesamten Ausdrucks vom Typ T_INT sind. Bei der Operation $1+1.0$ ist der zweite Operand jedoch eine Fließkommazahl. Damit beim Errechnen des Ergebnisses nicht der Wert nach dem Komma verloren geht, muss also auch das Ergebnis der Operation vom Typ T_FLOAT sein. Andererseits soll das Ergebnis der Operation $1 + true$ weder vom Typ T_INT noch vom Typ T_BOOL sein, sondern schlicht als fehlerhafter Ausdruck erkannt werden.

Zur Zuweisung des korrekten Typs zu einem Ausdruck in Abhängigkeit von seinen Parametern werden Typtabellen verwendet, wie sie in [Par09, P.21 Automatic Type Promotion] beschrieben wurden. Dabei wird nicht für jeden Operator eine eigene Tabelle benötigt. Beispielsweise ist die Typtabelle für den '+'-Operator identisch mit der des '$*$'-Operators. Genau genommen sind insgesamt vier Tabellen erforderlich:

Die erste Tabelle ist Tabelle 1 für die Gleich- und Ungleichheitsoperatoren '==' und '! ='. Wie gut zu erkennen, ist der Operator für alle Typen definiert.

Auch wenn die Operanden nicht vom gleichen Typ sind, können die beiden Gleichheitsoperatoren angewendet werden. Das Ergebnis der Operation ist stets ein boolescher Wahrheitswert.

	void	bool	int	float	String	Set	R
void	-	-	-	-	-	-	-
bool	-	bool	bool	bool	bool	bool	bool
int	-	bool	bool	bool	bool	bool	bool
float	-	bool	bool	bool	bool	bool	bool
String	-	bool	bool	bool	bool	bool	bool
Set	-	bool	bool	bool	bool	bool	bool
R	-	bool	bool	bool	bool	bool	bool

Tabelle 1: Typtabelle für den Gleich- und Ungleichheitsoperator

Deutlich anders sieht es bei Tabelle 2 mit den booleschen Operatoren '&&' und '||' aus. Zwar erzeugen diese Operatoren ebenfalls stets einen booleschen Wahrheitswert, anders als die Gleichheitsoperatoren sind sie jedoch nur für boolesche Operanden definiert.

	void	bool	int	float	String	Set	R
void	-	-	-	-	-	-	-
bool	-	bool	-	-	-	-	-
int	-	-	-	-	-	-	-
float	-	-	-	-	-	-	-
String	-	-	-	-	-	-	-
Set	-	-	-	-	-	-	-
R	-	-	-	-	-	-	-

Tabelle 2: Typtabelle für die booleschen Operatoren

Die arithmetischen Operatoren wie '+' und '−' sind in Tabelle 3 abgebildet. Diese sind wie weiter oben beschrieben für Ganz- und Fließkommanzahlen definiert.

	void	bool	int	float	String	Set	R
void	-	-	-	-	-	-	-
bool	-	-	-	-	-	-	-
int	-	-	int	float	-	-	-
float	-	-	float	float	-	-	-
String	-	-	-	-	-	-	-
Set	-	-	-	-	-	-	-
R	-	-	-	-	-	-	-

Tabelle 3: Typtabelle für die arithmetischen Operatoren

Die letzte Tabelle ist Tabelle 4 mit den Vergleichsoperatoren wie '<' und '>='. Diese sind wie die arithmetischen Operatoren für Ganz- und Fließkommazahlen definiert, jedoch ist die Anwendung der Vergleichsoperatoren auch auf andere Datentypen möglich. Für Mengen (Typ T_SET) könnten die Operatoren verwendet werden um zu überprüfen, ob es sich bei der einen Menge um eine

Teilmenge der anderen handelt. Für Zeichenketten könnte entweder die Länge oder auch eine andere Metrik wie die Levenshtein Distanz verwendet werden.

Dies sind jedoch Überlegungen für kommende Entwicklungen. In der aktuellen Version von RLang sind die Vergleichsoperatoren nur für numerische Werte definiert.

	void	bool	int	float	String	Set	R
void	-	-	-	-	-	-	-
bool	-	-	-	-	-	-	-
int	-	-	bool	bool	-	-	-
float	-	-	bool	bool	-	-	-
String	-	-	-	-	-	-	-
Set	-	-	-	-	-	-	-
R	-	-	-	-	-	-	-

Tabelle 4: Typtabelle für die Vergleichsoperatoren

So wird jedem Ausdruck ein passender Typ zugewiesen. Das alleine wäre jedoch relativ sinnlos, wenn mit Hilfe dieser Typen keine Korrektheitsprüfung vorgenommem werden kann um den Entwickler auf eventuelle Fehler hinzuweisen.

Tatsächlich müssen zur vollständigen Überprüfung des Programmes nur einige wenige Knoten kontrolliert werden.

N_IF Bei der Verzweigung muss es sich bei der Bedingung um einen booleschen Ausdruck handeln.

N_WHILE Bei der Schleife muss es sich bei der Bedingung um einen booleschen Ausdruck handeln.

N_DECLARATION (mit Zuweisung) Der Typ des Ausdrucks muss dem Typ der Variablen entsprechen.

N_ASSIGNMENT Der Typ des Ausdrucks muss dem Typ der Variablen entsprechen.

N_CALL Die Parameter müssen den in der Funktionsdefinition festgelegten Typ haben.

N_RETURN (ausser T_VOID) Der Ausdruck muss dem Typ der Funktion entsprechen.

Stimmen die Typen nicht überein wird der Entwickler über eine Meldung über den Fehler informiert.

Um den Entwicklern die Arbeit an dieser Stelle weiter zu vereinfachen, wird als weitere Funktion die automatische Typkonvertierung eingeführt: Sie konvertiert unter bestimmten Bedingungen Werte von einem Typ automatisch in einen anderen. Dies wird im folgenden Beispiel veranschaulicht:

```
void examplefunction(float f) {
    ...
}

int i := 1;
examplefunction(i);
```

Hier wird zunächst die Funktion *examplefunction* mit einem *float* Parameter *f* definiert. Der Aufruf der Funktion erfolgt jedoch mit der *int* Variable *i*. Der *int* Wert 1 wird dabei automatisch in einen *float* Wert 1.0 konvertiert.

Bevor also die Typen innerhalb der Ausdrücke auf Korrektheit kontrolliert werden, ist zu überprüfen, ob die vorliegenden Typen verlustfrei in die benötigten Typen konvertiert werden können. Dazu verwendet man erneut eine Typtabelle. Die in Tabelle 5 stellt die eingesetzten Konvertierungen dar. In der Zeile befindet sich der Ursprungstyp, die Spalte beschreibt den möglichen Zieltyp. Der Ursprungstyp aus einer Zeile kann also in den Zieltyp aus den verschiedenen Spalten konvertiert werden.

Aktuell ist die Konvertierung von *int* nach *float* als einzige Konvertierung vorgesehen, jedoch sind auch andere Konvertierungen wie etwa von *int* oder *float* nach *bool* denkbar. Deshalb wurde an dieser Stelle die sehr allgemeine Lösung für das Problem gewählt.

	void	bool	int	float	String	Set	R
void	-	-	-	-	-	-	-
bool	-	-	-	-	-	-	-
int	-	-	-	float	-	-	-
float	-	-	-	-	-	-	-
String	-	-	-	-	-	-	-
Set	-	-	-	-	-	-	-
R	-	-	-	-	-	-	-

Tabelle 5: Typtabelle für die automatische Typkonvertierung

Mit Abschluss der Typprüfung sind alle von RLang vorgesehenen semanischen Analyseschritte durchgeführt worden. So kann sichergestellt werden, dass das Programm zumindest keine groben Fehler mehr enthält, die die Ausführung des Programms verhindern würden.

4.2 Ausführung

In Kapitel 4.1 wurde die Analyse des Programms ausführlich beschrieben. Diese Analyse umfasste jedoch lediglich die Vorbereitung der Ausführung und der frühzeitigen Erkennung von Fehlern. Dieses Kapitel erläutert die Ausführung des Programms.

Dazu wurde eine Bottom-Up Vorgehensweise gewählt. Dabei werden zunächst die untersten Aspekte der Ausführung analysiert und diese dann schrittweise in den größeren Zusammenhang eingebunden.

Bevor damit begonnen werden kann irgendetwas auszuwerten, benötigt man eine Datenstruktur, in der die Ergebnisse der Auswertungen hinterlegt werden können. Dazu wird der Datentyp *Value* definiert:

```
typedef struct {
    Type type;
    union {
        bool            as_bool;
        int             as_int;
        double          as_double;
        char            *as_String;
        rf_Set          *as_Set;
        rf_Relation     *as_R;
    };
} Value;
```

In diesem allgemeinen Datentyp können alle Datentypen aus RLang unter Verwendung eines geeigneten C Datentyps gespeichert werden. Die Datentypen für Mengen und Relationen stammen aus der RelaFix Bibliothek.

Zur eigentlichen Auswertung verwendet man wie schon bei der Analyse einen External Tree Walker, der unseren AST traversiert. Knoten des Typs N_BOOL, N_INT, N_FLOAT, N_STRING, N_SET und N_R aus Kapitel 3.4 haben keine eigenen Kindknoten sondern speichern lediglich eine Zeichenkette, die sie vom Parser erhalten haben.

Der erste Schritt der Ausführung ist nun, diese Zeichenketten in einen Value zu konvertieren. Dies wird im folgenden Beispiel dargestellt:

```
static
Value *
eval_int(const Ast *n_int) {
    Value *rval = value_new();
    value_set_int(rval, atoi(n_int->value));
    return rval;
}
```

Entscheidend ist hierbei die Funktion *atoi* aus der C Standardbibliothek, die eine Zeichenkette in einen *int* Wert konvertiert.

Der nächste Schritt ist die Auswertung der verschiedenen Operatoren. Dies sei exemplarisch am N_EQ Operator == gezeigt:

```
static
Value *
eval_eq(const Ast *n_eq) {
    Ast *op1 = n_eq->children[0];
    Ast *op2 = n_eq->children[1];

    Value *v1 = eval(op1);
    Value *v2 = eval(op2);
    Value *rval = v1;

    if(v1->type != v2->type) {
        value_set(rval, false);
    } else {
        switch(v1->type) {
        case T_BOOL:
            value_set_bool(rval, v1->as_bool == v2->as_bool);
            break;
        case T_INT:
            value_set_bool(rval, v1->as_int == v2->as_int);
            break;
        case T_FLOAT:
            value_set_bool(rval,
                abs((v1->as_float - v2->as_float)) < 0.05
            );
            break;
        case T_STRING:
            value_set_bool(rval,
                strcmp(v1->as_String, v2->as_String) == 0
            );
            break;
        case T_SET:
            value_set_bool(rval,
                rf_set_equal(v1->as_Set, v2->as_Set)
            );
            break;
        case T_R:
            value_set_bool(rval, false);
            break;
        }
    }
    value_free(v2);

    return rval;
}
```

Zunächst werden die beiden Operanden ausgewertet. Sind diese nicht vom gleichen Typ, kann direkt gesagt werden, dass die beiden Werte ungleich sein müssen. Sind beide Operanden aber typgleich, hängen die weiteren Schritte vom

jeweiligen Typ ab.

Für die beiden Typen T_BOOL und T_INT kann der native == Operator aus C zum Vergleich der Werte verwendet werden.

Bei Fließkommazahlen kann es aufgrund der internen Darstellung zu leichten Abweichungen kommen. Zwei Fließkommazahlen werden daher als gleich betrachtet, wenn sich diese um weniger als 0.05 unterscheiden. Dieser Wert ist an dieser Stelle willkürlich gewählt. Sollte er sich zu einem späteren Zeitpunkt als unpraktikabel erweisen, muss er nachträglich angepasst werden.

Zeichenketten werden mit Hilfe der *strcmp* Funktion aus der C Standardbibliothek verglichen.

Für den Vergleich zweier *Mengen* stellt RelaFix die rf_set_equal Funktion bereit.

Zum Vergleich von Relationen ist in RelaFix zum gegenwärtigen Zeitpunkt keine Vergleichsoperation implementiert.

Damit wurden nun alle Funktionen zur Auswertung einfacher Ausdrücke umgesetzt.

Der nächste Schritt ist die Zuweisung und Auswertung von Variablen, in denen das Ergebnis der ausgewerteten Ausdrücke gespeichert und wieder geladen werden kann. Dazu wird eine Datenstruktur benötigt, in dem der Wert gespeichert werden kann, der durch die Auswertung eines Ausdrucks erzeugt wird.

```
struct _memory {
    Symbol  *symbol;
    Value   *value;
};
```

Zur Zuordnung der Speicherbereiche zu den Variablen wird hier das Symbol der Variablen genutzt. Wie in Kapitel 4.1.1 beschrieben, sind die Variablen in verschachtelten Sichtbarkeitsbereichen gruppiert. Dies ist auch für die Speicherbereiche nachzubilden. Immer wenn ein Sichtbarkeitsbereich betreten wird, ist ein neuer Speicherbereich zu erstellen, in dem die Werte der Variablen hinterlegt werden können. Beim Verlassen des Sichtbarkeitsbereichs kann auch der dazugehörige Speicherbereich mit allen enthaltenen Werten gelöscht werden.

Wird eine Variable deklariert, wird dazu auch Speicher für den Wert erstellt und durch das Symbol identifiziert im Speicherbereich hinterlegt. Wird eine Variable zugewiesen, wird zunächst der Speicher der zu dem Symbol gehört im aktuellen Speicherbereich gesucht, der alte Wert gelöscht und der neue an diese Stelle geschrieben. Wird eine Variable gelesen, wird nur der Speicher gesucht und der enthaltene Wert zurückgegeben.

Bei Funktionsaufrufen wird zunächst einmal ein Speicherbereich benötigt, um die Parameter, die an die Funktion übergeben wurden, im Funktionskörper verfügbar zu machen. Jeder Parameter wird nun ausgewertet und als Variable im Speicherbereich der Funktionsparameter hinterlegt. Bei vordefinierten Funktionen werden die Parameter aus diesem Speicherbereich wieder ausgelesen und an eine native C Funktion übergeben. Bei benutzerdefinierten Funktionen wird zunächst ein weiterer Speicherbereich erstellt und dann die Ausführung mit dem Code aus der Funktionsdefinition fortgesetzt.

Erreicht eine solche benutzerdefinierte Funktion ein N_RETURN Statement muss die Ausführung des Programms unterbrochen und beim ursprünglichen

Funktionsaufruf fortgesetzt werden. Bei einem N_RETURN wird der Wert also nicht wie etwa bei den Operatoren einfach zurückgegeben, sondern in eine globale Variable geschrieben.

```
static
void
returnstat(const Ast *return_stmt) {
    Ast *expr = return_stmt->children[0];

    if(expr != NULL) {
        callstack_returnvalue = eval(expr);
    } else {
        callstack_returnvalue = value_new();
    }
}
```

Die sequenzielle Ausführung der RLang Statements innerhalb des N_BLOCK wird daraufhin unterbrochen.

```
static
void
block(const Ast *block_stmt) {
    MemorySpace *old_memspace = current_memspace;
    current_memspace = memspace_new(current_memspace);

    for(int i = 0; i < block_stmt->children.length &&
        callstack_returnvalue == NULL; i++) {

        exec(block_stmt->children[i]);
    }

    memspace_free(current_memspace);
    current_memspace = old_memspace;
}
```

Dadurch bewegt man sich im AST *aufwärts*, bis der N_CALL Knoten erreicht ist. Im Funktionsaufruf N_CALL wird der Wert ausgelesen, die globale Variable auf *NULL* zurückgesetzt und der Wert als Ergebnis der aufgerufenen Funktion zurückgegeben.

Die letzten noch zu implementierenden Funktionalitäten sind Verzweigungen und Schleifen. Da C selbst ebenfalls über Verzweigungen und Schleifen verfügt, ist die Implementierung relativ selbsterklärend wie hier am N_IF gezeigt wird.

```
static
void
ifstat(const Ast *if_stmt) {
    Ast *cond_expr = if_stmt->children[0];
    Ast *then_stmt = if_stmt->children[1];
    Ast *else_stmt = if_stmt->children[2];

    Value *cond;
    if((cond = eval(cond_expr))->as_bool) {
        exec(then_stmt);
    } else {
        exec(else_stmt);
    }
    value_free(cond);
}
```

Damit wurde eine Implementierung für alle im AST definierten Knoten erstellt.

Zusammenfassung: Zuerst wurde gezeigt, wie man mit Hilfe von Flex und Bison einen Lexer und einen Parser zum Einlesen der Befehle der Sprache erstellt. Dieser Parser generiert aus dem Programmcode einen AST. Auf Grundlage dieses AST wurden *Sichtbarkeitsbereiche* und *Symbole* erstellt und diese einer ersten semantischen Überprüfung unterzogen. Danach wurden allen Ausdrücken die jeweiligen *Typen* zugeordnet und diese wenn nötig in einen anderen *Typ* konvertiert. Mit Hilfe dieser *Typen* wurde im Rahmen einer zweiten semantischen Überprüfung die Korrektheit der vorhandenen Ausdrücke sichergestellt. Im letzten Schritt wurden dann im Interpreter die mit den RLang Befehlen verbundenen Aktionen ausgeführt.

Dies schließt die Entwicklung der RLang Sprache und des Interpreters im Rahmen der definierten Anforderungen ab.

5 Ausblick

5.1 Fehlerbehandlung

Ein Kapitel, das in der bisherigen Entwicklung von RLang nur im Grundsatz umgesetzt wurde, ist die Fehlerbehandlung.

Bei der Entwicklung eines Programmes ist es normal, dass der Programmcode nicht bereits im ersten Anlauf völlig fehlerfrei ist. Dies gilt insbesondere dann, wenn die Entwickler des Programms wenig Erfahrung mit der eingesetzten Programmiersprache besitzen, oder sie länger nicht benutzt haben. Die Suche nach Fehlern im Programmcode (den sogenannten "Bugs") ist nicht nur für den Entwickler äußerst anstrengend und mühsam, sondern verschlingt auch knappe Projektresourcen. Daher ist es wichtig, die Entwickler bei der Arbeit an ihrem Programm und der Suche nach Fehlern zu unterstützen.

Die erste Stelle an der Fehler auftreten können ist der syntaktische Aufbau des Programms. Zwar bietet der von RLang eingesetzte Bison-Parser bereits eine rudimentäre Fehlerausgabe, diese gibt jedoch nur mögliche erwartete Token und das tatsächlich erhaltene Token aus. Dies richtet sich jedoch eher an den Entwickler der von Bison genutzten Grammatik. Dem Entwickler eines RLang Programmes bietet diese Ausgabe hingegen nur wenig Hilfestellung bei der Suche nach einem Fehler, da insbesondere die Zeilennummer und der tatsächliche Text in dem der Fehler aufgetreten ist, von Bison nicht ausgegeben werden. Zu diesem Zweck bietet Bison einige Hilfsmittel, mit denen sich eine eigene ausführlichere Fehlerausgabe implementieren lässt. (siehe dazu [FSF99, Kapitel 4.7 The Error Reporting Function *yyerror*])

Eine Fehlerausgabe sollte wenn möglich die Zeilennummer und die fehlerhafte Zeile selbst ausgeben. Zusätzlich wäre es darüber hinaus möglich, den genauen Bereich des Fehlers innerhalb der Zeile einzugrenzen. (siehe dazu [FSF99, Kapitel 1.6 Tracking Locations] Ergänzt durch eine entsprechende Fehlermeldung wäre es damit leicht möglich, schnell den genauen Ort eines syntaktischen Fehlers ausfindig zu machen.

Den nächsten wichtigen Punkt stellen die Variablen- und Funktionssymbole dar. Zwar führt RLang hier bereits viele Prüfungen durch, dennoch ließe sich auch hier noch etwas verbessern und für Entwickler von RLang Programmen klarer gestalten. Bereits durchgeführt werden Prüfungen auf folgende Fehler:

- Verwendung von unzulässigen Namen bei der Deklaration (Keywords, Typbezeichner, ...)
- Verwendung von undefinierten Variablen
- Verwendung von undefinierten Funktionen
- Verwendung von nicht-zugewiesenen Variablen
- Deklaration von im aktuellen Sichtbarkeitsbereich bereits deklarierten Variablen
- Deklaration von Variablen, die nie gelesenen werden

Bisher wird aber nur auf der Konsole ausgegeben, dass ein solcher Fehler gefunden wurde. Eine Fehlerausgabe mit Zeilennummer und Zeile selbst wäre wünschenswert.

Der letzte Punkt der Fehlererkennung ist die Überprüfung der Typen. Hier führt RLang alle notwendigen Tests durch und gibt die Fehlerhaften Ausdrücke an, dennoch wäre eine Fehlerausgabe, die dem Entwickler die genaue Position des Fehlers nennt, eine nützliche Verbesserung.

5.2 Synax für Mengen und Relationen

Die Anforderungen die an RLang im Rahmen dieser Bachelorarbeit gestellt wurden, sehen die Anbindung von RLang lediglich als externe Funktionen vor. Dies ist jedoch nicht zuletzt nur dem begrenzten Zeitumfang des Projektes geschuldet. In einem Folgeprojekt wäre es möglich, RLang um diverse Sprachkonstrukte zu erweitern, die die Arbeit mit Mengen und Relationen vereinfachen und beschleunigen.

Die folgenden Codefragmente sollen Beispiele für verschiedene syntaktische Konstrukte sein. Sie sind in der Version von RLang, die im Rahmen dieser Bachelorarbeit entsteht, nicht implementiert und dienen lediglich als Anhaltspunkte für künftige Entwicklungsprojekte.

Der RLang Code zur Definition einer Menge könnte beispielsweise wie folgt aussehen:

```
Set boolean := { 0 1 };
```

Hier würde eine Variable definiert, die auf eine Menge mit den zwei Symbolen für eine boolesche Algebra verweist. Die einzelnen Elemente werden dabei durch Leerzeichen voneinander getrennt. Zu beachten ist dabei, dass die Elemente keinen eigenen Datentyp haben, also keine Unterscheidung zwischen der Zahl 1 und der Zeichenkette '1' stattfindet.

Die Elemente der Menge können durchaus längere Bezeichner haben, eine Definition wie die Folgende ist also ebenfalls möglich:

```
Set boolean := { null eins };
```

Mit dieser Notation ist es zudem möglich, Mengen ineinander zu verschachteln, wie das folgende Beispiel veranschaulichen soll:

```
Set exampleset := { 0 1 { 2 3 } 4 {} };
```

Diese Menge würde also die Symbole 0 1 und 4 beinhalten, sowie eine Menge mit den Symbolen 2 und 3. Das letzte Element der Menge ist ein Beispiel für die Notation der leeren Menge $\emptyset$.

Zur Bestimmung der Kardinalität einer Menge könnten die aus der mathematischen Notation entlehnten | Symbole dienen:

```
int i := |exampleset|; // i = 5
```

Zur Definition einer Relation eignen sich, abhängig von der Relation, verschiedene Darstellungsformen. Die tabellarische Definition einer homogenen Relation könnte wie folgt aussehen:

```
Set onetwothree := { 1 2 3 };
R onetwothree_lt :=
    onetwothree x onetwothree : { 0 1 1 || 0 0 1 || 0 0 0 };
```

onetwothree x onetwothree wären in diesem Fall die Domänen der Relation, die zu Beginn definiert worden sind. Eine Definition der Mengen innerhalb der Relation sollte aber auch möglich sein:

```
R onetwothree_lt :=
    { 1 2 3 } x onetwothree : { 0 1 1 || 0 0 1 || 0 0 0 };
```

Auf die Definition der Domänen folgt die Wertetabelle, der zu entnehmen ist, ob zwei Elemente in Relation zueinander stehen (Wert 1) oder nicht (Wert 0). Die Spalten der Tabelle werden wie die Elemente der Mengen mit Leerzeichen voneinander getrennt, während Zeilen durch || voneinander getrennt werden. Wie oben gezeigt ermöglicht dies eine kompakte Darstellung als auch eine übersichtliche, mit Kommentaren versehene Darstellung:

```
R onetwothree_lt :=
    onetwothree x onetwothree : /* 1 2 3 */
                          /* 1 */ { 0 1 1 |
                          /* 2 */ | 0 0 1 |
                          /* 3 */ | 0 0 0 };
```

5.3 Optimierung / LLVM

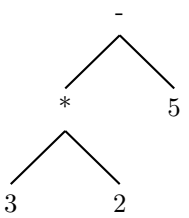
Ein weiterer Punkt in der Reihe der naheliegenden Verbesserungen für RLang ist die Optimierung der Ausführung. Zwar ist die Ausführungsgeschwindigkeit von in RLang entwickelten Programmen voraussichtlich eher zweitrangig, dennoch sollte man das Thema nicht von vornherein völlig ignorieren.

Der Ansatzpunkt für Optimierungen ist der in einem Zwischenschritt durch den RLang Interpreter generierte AST.

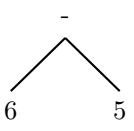
Am Ausdruck $(3 * 2 - 5) * a + a$ können hier einige mögliche Optimierungsschritte veranschaulicht werden. Wie in Kapitel 3.3 genauer erläutert, würde dieser Ausdrück zunächst in einen abstrakten Syntaxbaum überführt, an dem die folgenden Optimierungschritte durchgeführt werden können:

Der erste Schritt ist die Reduktion konstanter Ausdrücke. Diese finden sich häufig in Programmen, wenn der Entwickler beabsichtigt, Rechnungen nachvollziehbarer zu gestalten. Beispielsweise wäre ein für sich genommener Wert wie 9.8596 weniger Aussagekräftig als der Ausdruck 3.14**2, bei dem auf den ersten Blick deutlich wird, dass es sich um π^2 handelt.

Bezogen auf das Beispiel existieren für diesen Optimierungsschritt zwei Anwendungsfälle. Diese befinden sich im Teilbaum



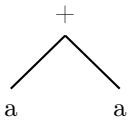
Da beide Operanden des $*$-Baums konstant sind, kann man mit Sicherheit sagen, dass das Ergebnis dieser Rechnung stets 6 sein wird. Daher kann der $*$-Baum durch einen Knoten mit der Konstanten 6 ersetzt werden.



Nach Anwendung dieser Regel erfüllt auch der entstandene Teilbaum die Bedingung zweier konstanter Operanden. Daher kann der gesamte ursprüngliche Teilbaum zur Konstante 1 reduziert werden.

Der nächste Schritt ist die Entfernung neutraler Operationen. Dies beschreibt beispielsweise eine Addition mit 0, aber auch einige boolesche Operationen wie $b \mid\mid false$. In diesem Fall ist die neutrale Operation der Ausdruck $1 * a$, der sich zum Ausdrück a reduzieren lässt.

Alles was vom ursprünglichen Baum übrig bleibt, ist der Teilbaum



An diesem Baum kann nun noch ein letzter Schritt durchgeführt werden. Dazu wird der Ausdruck $a+a$ durch den Ausdruck $a*2$ ersetzt. Die Optimierung erschliesst sich hier nicht auf den ersten Blick, da anders als bei den vorherigen Schritten, keine Knoten wegfallen. Der Vorteil der eingesetzten Operation findet sich erst im Interpreter. Dieser muss nämlich beim Ausdruck $a + a$ zwei mal die Variable a auflösen, während beim Ausdruck $a * 2$ die Variable nur ein einziges Mal aufgelöst werden muss. Da das Ausflösen der Variable deutlich aufwendiger ist als die Multiplikation mit der Zahlenkonstante 2, wird die Auswertung weiter optimiert.

Nun sind diese 3 Operationen bei weitem nicht die einzigen Optimierungen, die man an einem Programm vornehmen kann. Bevor jedoch an weiteren Optimierungsmöglichkeiten im Detail gearbeitet wird, sollte ein Blick über den Tellerrand geworfen werden.

In Kapitel 3.3 wurde beschrieben, wie ausgehend vom RLang spezifischen Syntaxbaum eine abstraktere Darstellung geschaffen werden kann. Die Motivation hinter der Abstrahierung eines vorliegenden Problems ist üblicherweise die Möglichkeit, bekannte, allgemeine Lösungsstrategien auf das Problem anwenden zu können. Ein Projekt, das Lösungsstrategien zur Optimierung von Programmiersprachen anbietet, ist das LLVM Projekt. Das LLVM Projekt ist eine modulare Compiler-Unterbau-Architektur mit übergreifend optimierendem Übersetzungskonzept [Lat02]. Seit seiner Vorstellung wurde LLVM kontinuierlich weiterentwickelt und beinhaltet heute eine Vielzahl von Unterprojekten und Erweiterungen aus der aktuellen Compilerforschung und -entwicklung. Um die Vorteile dieser Entwicklungen nutzbar zu machen, muss lediglich der RLang AST in einen LLVM AST überführt werden. Dabei ist zunächst zu klären, ob und in welchem Umfang die von LLVM angebotenen Funktionalitäten genutzt werden sollen. Die Umstellung auf den von LLVM bereitgestellten Compilerunterbau ist jedoch ein vielversprechender Ansatz, der in einem Folgeprojekt näher untersucht werden sollte.

6 Zusammenfassung und Fazit

In der vorliegenden Arbeit wurde eine erste Implementierung einer universellen Programmiersprache für Relationen entwickelt.

In Kapitel 1 wurde dazu die vorliegende Problemstellung betrachtet und daraus die Ziele des Projektes entwickelt. In Kapitel 2 wurden diese Ziele analysiert, um daraus die weitere Vorgehensweise abzuleiten. Anhand dieser Vorgehensweise sind in Kapitel 3 die grundlegenden Entwurfsentscheidungen getroffen und spezifiziert worden. Aufbauend auf diesem Entwurf wurde dann in Kapitel 4 dargestellt, wie die konkreten Ziele des Projektes implementiert worden sind. Diese Implementierung bietet zwar bereits viele nützliche Funktionen, bietet jedoch auch viel Potenzial für weitere Entwicklungen, wie der relativ umfangreiche Ausblick in Kapitel 5 zeigt.

Literatur

[Ber12] Peter Berger. "Entwicklung eines Programmes und zugehöriger Sprache zur Analyse von Relationen". Hochschule Bonn Rhein-Sieg, 2012.

[FSF99] Inc. Free Software Foundation. *Bison*. 1999. URL: `https://www.gnu.org/software/bison` (besucht am 24. 08. 2013).

[Lat02] Chris Lattner. "LLVM: An Infrastructure for Multi-Stage Optimization". *See* `http://llvm.cs.uiuc.edu`. Magisterarb. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, 2002.

[Par09] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. 1st. Pragmatic Bookshelf, 2009. ISBN: 193435645X, 9781934356456.

[VW06] Gottfried Vossen und Kurt-Ulrich Witt. *Grundkurs theoretische Informatik - eine anwendungsbezogene Einführung für Studierende in allen Informatik-Studiengängen (4. Aufl.)* Vieweg, 2006, S. I–XII, 1–433. ISBN: 978-3-8348-0153-1.