**Hochschule**
**Bonn-Rhein-Sieg**
*University of Applied Sciences*

**Fachbereich Informatik**
*Department of Computer Science*

# Abschlussarbeit

im Bachelor Studiengang Computer Science

## Entwicklung eines Programms und zugehöriger Sprache zur Analyse von Relationen

**von**
**Peter Berger**

13. Februar 2012

Erstprüfer: Prof. Dr. Martin Eric Müller
Zweitprüfer: Prof. Dr. Peter Becker

Eingereicht am:

## Eidesstattliche Erklärung

Ich versichere an Eides statt, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

(Ort, Datum, Unterschrift)

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Listings

## Abkürzungsverzeichnis

| | |
|---|---|
| BNF | Backus-Naur-Form |
| DSL | domain-specific language |
| FSF | Free Software Foundation |
| GCC | GNU Compiler Collection |
| GPL | General Public License |
| MSC | Microsoft C |
| SVN | Subversion |

# 1 Einleitung

In dieser Arbeit wird es darum gehen ein Programm zu entwerfen und zu implementieren. Dieses Programm wird als eine Art Taschenrechner dienen und Befehle aus einer domänenspezifischen Sprache akzeptieren. Diese Sprache wird ebenso in dieser Arbeit entworfen und spezifiziert.

## 1.1 Problemstellung

In verschiedenen Bereichen der Informatik, sowie Mathematik, ist die Betrachtung von Relationen und deren Eigenschaften notwendig. Zum Beispiel bei der Arbeit mit relationalen Programmen oder vorhandenen Relationen. Bei der Findung einer Lösung müssen oft verschiedene Relationsvarianten auf ihre Eigenschaften überprüft werden. Dies ist jedoch problematisch, da bei komplexen Relationen die Analyse per Hand sehr zeitaufwendig und fehleranfällig ist.

Daher wird ein Programm benötigt, das den Nutzer bei diesen Aufgaben unterstützt und leicht zu anzuwenden ist. Des weiteren soll es in C geschrieben werden um es auf möglichst vielen Plattformen (Linux, Windows, ...) einsetzen zu können.

## 1.2 Ziel

In dieser Abschlussarbeit soll eine erste lauffähige Version eines Programms entstehen, welches dem Informatiker oder Mathematiker bei den in der Problemstellung beschriebenen Tätigkeiten unterstützt. Eine eigene domänenspezifische Sprache soll dabei die Bedienung einfach halten. Die folgenden Anforderungen werden an die erste Version des Programms gestellt.

| Zuordnung | Nr. | Anforderung |
|---|---|---|
| Programm | 1 | Implementierung in C |
| | 2 | Einfache Portierung auf andere Plattformen |
| | 3 | Konsolenanwendung (keine graphische Benutzeroberfläche) |
| | 4 | Eingabe erfolgt über das Einlesen einer Datei. |
| | 5 | Ausgabe in Konsole |
| | 6 | Informative Fehlermeldung bei falscher Eingabe |
| | 7 | Implementiert die folgende domänenspezifische Sprache |
| Domänen- | 8 | Definieren von Domänen durch Nutzer |
| spezifische | 9 | Löschen von Domänen durch Nutzer |
| Sprache | 10 | Definieren von Relationen als Tabelle durch Nutzer |
| | 11 | Löschen von Relationen durch Nutzer |
| | 12 | Definieren von Operationen als Tabelle durch Nutzer |
| | 13 | Definieren von Operationen als Formel mit Variablen durch Nutzer |
| | 14 | Löschen von Operationen durch Nutzer |
| | 15 | Definieren von Negationen durch Nutzer |
| | 16 | Löschen von Negationen durch Nutzer |
| | 17 | Hinzuladen von Dateien dieser Sprache durch Nutzer |
| | 18 | Ändern des Arbeitsverzeichnis durch Nutzer |
| | 19 | Kein festlegen von Domänen für die Berechnung von Ausdrücken |
| | 20 | Ausdrücke sind schachtelbar |
| | 21 | In Ausdrücken können Funktionen aufgerufen werden |
| | 22 | Es kann die Beschreibung einer Funktion |

| | | |
|---|---|---|
| | | ausgegeben werden |
| | 23 | Der Nutzer kann prüfen ob eine Relation aRb existiert |
| | 24 | Der Nutzer kann Relationen auf Antisymmetrie prüfen |
| | 25 | Der Nutzer kann Relationen auf Asymmetrie prüfen |
| | 26 | Der Nutzer kann Relationen auf Difunktionalität prüfen |
| | 27 | Der Nutzer kann Relationen auf Äquivalenz prüfen |
| | 28 | Der Nutzer kann Relationen auf Irreflexivität prüfen |
| | 29 | Der Nutzer kann Relationen auf „poset“ Eigenschaft prüfen |
| | 30 | Der Nutzer kann Relationen auf „preorder“ Eigenschaft prüfen |
| | 31 | Der Nutzer kann Relationen auf Reflexivität prüfen |
| | 32 | Der Nutzer kann Relationen auf Symmetrie prüfen |
| | 33 | Der Nutzer kann Relationen auf Transitivität prüfen |
| | 34 | Der Nutzer kann das Komplement einer Relation bilden |
| | 35 | Der Nutzer kann das Inverse einer Relation bilden |
| | 36 | Der Nutzer kann die Vereinigung von zwei Relationen bilden |
| | 37 | Der Nutzer kann den Schnitt von zwei Relationen bilden |
| | 38 | Generierung der „meet“ Funktion auf Basis einer Kleinergleichrelation |
| | 39 | Generierung der „join“ Funktion auf Basis einer Kleinergleichrelation |
| | 40 | Erzeugung einer Relation mit $xRx, \forall x$ (id) |
| | 41 | Erzeugung einer leeren Relation |
| | 42 | Erzeugung einer Relation in der alle $xRy$ existieren |
| | 43 | Concatenation zweier Relationen |
| | 44 | Finden des Maximums einer Kleinergleichrelation |
| | 45 | Finden des Minimums einer Kleinergleichrelation |
| | 46 | Finden der oberen Schranke von Elementen einer $\leq$ Relation |
| | 47 | Finden der unteren Schranke von Elementen einer $\leq$ Relation |
| | 48 | Finden des Supremums |
| | 49 | Finden des Infimums |
| Dokumentation | 50 | Code enthält wenn nötig Kommentare |
| | 51 | Dokumentation von Funktionen und Strukturen |
| | 52 | Spezifikation der domänenspezifischen Sprache |
| Alternativ | 53 | Interaktive Eingabe |

Die Spezifikation der domänenspezifischen Sprache wird in diesem Dokument erfolgen.

### 1.3 Methode

Das Projekt wird anhand des Phasenmodells durchgeführt. Es wird in die 4 Hauptphasen Analyse, Entwurf, Implementierung und Evaluation unterteilt. Jede dieser Phasen kann selbst auch unterteilt sein. Am Ende jeder Hauptphase steht ein Meilenstein. Es darf erst mit der nächsten Phase begonnen werden, wenn der Meilenstein der vorherigen Phase erreicht wurde. Sollte ein Meilenstein zum festgelegten Datum nicht erreicht worden sein, so muss man wenn möglich Anforderungen mit niedriger Priorität wegfallen lassen um am Ende wenigstens ein abgeschlossenes Projekt zu haben. Auch wenn es dann nicht alle Anforderungen erfüllt.

Eine Aufteilung der Phasen und ihre zeitliche Einteilung findet sich im Gantt-Diagramm auf Seite 58 im Anhang.

# 2 Analyse

In diesem Abschnitt der Arbeit werden alle die Elemente näher betrachtet, die zur Lösung der Problemstellung nötig sind. Dazu gehören mathematische Konstrukte genauso wie Programme.

## 2.1 Mathematische Begriffe

### 2.1.1 Domain

Eine Domain wird oft auch als Menge bezeichnet. Sie ist die Zusammenfassung von Elementen. Man sagt auch: „Eine Menge enthält Elemente." Enthält eine Menge keine Elemente, so spricht man von einer leeren Menge. Des weiteren können auch Mengen selbst ein Element einer anderen Menge sein.

Mathematisch wird eine Menge wie folgt beschrieben:

$$M = \{a, b, c\}$$

Wobei M der Name der Menge ist und a, b und c die Elemente der Menge sind.

### 2.1.2 Algebraische Strukturen

Eine algebraische Struktur besteht aus einer Menge die nicht leer ist und Operationen die auf dieser Menge operieren. Das Ergebnis einer solchen Operation muss wieder ein Element der selben Menge sein.

Ein Beispiel für eine algebraische Struktur ist die boolesche Algebra: $(M, and, or, not)$ mit $M = \{0, 1\}$

Von einer relationalen Algebra spricht man, wenn die Menge aus einem Satz von Relationen besteht und die Operationen Schnitt, Vereinigung, Konkatenation, Negation, Transposition, größtes Element, kleinstes Element und 1 definiert sind.(Müller 2012, Seite 32)

Bekannte algebraische Strukturen sind:

- Boole
- Heyting
- Diamant
- Doppel Diamant

### 2.1.3 Relation

Eine Relation ist eine Teilmenge eines kartesischen Produkts. Sie gibt an ob eine Beziehung zwischen den Elementen der Mengen besteht. Die mathematische Schreibweise sieht wie folgt für eine zweistellige Relation aus: $R \subseteq A \times B$ Des weiteren kann eine Relation zwischen zwei Elementen $x \in A$ und $y \in B$ durch $R(x, y)$ oder $xRy$ ausgedrückt werden. Auch ist zu beachten, dass $xRy \neq yRx$ ist.

Zweistellige Relationen lassen sich gut in einer booleschen Tabelle erfassen, wobei die Zeilen die Elemente der ersten Menge repräsentieren und die Spalten die Elemente der zweiten Menge.(Witt 2007, Seite 87)

Besondere Relationen sind die komplementäre Relation und konverse Relation. Sie werden jeweils in Bezug zu einer original Relation betrachtet. Wenn $R$ die originale Relation ist, so beinhaltet die konverse Relation $\breve{R}$ folgende Paare $x\breve{R}y \Leftrightarrow yRx$. Und die komplementäre Relation $\bar{R}$ enthält die Paare $x\bar{R}y \Leftrightarrow \neg(xRy)$. (Müller 2012, Seite 25)

Das Bilden einer Relation $R_1 : A \to C$ aus den Relationen $R_2 : A \to B$ und $R_3 : B \to C$ nennt man Konkatenation. Hierbei gilt: $\forall xR_1z \to xR_2y \wedge yR_3z$

Eine spezielle Gruppe von Relationen sind solche zweistelligen Relationen, die Elemente aus der selben Menge in Beziehung setzen. Sie werden als homogen bezeichnet. $R \subseteq A \times A$. Für diese Gruppe der Relationen wurden einige Eigenschaften definiert, um sie besser Klassifizieren zu können. Folgend eine Liste der wichtigsten Eigenschaften:

| | | |
|---|---|---|
| Reflexivität | wenn | $xRx$ |
| Symmetrie | wenn | $xRy \to yRx$ |
| Antisymmetrie | wenn | $xRy \wedge yRx \to x = y$ |
| Transitivität | wenn | $xRy \wedge yRz \to xRz$ |
| Difunktionalität | wenn | $xRy \wedge zRy \wedge zRw \to xRw$ |
| Äquivalenz | wenn | $symmetrisch \wedge transitiv \wedge reflexiv$ |
| Halbordnung (poset) | wenn | $reflexiv \wedge transitiv \wedge antisymmetrisch$ |

$\forall \quad w, x, y, z \in A$
(Müller 2012, Seite 25f.)

Quasiordnung (preorder) wenn $reflexiv \wedge transitiv$
(Berghammer 2008, Seite 6)

$M$ ist eine Ordnung und $N \subseteq M$ und $a \in M$

| | | |
|---|---|---|
| $a$ ist obere Schranke (upperbound) von $N$ | wenn | $\forall b \in N, b \leq a$ |
| $a$ ist untere Schranke (lowerbound) von $N$ | wenn | $\forall b \in N, a \leq b$ |
| $a$ ist Maximum von $M$ | wenn | $\forall b \in M \setminus a, a > b$ |
| $a$ ist Minimum von $M$ | wenn | $\forall b \in M \setminus a, a < b$ |
| $a$ ist Infimum von $N$ | wenn | a größtes Element der unteren Schranke von $N$ ist. |
| $a$ ist Supremum von $N$ | wenn | a kleinstes Element der oberen Schranke von $N$ ist. |

(Berghammer 2008, Seite 8f.)

| | | |
|---|---|---|
| Irreflexivität | wenn | $xRy \to x \neq y$ |
| Asymmetrie | wenn | $xRy \to \neg yRx$ |

$\forall \quad x, y \in A$
(Berghammer 2008, Seite 212)

### 2.1.4 Operation

Eine Operation ist eine Verknüpfung von n Elementen aus Mengen, deren Ergebnis wiederum ein Element ist. Die Elemente können, wenn die Operation so definiert wurde, aus verschiedenen Mengen stammen. Folgend die mathematische Schreibweise zur Definiti-

on einer Operation die zwei Elemente miteinander verknüpft:

$$f : A \times B \to C$$

Wobei hier f der Name der Operation ist, welche ein Element aus der Menge A mit einem Element der Menge B verknüpft. Das Ergebnis muss ein Element aus der Menge C sein.(Witt 2007, Seite 99)

Wichtige Operationen, die in den bekannten algebraischen Strukturen verwendet werden sind: Negation, Vereinigung, Schnitt, Meet und Join.

#### Negation

Die Negation ist eine Operation auf einem Element und hat die Form $f : A \to A$. Die bekannteste Form der Negation ist die boolesche, welche als $\neg 1 = 0$ und $\neg 0 = 1$ definiert ist. Komplizierter wird die Definition schon für die Heytingalgebra mit 3 Elementen $\{0, a, 1\}$. Sie ist wie folgt definiert: $\neg 0 = 1$, $\neg a = 0$ und $\neg 1 = 0$.

#### Vereinigung

Eine Vereinigung, im Englischen als union bezeichnet, ist eine Operation auf Mengen, deren Ergebnis eine Menge ist die alle Elemente aus den Eingabemengen enthält.

#### Schnitt

Ein Schnitt, im Englischen als intersection bezeichnet, ist eine Operation auf Mengen, deren Ergebnis eine Menge ist die nur die Elemente enthält, welche auch in allen Eingabemengen enthalten sind.

#### Meet

Meet ist eine Operation auf einer Ordnungsrelation. Ordnungsrelation bedeutet, das die Relation die Eigenschaften Reflexivität, Antisymmetrie und Transitivität besitzt. Eine bekannte Ordnungsrelation ist die $\leq$ Relation. Die Meet Operation liefert auf Basis einer solchen Ordnungsrelation für 2 Elemente das nächste kleinere gemeinsame eindeutige Element.

#### Join

Wie die Meet Operation basiert auch die Join Operation auf einer Ordnungsrelation. Jedoch liefert diese für 2 Elemente das nächste größere gemeinsame eindeutige Element.

### 2.2 Domänenspezifische Sprache

Wie in Kapitel 1.2 bereits erwähnt, soll die Bedienung des Programms über eine domänenspezifische Sprache geschehen. Was aber bedeutet dies nun genau?

Die meisten Informatiker kennen Sprachen wie C, Java, Fortran usw. welche auch als Universalsprachen bezeichnet werden. Diese Bezeichnung kommt daher, da sie zur Lösung

jeglicher Aufgabenstellung verwendet werden können. Auf Grund ihrer Komplexität sind die Lösungen meist auch komplex und um diese Sprachen verstehen und nutzen zu können ist eine längere Einarbeitungszeit nötig. Da es wünschenswert ist ein immer wiederkehrendes Problem einfach und schnell lösen zu können, existieren die domänenspezifischen Sprachen. Sie setzten genau hier an indem sie nur Probleme aus einem abgegrenzten Bereich lösen können. Dadurch reduziert sich die Komplexität der Lösung und die Sprache wird für den Anwender verständlicher.

„The basic idea of a domain specific language (DSL) is a computer language that's targeted to a particular kind of problem, rather than a general purpose language that's aimed at any kind of software problem." (Fowler 24.01.2012, `http://www.martinfowler.com/bliki/DomainSpecificLanguage.html`)

### 2.3 Parsergenerator

Ein Parser wird benötigt um die Eingaben des Nutzers, welche den Regeln der DSL des Programms RelaFix entsprechen müssen, auswerten zu können. Ein Parser wird meist mit Hilfe von Werkzeugen erstellt, so genannten Parsergeneratoren. Der Vorteil ist, dass mit relativ wenig Aufwand ein Parser für eine komplexe Sprache generiert werden kann. Des weiteren wird der Generator die Spracheigenschaften prüfen und Warnungen bei Mehrdeutigkeit ausgeben. Auch darf angenommen werden, dass der generierte Parser keine Fehler enthält. Der Einsatz eines Parsergenerators ist also sinnvoll. Folgend eine Auflistung bekannter Parsergeneratoren:

- ANTLR
- Bison
- Coco/R
- GOLD
- Yacc
- ...

Welcher Parsergenerator kommt für dieses Projekt in Frage?

Um dies zu Entscheiden müssen erst Anforderungen festgelegt werden, die der Generator erfüllen soll. Diese sind:

- generierter Parser muss in der Sprache C sein
- Parsergenerator sollte auf möglichst alle Systeme portierbar sein
- generierte Parser muss frei lizenzierbar sein
- Parsergenerator muss ausgereift sein (Keine Fehler)
- kontextfreie Grammatik

Der Parser der diese Anforderungen erfüllt ist Bison. Er ist unter der GPL lizenziert und somit OpenSource. Da der Quellcode frei zugänglich und in C geschrieben ist, ist es auch leicht zu portieren. Der generierte Parser enthält zur Lizenzierung einen extra Abschnitt, der es erlaubt ihn unter beliebiger Lizenz zu verwenden. Des weiteren ist Bison auch ausgereift. Es wird seit 1984 entwickelt und die neueste Version erschien im Mai 2011. Er beherrscht kontextfreie Sprachen. (Free Software Foundation Inc., 24.01.2012, `http://www.gnu.org/software/bison/`)

# 3 Entwicklung

## 3.1 Allgemein

Zur Entwicklung des Programms wird auf die Compiler gcc sowie MSC2010 zurückgegriffen. Dadurch sollte der entstehende Code auch zu anderen Compilern größtenteils kompatibel sein.

Zur Verwaltung des Projekts wird das Programm SVN eingesetzt. Man speichert immer nach dem Fertigstellen eines neuen Features das Projekt mit SVN. Durch den Einsatz dieses Programms ist es dann möglich zu jeder vorherigen Version zurückzukehren, falls man feststellt das man einen Fehler gemacht hat. Zudem kann dieses Vorgehen auch bei der Fehlersuche nützlich sein. Man kann herausfinden ab welcher Version ein Fehler auftritt und kann anhand der Änderungen dann den Fehler lokalisieren.

Bei der Programmierung mit Zeigern, passiert es öfters, dass vergessen wird Speicher freizugeben oder noch schlimmer, dass auf ungültigen Speicher zugegriffen wird. Um dem entgegen zu wirken wird das Programm DrMemory eingesetzt, mit dem diese Art von Fehlern entdeckt werden können. So kann zumindest sichergestellt werden, dass das Programm im fehlerfreien Betrieb keine Speicherlecks hat.

## 3.2 Sprachentwurf

Um möglichst vielen Personen einen Nutzen des Programms RelaFix zu ermöglichen, werden die Schlüsselwörter der domänenspezifischen Sprache in Englisch gehalten. Die Sprache wird sich an der mathematischen Schreibweise orientieren.

### 3.2.1 Elemente

Die Probleme die mit dieser Sprache gelöst werden sollen sind das Herausfinden der Eigenschaften einer Relation, sowie das Erzeugen neuer Relationen auf der Basis einer anderen (Konverse, Komplement). Auch sollen Operationen definiert und angewendet werden können. Es werden also folgende Elemente benötigt:

- Definieren von Mengen, Negationen, Relationen und Operationen
- Löschen von Mengen, Negationen, Relationen und Operationen
- Mathematische Ausdrücke um mit Operationen rechnen zu können
- Funktionen mit Parametern und Rückgabewert um Eigenschaften abfragen oder neue Relationen (Konverse, Komplement) erzeugen zu können.

Da die Verwendung von 3 oder höherstelligen Relationen selten ist, wird sich diese Version auf 2 stellige Relationen beschränken. Dies erlaubt zudem für das Testen von Relationen oder die Berechnung von Operationen die Infix Schreibweise zu verwenden. Dies wird die Übersichtlichkeit fördern.

Weitere Punkte die aus der Anforderungsliste hervorgehen sind:

- Hinzuladen von Dateien
- Festlegen des Arbeitsverzeichnis aus dem die Dateien geladen werden

### Dateiformat

Da das Programm sehr portabel sein soll, wird auch die Sprache darauf ausgelegt sein müssen. Dies bedeutet, das RelaFix -Dateien im ASCII Zeichensatz geschrieben sein müssen, da dieser Zeichensatz auf allen bekannten, neuen und alten Systemen umgesetzt ist.

Als Postfix für Dateien wird die Zeichenkette rfc (RelaFixCode) verwendet. Die Verwendung ist jedoch nicht zwingend.

### Whitespaces

Whitespaces sind die Zeichen in einer Computersprache, welche Tokens voneinander abtrennen. Es spielt keine Rolle ob zwischen 2 Token 1 oder mehrere Whitespaces stehen. Diese Sprache wird als Whitespaces Leerzeichen, Tabulator und das neue Zeilen Zeichen akzeptieren. Wobei eine neue Zeile eine der folgenden 4 Möglichkeiten ist. \n\r, \r\n, \r oder \n.

### Kommentare

Wie in jeder Computersprache soll es auch in dieser Sprache Kommentare geben. Sie werden mit dem Zeichen # gestartet und gehen ab diesem Zeichen bis zum Ende der Zeile.

### Befehl

Die RelaFix -Sprache wird aus Befehlen bestehen die hintereinander abgearbeitet werden. Um beim Parsen eine eindeutige Trennung zwischen Befehlen zu haben, wird ein Befehl mit „;“ beendet. Diese Vorgehensweise ist aus Sprachen wie C, Java usw. bekannt.

⟨ Befehl 1⟩; ⟨Befehl 2⟩;
⟨Befehl 3⟩;

### Id

Eine Id ist ein Token, welches den Bezeichner für eine Variable angibt. In der RelaFix -Sprache sind Zeichenketten bestehend aus den Zeichen a..z, A..Z, 0..9 sowie _ als Id erlaubt. Diese Zeichen können auch mit den gängigen Tastaturen in unterschiedlichen Ländern direkt eingegeben werden.

### Definitionen

Die Idee hinter einer domänenspezifischen Sprache ist es möglichst dem Anwendungsgebiet zu entsprechen. Daher werden die Definitionen der mathematischen Schreibweise angenähert. Des weiteren müssen die Bezeichner für die zu definierenden Objekte eindeutig sein. Ist ein Objekt definiert, so gilt dessen Bezeichner global und kann nicht für ein anderes Objekt genutzt werden!

Folgend die Formen der Definitionen:

Domain

DOMAIN $Bezeichner$ AS $Ausdruck$ ;

Bezeichner ist eine freie Id. Über diesen Bezeichner kann später auf diese Domain verwiesen werden. Ausdruck ist eine mathematische Berechnung, sie wird später erklärt. Hier sei jedoch angemerkt, dass eine Domain in einem Ausdruck temporär beschrieben werden kann. Der Rückgabewert eines Ausdrucks muss hier also eine temporäre Domain sein. Eine temporäre Domain wird wie folgt in einem Ausdruck definiert:

$\{Bezeichner1, .. , BezeichnerN\}$

Bezeichner ist hier eine Id, welche als Element der Domain aufgefasst wird. Der Bezeichner darf also auch der Name eines global definierten Objekts sein, wird aber nicht als dieses Objekt interpretiert.

Auch das Anlegen einer Domain als Element einer Domain ist erlaubt. Es ist jedoch zu beachten, dass für eine solche ein eigener Bezeichner angegeben werden muss. Dies geschieht in der Form

$\{Bezeichner : \{Bezeichner\}\}$

Möchte man eine global definierte Domain als Element einer Domain angeben, so schreibt man

$\{:Bezeichner\}$

Folgend ein Beispiel für eine gültige Definition:

```
DOMAIN boolean AS {0,1};
```

Negation

NEGATION $Bezeichner : Domain$ AS $E$ / $N$, .. , $E$ / $N$;

Bezeichner ist wieder eine freie Id. Domain ist ein Bezeichner einer globalen Domain. Die Elemente die im Anschluss folgen müssen aus dieser Domain stammen und vollständig sein. E ist die Id des Elements das negiert werden soll. N ist die Id des negierten Elements. Folgend ein Beispiel:

```
NEGATION not : boolean AS 0/1, 1/0;
```

Relation

RELATION $Bezeichner : Domain1$ -> $Domain2$ AS $Definition$ ;

Bezeichner muss eine Id sein, welche noch nicht verwendet wird. Domain muss jeweils der Bezeichner einer global definierten Domain sein. Die Relation ist dann $xRy$ mit $x \in Domain1$ und $y \in Domain2$. $Definition$ ist die Art und Weise wie die Relation definiert wird. Dies kann als Table, Formula oder als Ausdruck geschehen. Sie werden später erläutert. Hier jedoch der Hinweis, dass Table nur die Elemente 1 und 0 enthalten darf.

Des weiteren muss der Rückgabewert eines Ausdrucks eine Relation sein. Nachfolgend Beispiele:

```
RELATION kleiner : boolean -> boolean AS TABLE (
   0  1,
0: 0  1,
1: 0  0);

RELATION abc : boolean -> boolean AS FORMULA X and Y;

RELATION xyz : boolean -> boolean AS converse(abc);
```

Operation

OPERATION $Bezeichner$ : $Domain1$ x $Domain2$ -> $Domain3$ AS $Definition$ ;

Bezeichner ist eine freie Id. Domain1 und Domain2 sind die Bezeichner globaler Domains. Aus diesen Domains müssen die Operanden stammen. Das Ergebnis der Operation muss aus der Domain3 stammen. Definition kann wie bei der Relation ein Table, Formula oder Ausdruck sein. Das Ergebnis des Ausdrucks muss hier eine Operation sein. Folgend wieder Beispiele:

```
OPERATION or : boolean x boolean -> boolean AS TABLE (
   0  1,
0: 0  1,
1: 1  1);

OPERATION xor : boolean x boolean -> boolean AS FORMULA
(X and not(Y)) or (not(X) and Y);

OPERATION meet : boolean x boolean -> boolean AS generate_meet(leq);
```

### Löschen

Es sollte in der Sprache auch die Möglichkeit geben definierte Objekte wieder entfernen zu können. Hierbei gilt es zu beachten, dass zwischen verschiedenen Objekten Abhängigkeiten bestehen können. Zum Beispiel hängt eine Relation von Domänen ab. Dies bedeutet, dass wenn die Domäne gelöscht wird auch alle Objekte die von dieser abhängig sind gelöscht werden müssen. Das löschen geschieht in der Form:

DELETE $Bezeichner$ ;

Wobei Bezeichner der Name eines global definierten Objekts ist.
Folgend ein Beispiel:

```
DELETE boolean;
```

### Auflisten von Objekten

Es sollte in der RelaFix -Sprache einen Befehl geben, mit dem sich eine Liste der definierten Objekte ausgeben lässt. Dies ist zum Beispiel nützlich, wenn die Sprache als

Kommandozeileninterpreter implementiert wird. Hierfür ist folgender Befehl vorgesehen:

LIST $Type$ ;

Type muss eines der folgenden Schlüsselworte sein: DOMAINS, FUNCTIONS, NEGATIONS, OPERATIONS oder RELATIONS.
Beispiel:

```
LIST DOMAINS;
LIST FUNCTIONS;
```

#### Hinzuladen von Dateien

Eine Anforderung an die Sprache ist es, dass es möglich sein soll durch einen Befehl in der Sprache zusätzlich RelaFix -Dateien einzubinden. Dies ist sinnvoll, da so zum Beispiel Algebren wie Heyting vordefiniert werden können. Diese kann man dann wenn benötigt einfach laden und nutzen. Der Befehl lautet:

LOAD $Dateiname$ ;

Dateiname ist die Angabe zu einer Datei auf der Festplatte. Diese bezieht sich auf das aktuelle Arbeitsverzeichnis des Programms. Auch dieses Arbeitsverzeichnis kann in der RelaFix -Sprache angepasst werden. Hierfür gibt es den Befehl

PATH $Pfad$ ;

Pfad ist der Dateipfad auf den sich LOAD bezieht. Folgend ein Beispiel:

```
LOAD heyting.rfc;
PATH c:\userfiles\;
LOAD fun.rfc;
```

#### Definition durch Tabelle

Wie bei der Definition von Relationen und Operationen schon erwähnt, können diese durch Angabe einer Tabelle definiert werden. Eine Tabelle wird mit dem Schlüsselwort TABLE eingeleitet. Darauf folgt in Klammern eine Liste von Zeilen. Die erste Zeile stellt immer die Beschreibung der Spalten dar. Wobei jede folgende Zeile mit der Beschreibung der Zeile beginnt. Ein TABLE wird wie folgt definiert:

```
TABLE (
         Spalte1   Spalte2   SpalteN,
Zeile1:  Element   Element   Element,
Zeile2:  Element   Element   Element,
ZeileN:  Element   Element   Element);
```

Zeile und Spalte sind Bezeichner aus den Domänen. Zeile aus der ersten Domäne und Spalte aus der zweiten Domäne. Die Elemente in der Mitte müssen bei einer Relation 0

oder 1 sein. Bei einer Operation müssen sie ein Element aus der dritten Domäne sein. Die Reihenfolge der Spalten und Zeilen spielt keine Rolle. Jedoch müssen alle Elemente definiert werden! Auch kann die Formatierung durch Whitespaces beliebig sein. Man könnte zum Beispiel alles in eine Zeile schreiben. Nun ein Beispiel für eine gültige Definition:

```
RELATION kleiner : boolean -> boolean AS TABLE (
   0  1,
0: 0  1,
1: 0  0);

OPERATION or : boolean x boolean -> boolean AS TABLE (
   0  1,
0: 0  1,
1: 1  1);
```

### Ausdruck

Um mit den ganzen definierten Objekten etwas nützliches anfangen zu können muss es eine Möglichkeit geben mit diesen Objekten zu Rechnen. Man bezeichnet dies in einer Computersprache als Ausdruck (englisch: expression). Ein Ausdruck in der RelaFix - Sprache wird folgende Funktionalität bereitstellen:

- Rechnen mit Operationen in Infix-Schreibweise
- Testen von Relationen in Infix-Schreibweise
- Verschachtelte Ausdrücke
- Aufrufen von Funktionen
- Negieren
- Definieren einer temporären Domäne
- Angabe einer Relation, Operation, Domain oder Negation durch einen Bezeichner

Alle genannten Funktionalitäten können miteinander kombiniert werden. Zum Beispiel können die Ergebnisse zweier Funktionen mit einer oder-Operation verknüpft werden.

Ein ausgewerteter Ausdruck liefert immer genau ein Ergebnis. Ist der Ausdruck Teil einer Definition, so sollte das Ergebnis mit dem Typ der Definition übereinstimmen. Zum Beispiel sollte bei der Definition einer Relation auch eine Relation das Ergebnis eines Ausdrucks sein. Wird ein Ausdruck jedoch als eigener Befehl angegeben, so wird das Ergebnis ausgegeben. Die Ausgabe hängt hierbei vom Typ des Ergebnisses ab.

| Typ | Ausgabe |
|---|---|
| Element | Ausgabe des Elements |
| Domain | Ausgabe der Domaindefinition |
| Relation | Ausgabe der Relationsdefinition + Tabelle |
| Operation | Ausgabe der Operationsdefinition + Tabelle |
| Negation | Ausgabe der Negationsdefinition |
| Id | Globale Domain $\rightarrow$ siehe Domain |
| | Globale Relation $\rightarrow$ siehe Relation |
| | Globale Operation $\rightarrow$ siehe Operation |
| | Globale Negation $\rightarrow$ siehe Negation |
| | Funktionsname $\rightarrow$ Beschreibung der Funktion |

Operationen werden wie folgt geschrieben: $Element1\ Operator\ Element2$;. Wobei Operator der Bezeichner des Operators ist und Element1 und Element2 jeweils aus der Domäne die bei der Definition des Operators angegeben wurde stammen müssen. Sollen verschiedene Operationen verschachtelt werden, so müssen die inneren Operationen in Klammern geschrieben werden. Beispiel: $(1\ \ plus\ \ 2)\ \ mal\ \ (5\ \ minus\ \ 6)$; Das Ergebnis einer solchen Rechnung ist dann ein Element aus der Domäne der äußersten Operation.

Um die Existenz einer bestimmten Relation festzustellen, kann $x\ R\ y$ geschrieben werden. x und y sind jeweils Elemente der Domäne die in der Relation definiert ist. R ist der Bezeichner der Relation. Der Rückgabewert dieses Tests ist 0 wenn die Relation nicht existiert und 1 wenn sie existiert. Auch diese Tests können zum Beispiel mit einer Operation kombiniert werden: $(x\ \ R\ \ y)\ \ and\ \ (y\ \ R\ \ x)$;

In der RelaFix -Sprache werden auch einige Funktionen bereitgestellt mit denen die Eigenschaften einer Relation getestet oder mit denen Operationen erzeugt werden können. Diese Funktionen sind in einen Ausdruck integriert. Sie können dort an jeder Stelle aufgerufen werden. Ihre Schreibweise ist wie in C oder Java:

$Bezeichner(Argument1, .., ArgumentN)$;

Bezeichner ist der Name einer von der RelaFix -Sprache bereitgestellten Funktion. Nach diesm Namen folgt in Klammern eine Argumentenliste. Jedes Argument stellt dabei einen neuen Ausdruck dar. Manche Funktionen benötigen auch keine Argumente. Im folgenden Beispiel sieht man eine Funktion in einem Ausdruck:

```
(is_reflexive(leq)) or ((5 minus 2) eq 3);
```

Die Negation wird in der RelaFix -Sprache wie ein Funktionsaufruf gehandhabt. Man nutzt den Bezeichner als Funktionsbezeichner und übergibt das zu negierende Element als Parameter. Beispiel: `not(1);`

Gibt man in einem Ausdruck nur einen einzelnen Bezeichner an, so wird dieser zum Ergebnis des Ausdrucks. Handelt es sich bei dem Bezeichner zum Beispiel um eine Relation, so ist das Ergebnis diese Relation.

Des weiteren gibt es die Möglichkeit in einem Ausdruck eine temporäre Domäne zu definieren. Diese kann zum Beispiel als Argument für eine Funktion dienen oder zur Definition einer globalen Domäne verwendet werden. Die Schreibweise ist unter dem Punkt Domäne definiert.

### Text

Um Ausgaben einer Datei beschreiben zu können, gibt es in der RelaFix -Sprache die Möglichkeit Freitext auszugeben. Dazu schreibt man den Text in Anführungszeichen und beendet den Befehl mit einem Semikolon. Des weiteren gibt es das Ecapezeichen \. Jedes Zeichen das nach diesem Zeichen eingegeben wurde wird ausgegeben. Dies erlaubt das Ausgeben von Sonderzeichen wie Backslash und Anführungszeichen. Beispiel:

```
"Hello \"World\" :D" ;      # Ausgabe lautet: Hello "World" :D
```

### Formeln

Wenn schon einige Relationen und Operationen definiert wurden, wäre es schön wenn man bei einer neuen Definition die Option hätte diese durch Angabe einer Formel statt einer Tabelle zu definieren. Daher definiert die RelaFix -Sprache den Befehl FORMULA der bei der Definition einer Relation oder Operation verwendet werden kann. Die Beschreibung ist wie folgt:

FORMULA $Ausdruck$ ;

Für den Ausdruck gilt nun zu Beachten, dass jetzt die zwei Schlüsselwörter X und Y erlaubt sind. Sie sind die Platzhalter für Elemente aus den Domänen die bei der Definition angegeben werden. Die Formel wird für jede Kombination von X und Y durchgerechnet und das Ergebnis dazu in der Relation oder Operation abgespeichert. Auch hier ist zu beachten, das das Ergebnis den richtigen Typ hat. Für Relationen muss das Ergebnis 0 oder 1 sein. Für Operationen muss das Ergebnis ein Element aus dessen dritter Domäne sein.

Beispiel:

```
OPERATION xor : boolean x boolean -> boolean AS FORMULA
(X and not(Y)) or (not(X) and Y);
```

### Relafix Funktionen

Um die Anforderung zu erfüllen Relationen auf ihre Eigenschaften prüfen zu können, muss die RelaFix -Sprache Funktionen bereitstellen mit denen diese Eigenschaften abgefragt werden können. Im folgenden werden alle diese Funktionen aufgelistet. Ihre Funktionsweise entspricht der mathematischen Definition. Die Syntax der Beschreibung hat immer die Form:

$Bezeichner$($Argumenttyp$): $Rueckgabetyp$

`is_reflexive(Relation): Element aus {0,1}` (0 = nein, 1 = ja)
`is_irreflxive(Relation): Element aus {0,1}`
`is_symmetric(Relation): Element aus {0,1}`
`is_transitive(Relation): Element aus {0,1}`
`is_antisymmetric(Relation): Element aus {0,1}`
`is_asymmetric(Relation): Element aus {0,1}`
`is_equivalent(Relation): Element aus {0,1}`
`is_difunctional(Relation): Element aus {0,1}`

`is_preorder(Relation): Element aus {0,1}`
`is_poset(Relation): Element aus {0,1}`

`id(Domain): Relation` (es existieren $xRx, \forall x \in Domain$)
`empty(Domain, Domain): Relation`
`full(Domain, Domain): Relation` (Alle $xRy$ existieren)

`complement(Relation): Relation`
`converse(Relation): Relation`
`union(Relation, Relation): Relation` (Domänen müssen gleich sein)
`intersection(Relation, Relation): Relation` (Domänen müssen gleich sein)
`concat(Relation, Relation): Relation`

`max(Relation): Element` (Die Relation muss eine $\leq$ Relation sein)
`min(Relation): Element` (Die Relation muss eine $\leq$ Relation sein)
`upperbound(Relation, Domain): Domain` (Die Relation muss eine $\leq$ Relation sein)
`lowerbound(Relation, Domain): Domain` (Die Relation muss eine $\leq$ Relation sein)
`supremum(Relation, Domain): Element` (Die Relation muss eine $\leq$ Relation sein)
`infimum(Relation, Domain): Element` (Die Relation muss eine $\leq$ Relation sein)

`generate_meet(Relation): Operation` (Die Relation muss eine $\leq$ Relation sein)
`generate_join(Relation): Operation` (Die Relation muss eine $\leq$ Relation sein)

generate_meet() und generate_join() erzeugen jeweils Operationen mit denen in einem Ausdruck das meet oder join zweier Elemente ermittelt werden kann.

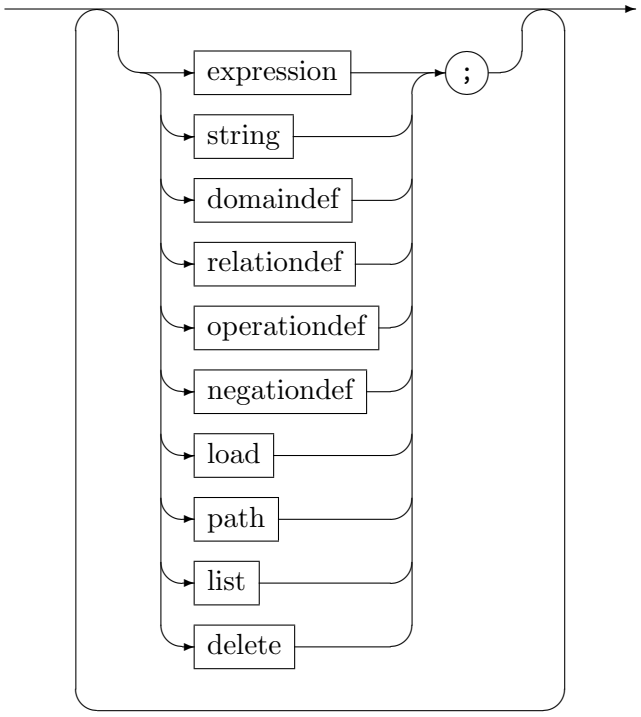
### 3.2.2 Schlüsselwörter

| | |
|---|---|
| AS | NEGATIONS |
| DELETE | OPERATION |
| DOMAIN | OPERATIONS |
| DOMAINS | PATH |
| FORMULA | RELATION |
| FUNCTIONS | RELATIONS |
| LIST | TABLE |
| LOAD | X |
| NEGATION | Y |

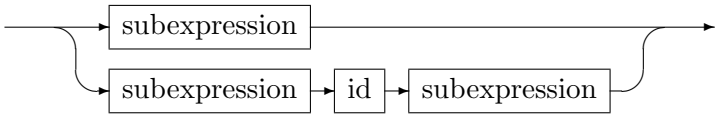
### 3.2.3 Syntax

Folgend der Aufbau der Sprache als Syntax-/Raildiagramm. Abgerundete Felder stellen Terminalsymbole dar, eckige Felder sind Nichtterminale.

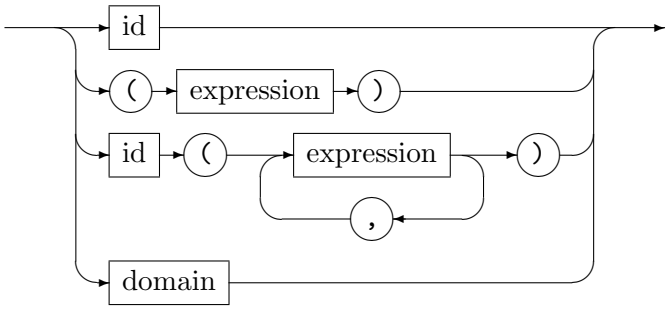
*start*



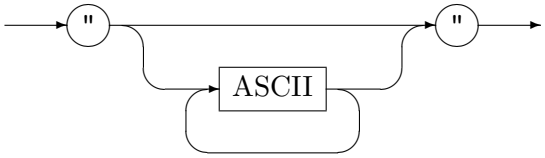

*expression*



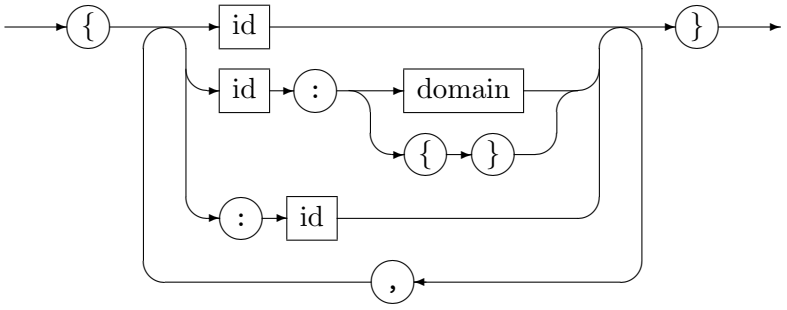

*subexpression*



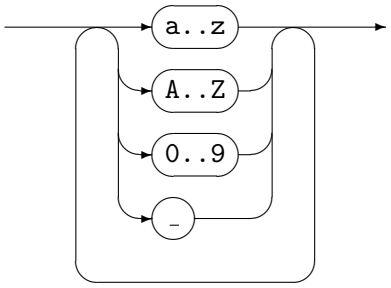

*string*



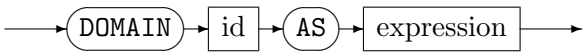
*domain*



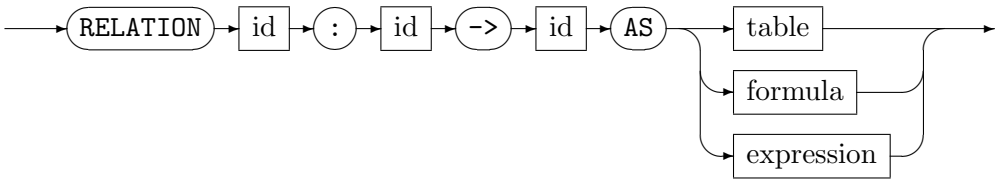

*id*



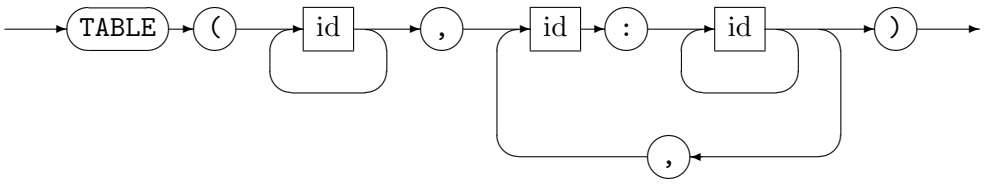

*domaindef*



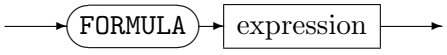

*relationdef*



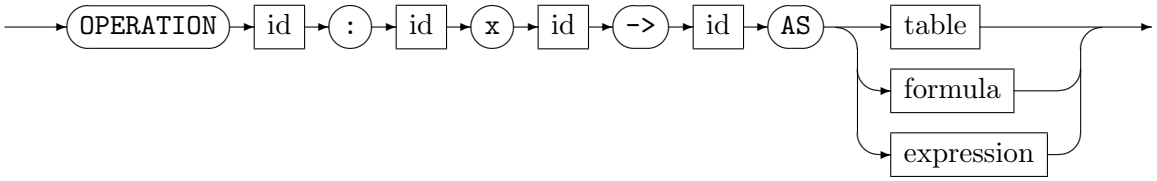

*table*



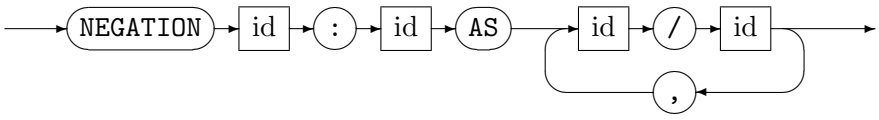

*formula*



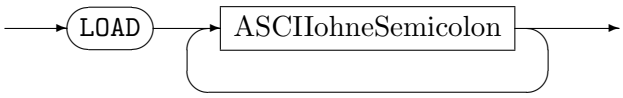

*operationdef*



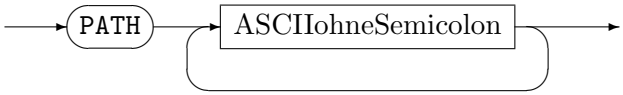
*negationdef*



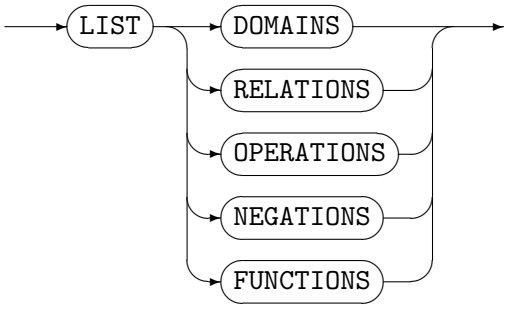

*load*




*path*




*list*




*delete*


### 3.3 Programmentwurf

Die Programmiersprache für die Implementierung dieser Arbeit ist C, da dies eine Anforderung an das Programm ist. Da das Projekt eventuell als OpenSource veröffentlicht werden soll, wird für die Nationalsprache des Codes Englisch verwendet. Dies ermöglicht einer größeren Menge an Personen den Code zu modifizieren. Des weiteren wird sich der Code an den Paradigmen der Kapselung und Wiederverwendbarkeit orientieren.

Um keine Abhängigkeiten zu erzeugen wird auf den Einsatz einer Bibliothek eines Drittanbieters verzichtet und Basisstrukturen selbst implementiert. Folgend eine Auflistung der benötigten Basisstrukturen:

- Array
- Stack
- List

### 3.3.1 Kapselung

Bei der Kapselung werden Strukturen und Funktionen vor dem Nutzer versteckt. Dies hat den Vorteil das er mit der Struktur nur noch über zur Verfügung gestellte Funktionen arbeiten kann und somit kein Wissen über den inneren Aufbau benötigt (Louis 2004, S. 647). Der größte Pluspunkt ist jedoch, dass die komplette Implementierung hinter dem Interface jederzeit geändert werden kann, ohne das sonstiger Programmcode geändert werden muss.

### 3.3.2 Wiederverwendbarkeit

Wiederverwendbarkeit bedeutet den Code so zu schreiben das er für verschiedene Datentypen genutzt werden kann. Dies bringt eine Codereduktion mit sich und vereinfacht somit die Wartbarkeit. Ausgereifte Module tragen zur Codestabilität bei, da die meisten Fehler schon behoben wurden. In C wird diese Wiederverwendbarkeit durch den Einsatz des void-Pointers erreicht, wodurch jedoch die Typensicherheit nicht mehr gewährleistet ist. Die Vorteile überwiegen aber. In RelaFix wird dies zum Beispiel bei *list* oder *array* angewendet.

### 3.3.3 Namensraum

Um Kollisionen der Funktions- und Strukturnamen mit anderen Bibliotheken auszuschließen wird ein Präfix benötigt, welches vor jeden globalen Funktions-, Struktur- und Variablennamen gesetzt werden muss. Da der Name des Programms RelaFix ist, wird der Präfix RF lauten. Folgend Beispiele:

- Variable: rf_counter
- Struktur: RF_LIST
- Funktion: rf_list_create

### 3.3.4 Boolesche Ausdrücke

Da C keine booleschen Ausdrücke kennt, diese aber den Code verständlicher erscheinen lassen, werden Sie für RelaFix wie folgt definiert:

Listing 1: Definition von RF_BOOL

```
1 #define RF_FALSE     0
2 #define RF_TRUE      !0
3 typedef unsigned char RF_BOOL;
```

### 3.3.5 Basisstrukturen

Um das Programm RelaFix zu realisieren werden Strukturen benötigt in welchen Daten wie zum Beispiel die Elemente einer Domain gespeichert werden können. Des weiteren müssen auch die beim Parsen erzeugten Strukturen festgehalten werden. Um möglichst portabel zu bleiben, wird auf den Einsatz einer externen Programmbibliothek die diese

Funktionalität liefert verzichtet. Daher sind für das Programm RelaFix die nötigen Strukturen zu implementieren. Insgesamt werden 3 Strukturen benötigt. Dies sind eine Liste, ein Array und ein Stack.

Die Liste soll flexibel ausgelegt sein, so das Elemente effizient an jeder Stelle eingefügt oder gelöscht werden können. Außerdem sollen die Elemente effizient in beide Richtungen durchlaufen werden können.

Im Gegensatz zur Liste ist die Aufgabe des Arrays einen schnellen Zugriff auf jedes Element zu gewährleisten. Dies hat jedoch den Nachteil, das neue Elemente nur am Ende des Arrays effizient eingefügt und gelöscht werden können. Der Einsatz eines Arrays lohnt sich also wenn die Daten einmal angelegt werden und danach die meiste Zeit auf ihnen operiert wird. Der Vorteil gegenüber C-Arrays ist, das dieses Array seine Größe kennt und es wachsen kann.

Ein Stack ist eine Struktur in die man Objekte legen kann. Wenn man die Objekte wieder auslesen möchte, so geschieht das in umgekehrter Reihenfolge. Somit ist das erste Element das vom Stack genommen wird das Element, welches zuletzt auf den Stack gelegt wurde. Der Stack der für diese Arbeit implementiert wird, soll auf dem Array aufbauen.

### 3.3.6 Organisation

Bei dieser Arbeit handelt es sich um das Erstellen einer ersten Version eines Programms, dass später noch erweitert werden soll. Diese erste Version soll eine Ein- und Ausgabe auf der Kommandozeile besitzen. Es ist aber angedacht, dass später eine graphische Benutzeroberfläche entwickelt werden soll. Es ist auch vorstellbar eine Programmvariante zu haben, welche Dateien in der RelaFix-Sprache ließt und LaTeX-Dateien als Ausgabe erzeugt. Daher ist es wichtig sich an diesem Punkt darüber Gedanken zu machen wie das Projekt anzulegen ist.

Eine Möglichkeit ist es, das Programm als Kommandozeilenprogramm zu schreiben, so das ein seperates graphisches Frontend über Pipes mit ihm kommunizieren kann. Dies ist jedoch umständlich und zudem soll das Programm möglichst plattformunabhängig sein. Auch ist nicht gewährleistet das Pipes auf jedem System zur Verfügung stehen. Daher scheidet diese Variante aus.

Eine andere Möglichkeit ist es das Programm als eine Programmbibliothek zu schreiben, welche von dem jeweiligen Frontend eingebunden wird. Damit kann dem Frontend ein direkter Zugang zu Datenstrukturen ermöglicht werden. Des weiteren wird die Programmbibliothek komplett von der Ein- und Ausgabe abgekoppelt. Die Ein- und Ausgabe ist Aufgabe des Frontends. Die Bibliothek fordert alle benötigten Daten vom Frontend an. Da diese Variante flexibler ist, wird sie in diesem Projekt verwendet.

Das Frontend wird sich nur um die Ein- und Ausgabe kümmern. Angenommene Befehle werden an die Programmbibliothek weitergeleitet. Diese muss dafür von dem Frontend eingebunden werden, wodurch dem Frontend Funktionen zur Verfügung stehen, die dafür genutzt werden können. Auf diese Weise kann das Frontend die Eingaben zur Bearbeitung an die Bibliothek weiterreichen oder für die Benutzeroberfläche benötigte Informationen erfragen.

Die Aufgabe der Programmbibliothek ist es die Daten zu Verwalten und Berechnungen durchzuführen. Sowie das Parsen der RelaFix-Sprache. Da die Bibliothek keine Ein- und Ausgabe kennt, wird sie einige Funktionen deklarieren die vom Frontend implementiert

werden müssen. Dies wäre zum Beispiel rf_parser_error. Mit dieser Funktion kann die Bibliothek dem Frontend Fehler beim Parsen mitteilen. Das Frontend kann diese dann in passender Weise ausgeben. In einer graphischen Oberfläche zum Beispiel in einem PopUp-Fenster. Oder in der Kommandozeile bei einem Kommandozeilenfrontend.

Das Anwendungsdiagramm in Abbildung 1 macht diese Aufgabenverteilung ersichtlich.

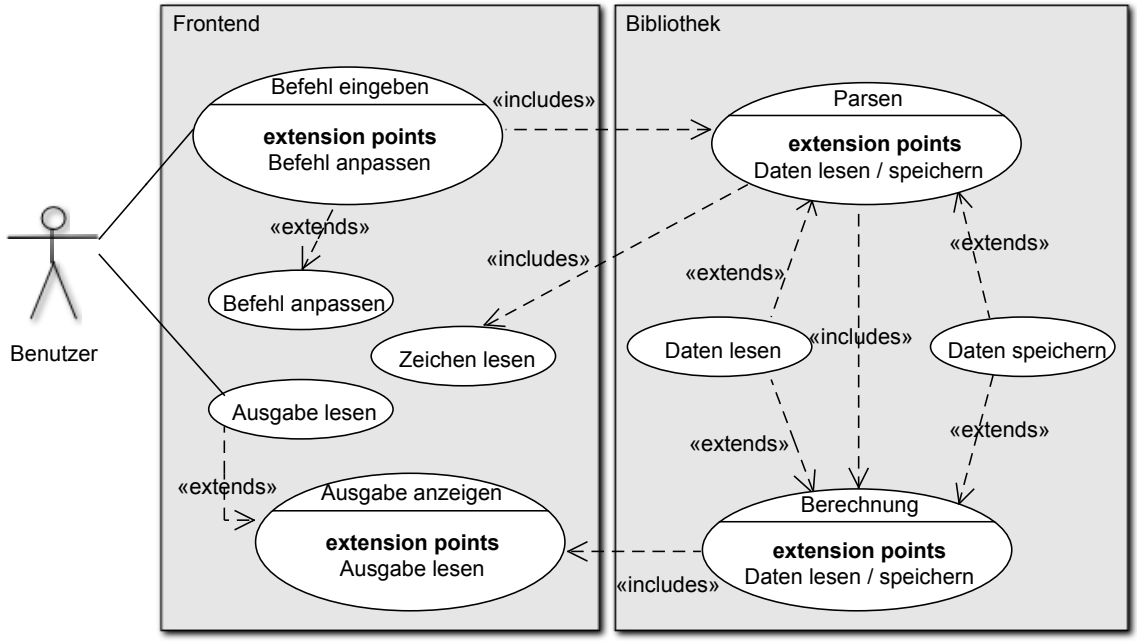



Abbildung 1: Aufgaben Frontend und Bibliothek

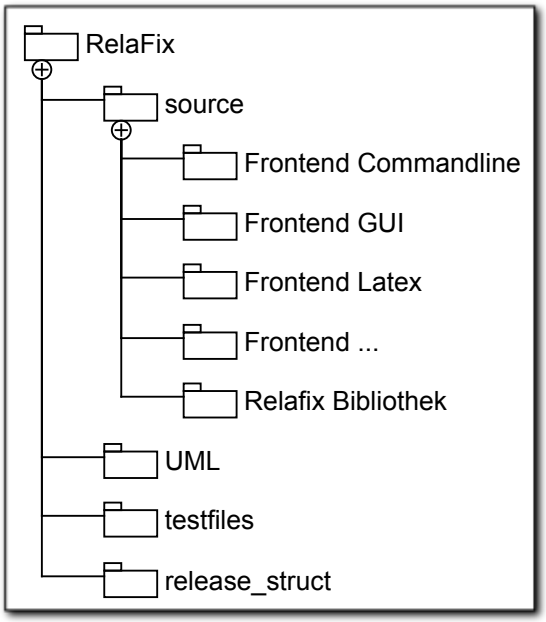



Abbildung 2: Verzeichnisstruktur

Im folgenden wird nun auf die Verzeichnisstruktur auf der Festplatte eingegangen. Da das Programm nun in Bibliothek und Frontend aufgeteilt ist, sollte es auch auf der Festplatte getrennt sein. Die Dateien die zur Bibliothek gehören werden in einem eigenen Verzeich-

nis gespeichert. Genauso die Dateien des jeweiligen Frontends. Abbildung 2 zeigt die Aufteilung im Dateiverzeichnis. Zu beachten ist, dass die Ordner im Verzeichnis *source* hier nur symbolische Namen haben. Auch zu sehen sind die Ordner *UML, testfiles* und *release_struct*. *UML* enthält Diagramme zum Projekt, die in dieser Arbeit und in der Dokumentation gezeigt werden. In *testfiles* werden Dateien mit RelaFix-Code gespeichert, mit denen getestet werden kann ob das Programm nach Änderungen noch korrekt funktioniert. Zuletzt gibt es noch das Verzeichnis *release_struct*, welches alle Daten enthält die ein Release des Programms benötigt. Wird das Programm mit Hilfe des Makefiles compiliert, so wird von diesem Ordner eine Kopie erstellt und das erzeugte Programm hinzugefügt. Das neue Verzeichnis kann dann als fertiges Programm weitergegeben werden.

### 3.3.7 Programmaufbau

Hier wird nun der Aufbau der RelaFix -Bibliothek beschrieben. Zunächst wird aber auf den gewählten Parser eingegangen.

#### Parser

Im Kapitel 2.3 wurde als Parser *bison* gewählt. Um mit ihm einen Parser zu generieren muss man eine Datei anlegen, welche die Regeln zur Parsergenerierung enthält. In diesem Projekt stehen diese Regeln in der Datei *parser.y*. Eine Regel ist vergleichbar mit einer Regel der BNF. Man kann jedoch jeder dieser Regel einen Codeblock anfügen, der wenn diese Regel beim parsen angewendet wird, ausgeführt wird. Dies ist die Schnittstelle zwischen Parser und eigenem Programm. Folgend in Listing 2 ein Beispiel solch einer Regel mit Codeblock. Der Codeblock steht in den geschweiften Klammern und wird ausgeführt nach dem das Semikolon der Regel *statement* eingelesen wurde. Im Codebock selbst kann mit $n auf die einzelenen Daten der Elemente einer Regel zugegriffen werden. Hier wird mit $2 auf *string* zugegriffen. Mit @n kann auf die Position des eingelesenen Zeichens zugegriffen werden um dies bei einer Fehlermeldung angeben zu können.

Listing 2: Parserregel mit Codeblock

```
1  input:  input statement
2      ;
3
4  statement:  "write" string ';'
5  {
6      if($2)
7          puts($2);
8      else
9          printf("error␣at␣\%d.\%d\\n",
10             @2.first_line, @2.first_column);
11 }
12 ;
13
14 string: ...
```

Da der Parser nur Token kennt muss ein Programm das *bison* benutzt die Funktion *yylex* implementieren. Diese wird dann nach dem Start des Parsers vom diesem aufgerufen. Die Aufgabe von *yylex* ist es die Eingabezeichen von einer beliebigen Quelle zu lesen,

auszuwerten und in ein Token umzuwandeln. Dieses Token wird dann an den Parser übergeben. Im Falle von RelaFix wird in dieser Funktion auf Keywörter und Kommentarzeilen geprüft. Zu finden ist diese Funktion in der Datei *lexer.c*.

Eine detaillierte Beschreibung des Parsergenerators *bison* findet man unter:

`http://www.gnu.org/software/bison/manual/index.html`

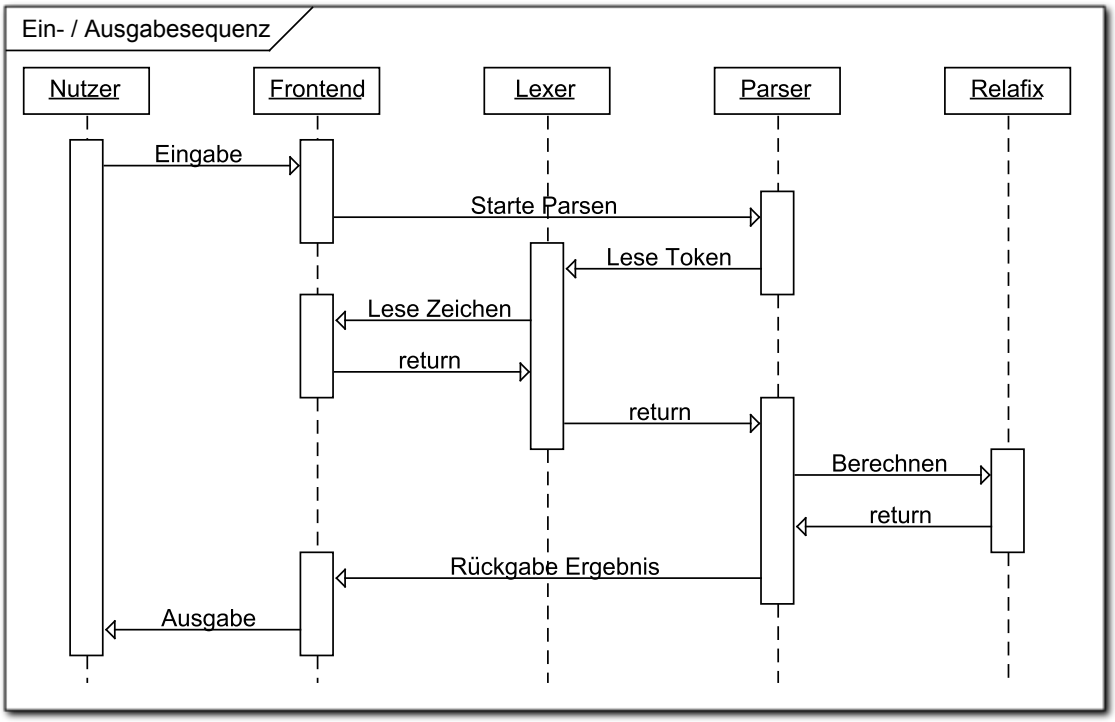



Abbildung 3: Ein- / Ausgabesequenz

In Abbildung 3 sind die benötigten Programmkomponenten zu sehen wie sie nach einer Befehlseingabe miteinander agieren. *Lexer, Parser* und *Relafix* bilden zusammen die RelaFix -Bibliothek.

### Fehlerbehandlung

Da die Bibliothek wie beschrieben keine Ausgabe hat, jedoch die Eingaben verarbeitet, muss es eine Möglichkeit geben dem Nutzer eine nützliche Fehlermeldung mitzuteilen. Hierzu wird eine Funktion definiert, die von einem Frontend implementiert werden muss. Im Programm hat diese Funktion den Namen *rf_parser_error*. An sie werden die Koordinaten an denen der Fehler aufgetreten ist, sowie eine Beschreibung des Fehlers selbst übergeben. Das Frontend kümmert sich dann um die Ausgabe.

Da die meisten Funktionen nicht die Position des auftretenden Fehlers kennen und damit nicht wild aus jeder Funktion Fehlermeldungen an das Frontend geschickt werden, ist noch eine zweite Strategie nötig. Es wird dazu eine Struktur RF_ERROR definiert. Diese wird an die jeweilige Funktion übergeben und im Falle eines auftretenden Fehlers von dieser mit Informationen gefüllt. Auf diese Weise kann eine Beschreibung eines Fehlers der in einem tief verschachtelten Funktionsaufruf auftritt an den Parser geleitet werden. Dieser kann dann die Fehlerbeschreibung mit Position mittels *rf_parser_error* dem Frontend übergeben. Die Struktur RF_ERROR ist in der Datei *defines.h* definiert. Hier existiert keine Schnittstelle, die Struktur ist dazu gedacht direkt bearbeitet zu werden.

### Modellierung der mathematischen Strukturen

Da nun der grundlegende Aufbau des Programms mit Ein- / Ausgabe und dem Aufrufen des Parsers festgelegt sind, können die Datenstrukturen entworfen werden, welche die mathematischen Objekte die in der RelaFix -Sprache vorkommen im Programm abbilden.

Der Aufbau wird hier wie bei den Basisstrukturen dem Konzept der Kapselung und der Wiederverwendbarkeit folgen. Es wird C Strukturen geben, welche nur über Schnittstellenfunktionen genutzt werden können. Die mathematischen Objekte Domain, Relation, Operation und Element werden als solche auch in C-Strukturen angelegt um die mathematischen Daten zu halten. Dazu kommt noch ein Satz an Funktionen wie zum Beispiel das Ausrechnen einer Operation. Weiterhin gibt es in der in Kapitel 3.2 definierten Sprache die Möglichkeit Funktionen aufzurufen und Negationen durchzuführen. Funktionen werden ebenso in einer C-Struktur festgehalten. Negationen sind im Gegensatz zu den Funktionen zur Laufzeit festlegbar und erhalten daher eine seperate Struktur.

Es stellt sich die Frage wie Relationen im Speicher zu halten sind? Sie können als Graphen betrachtet werden, jedoch auch durch eine Tabelle beschrieben werden. Da verschiedenste Funktionen auf den Daten ausgeführt werden sollen, eignet sich eine Repräsentation als zweidimensionales Array im Programm gut, da sie einen schnellen Zugriff auf die einzelnen Relationen erlaubt und so die Findung von Algorithmen vereinfacht. Des weiteren muss für die Repräsentation einer einzelnen Relation nur zwischen *existiert* und *existiert nicht* unterschieden werden, also 0 oder 1. Diese Information kann platzsparend auf Bitbasis abgespeichert werden.

Bei der Operation wird auch ein zweidimensionales Array benötigt um alle möglichen Lösungen dieser Speichern zu können. Die Lösungen werden zur Laufzeit durch eine Eingabe in der RelaFix -Sprache definiert.

Die Datensätze der *Relation*, sowie der *Operation* sind sich sehr ähnlich. Daher wird das zweidimensionale Array in eine eigene Struktur ausgelagert. Die *Relation* oder *Operation* kann dann über Schnittstellenfunktionen die Daten lesen und schreiben. Da der Vorgang des Zugriffs auf die Daten nun hinter der Schnittstelle verschwindet, muss bei der Implementierung von Algorithmen auf *Relationen* oder *Operationen* nicht mehr darauf geachtet werden wie die Daten zu adressieren sind. Dies erlaubt es die Implementierung auch später noch ändern zu können ohne den Rest des Programms anrühren zu müssen. Die neue Struktur wird im Programm mit *Table* bezeichnet.

Im weiteren Verlauf der Arbeit wird genauer auf den Aufbau der einzelnen Strukturen eingegangen. Folgend in Abbildung 4 eine grobe Übersicht wie die einzelnen Objekte zusammenhängen.

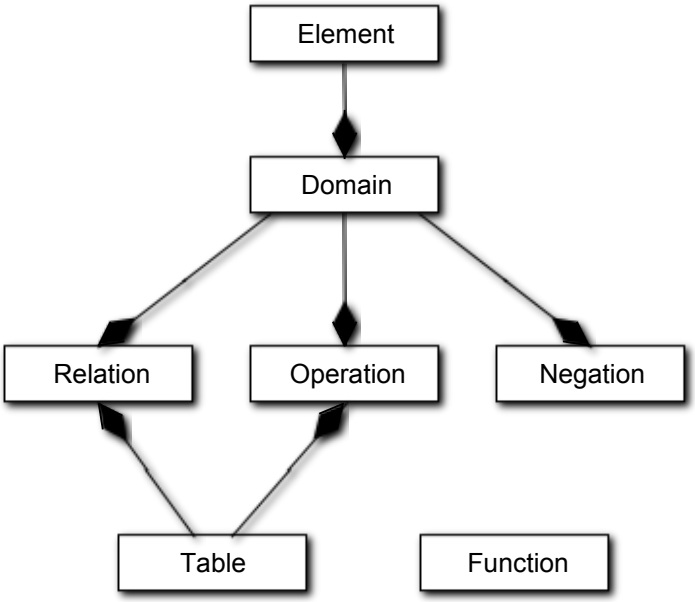



Abbildung 4: Komposition der Strukturen

**Datenhaltung**

Die RelaFix -Bibliothek muss sich die Daten die über die RelaFix -Sprache definiert werden merken können und bei Bedarf zur Berechnung nutzen. Die Objekte die mit der Sprache angelegt werden können sind:

- Domain
- Relation
- Operation
- Negation

Hinzu kommen noch die *Funktionen*, welche im Programm angelegt werden. Da der Parser oft bei der Auswertung von IDs nur auf bestimmte Objekte wie *Domain* oder *Operation* testen muss, wird für die Speicherung der Objekte jeweils eine eigene Liste verwendet. Hier ist anzumerken, dass die Nutzung einer Hashmap an dieser Stelle effizienter ist. Diese steht in dieser Arbeit aber nicht zur Verfügung und auf eine eigene Implementierung wurde aus Zeitgründen verzichtet. Da aber bei der geplanten Nutzung des Programms kaum Daten anfallen, sollte die Nutzung von Listen kein Problem ergeben. Folgend die Auflistung der Funktionen die genutzt werden um auf die Operationen zuzugreifen. Um auf andere Objekte zuzugreifen muss nur der Name angepasst werden. Zum Beispiel statt Operation, Negation.

- rf_parser_add_operation
- rf_parser_delete_operation
- rf_parser_get_operation

### Relafix Definitionen

Das fertige Programm soll Standarddomänen, -relationen und operationen wie zum Beispiel *Heyting* und *Boolean* mitliefern. Dies wird hier so gelöst, dass das fertige Programm in einem Verzeichnis ausgeliefert wird, in welchem sich Dateien mit Definitionen für *Heyting* oder andere Relationen in der RelaFix -Sprache befinden. Diese Dateien können dann falls benötigt vom Nutzer in der RelaFix -Sprache geladen werden. Auf diese Weise wird auch ein festes Einprogrammieren umgangen!

# 4 Implementierung

## 4.1 Basisstrukturen

### Array

In Programmen wird generell ein Array benötigt um eine größere Anzahl gleicher Daten im Speicher zu halten. Ein in C erstelltes Array ist nicht sehr flexibel, da es seine Größe nicht kennt und die Größe nach dem erstellen nicht änderbar ist. Um bei der Entwicklung des Programms RelaFix  sich nicht um diese Probleme kümmern zu müssen wird eine neue Array Struktur definiert, welche sich die Größe merkt und dynamisch wachsen kann.

Anforderungen:

- Aufnahme variabler Anzahl von Elementen
- Größe ist nach der Initialisierung nicht fest. Es können Elemente angehängt werden.
- Direkter Zugriff auf jedes Element
- Elemente müssen später durchsucht werden können

Abbildung 5 zeigt die interne Struktur des Arrays. In *length* wird die Größe des allokierten Speicherbereiches, auf den mit *array* zugegriffen wird, festgehalten. *count* hingegen gibt die Anzahl der genutzten Speicherstellen an. *array* selbst ist ein Pointer auf ein C-Array, dass Pointer des Typs void speichert. Durch die Nutzung der void-Pointer wird die Wiederverwendbarkeit ermöglicht.

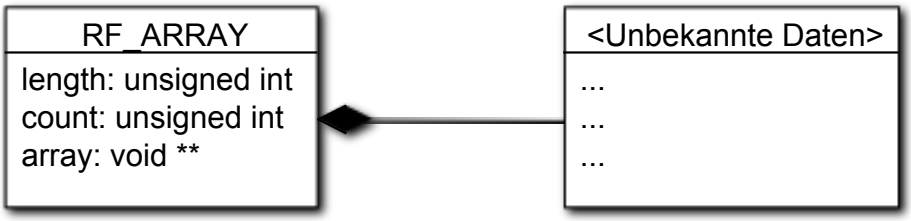



Abbildung 5: Struktur des Arrays

Folgenden Funktionen werden benötigt, damit man mit der Struktur arbeiten kann:

- rf_array_create
- rf_array_destroy
- rf_array_append
- rf_array_read
- rf_array_write
- rf_array_delete
- rf_array_merge

- rf_array_size
- rf_array_swap_order

rf_array_create und rf_array_destroy werden dazu genutzt um die Struktur im Speicher anzulegen, sowie abzubauen. Mit rf_array_read, rf_array_write und rf_array_delete kann auf die einzelnen Daten direkt zugegriffen werden. Um Daten an das Array anzuhängen kann die Funktion rf_array_append genutzt werden, welche falls nötig neuen Speicher allokiert. Des weiteren ist es mit rf_array_merge möglich zwei Arrays zusammenzuführen. Eine Umkehrung des Arrays kann mit der Funktion rf_array_swap erreicht werden. Und die Größe des Arrays ermittelt man mit einem Aufruf von rf_array_size.

Wenn neue Daten mit rf_array_append an das Array angehangen werden und der allokierte Speicher aufgebraucht ist, wird das Array automatisch mehr Speicher allokieren. Hierbei wird jedesmal der vorhandene reservierte Speicher verdoppelt um effizient zu bleiben.

### Stack

Ein Stack wird in RelaFix  benötigt um Dateien verschachtelt laden zu können. Die zuletzt geöffnete Datei kann dann einfach auf den Stack gelegt werden und die neue Datei geladen. Nachdem die Datei geladen wurde kann die zuvor geöffnete Datei wieder vom Stack geholt werden.

Der Stack baut hier auf dem Array auf. Da der Zugriff durch die Kapselung nur über spezielle Funktionen möglich ist, ist es dem Nutzer nicht möglich wahlweise auf verschiedene Positionen des genutzten Arrays zuzugreifen. Die Wiederverwendbarkeit wird wie zuvor schon beim Array durch die Verwendung von void-Pointern hergestellt.

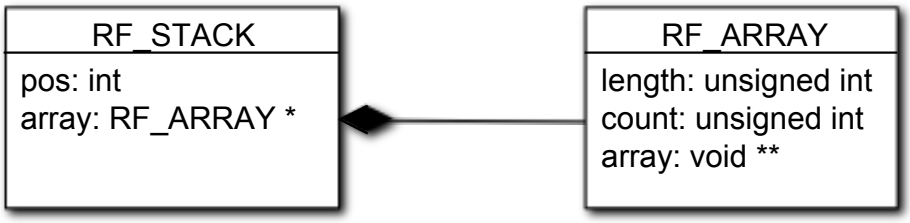



Abbildung 6: Struktur des Stack

Mit pos wird die Speicherstelle im Array festgehalten. Diese Stelle ist zugleich der Gipfel des Stacks.

Folgende Funktionen werden benötigt um auf dem Stack operieren zu können:

- rf_stack_create
- rf_stack_destroy
- rf_stack_push
- rf_stack_pop

Mit rf_stack_create und rf_stack_destroy wird ein Stack angelegt und wieder freigegeben. Daten werden mit rf_stack_push auf den Stack gelegt und können mit rf_stack_pop wieder zurückgeholt werden. Wobei rf_stack_pop immer nur den obersten Datensatz zurückgeben kann.

### Liste

Um das Programm RelaFix zu realisieren wird eine Struktur benötigt in der die Elemente einer Domain gespeichert werden können. Des weiteren müssen auch die beim Parsen erzeugten Strukturen festgehalten werden. An die Struktur werden folgende Anforderungen gestellt:

- Aufnahme variabler Anzahl von Elementen
- Elemente müssen nicht sortiert sein
- Elemente müssen später durchsucht werden können
- Elemente müssen zu jeder Zeit gelöscht werden können
- Auf Elemente muss zu jeder Zeit zugegriffen werden können

Da die Elemente nicht sortiert sind und bei der Suche der Reihe nach abgearbeitet werden eignet sich als dynamische Speicherstruktur eine Liste am besten.

Die Abbildung 7 zeigt die Strukturen RF_LIST, RF_LIST_ITEM und RF_LIST_ITERATOR der Liste. Es handelt sich hier um eine doppelt verlinkte Liste.

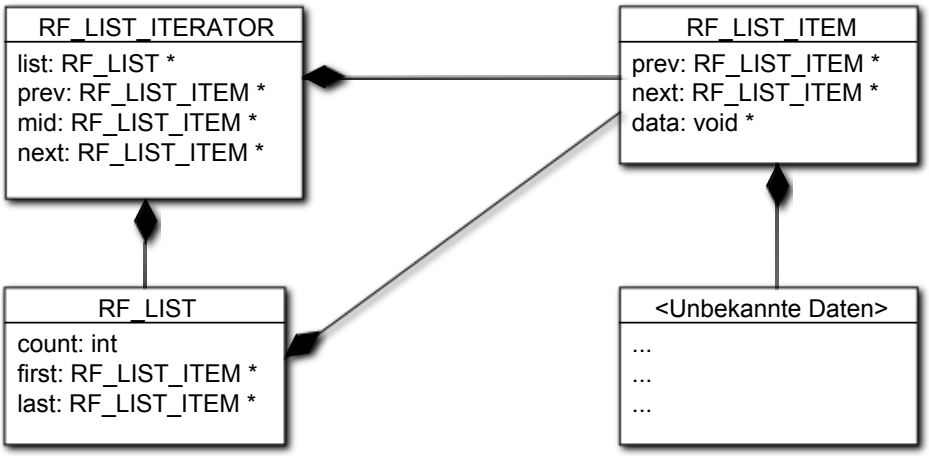



Abbildung 7: Strukturen der Liste

RF_LIST ist die Hauptstruktur der Liste. In ihr wird das erste und letzte Element festgehalten, sowie die Anzahl aller Elemente. Die Informationen werden benötigt um Iteratoren erzeugen zu können. Der Nutzer kann mit folgenden Funktionen eine Liste erzeugen, zerstören, die Anzahl der Elemente abfragen und Elemente hinzufügen.

- rf_list_append
- rf_list_create

- rf_list_destroy
- rf_list_size

RF_LIST_ITEM stellt einen Knoten der verketteten Liste dar. Er enthält Verweise auf seinen Vorgänger und Nachfolger, sowie einen Zeiger auf unbekannte Daten.

Die Struktur RF_LIST_ITERATOR erlaubt es die Liste vom Beginn oder Ende in beide Richtungen zu durchlaufen und gibt somit Zugriff auf die Daten in der Liste. Dazu hält der Iterator Verweise auf das aktuelle, vorherige und nächste Element. Die folgenden Funktionen dienen dem Zugriff auf die Daten:

- rf_list_delete
- rf_list_has_next
- rf_list_has_prev
- rf_list_next
- rf_list_prev

Iteratoren können für den Listenanfang oder das Listenende erzeugt werden. Sie enthalten danach jeweils nur einen Verweis in prev oder next! Das Bedeutet, dass der Iterator nach der Erzeugung entweder vor dem ersten Element steht oder hinter dem letzten Element. Es muss folglich zuerst mit rf_list_next oder rf_list_prev zum nächsten Element gesprungen werden. Dies mag im ersten Augenblick umständlich erscheinen. Es eröffnet jedoch die Möglichkeit eine saubere Schleife ohne unnötige Tests zu schreiben.

- rf_list_get_begin
- rf_list_get_end

Folgend ein Beispiel eines Iteratordurchlaufs:

Listing 3: Nutzung von Iteratoren

```
1  void do_something(RF_LIST *list){
2      RF_LIST_ITERATOR *iterator;
3
4      if(!list)
5          return;
6
7      iterator = rf_list_get_begin(list);
8
9      while(rf_list_has_next(iterator))
10     {
11         rf_list_next(iterator);
12         rf_list_delete(iterator);  /* deletes the item */
13     }
14
15     rf_list_delete_iterator(iterator);
16 }
```

Daten werden an die Liste mit rf_list_append angehangen. Die Liste kennt von den Daten nur die Position im Speicher. Typ und Größe sind unbekannt. Da die Übergabe der Daten als void-Pointer geschieht, kann die Liste für alle Daten verwendet werden und ist somit wiederverwendbar. Der Nutzer hat selbst dafür zu sorgen das er die Daten in den richtigen Typ umwandelt! Dies ist in C aber üblich (Siehe qsort in der C Bibliothek).

## 4.2 Mathematische Strukturen

### Element

Elemente sind in der RelaFix -Sprache beliebige Bezeichner (außer Schlüsselwörter), welche zu einer oder mehrere Domänen gehören. Sie können aber auch selbst eine Domäne sein! Daher wird zur Repräsentation eines Elements eine Struktur benötigt, die sowohl einen Bezeichner wie auch eine Domäne darstellen kann. Es ist auch zu beachten, das in der RelaFix -Sprache der Domäne selbst ein Bezeichner zugewiesen werden muss. Der Name der Struktur im Programm ist RF_ELEMENT. Der Aufbau ist in Abbildung 8 dargestellt.

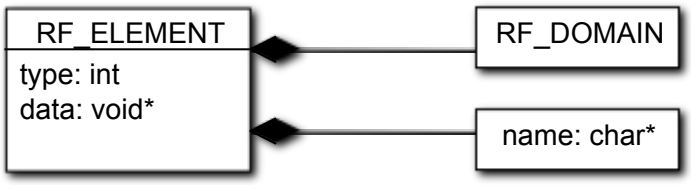



Abbildung 8: Struktur des Elements

*type* gibt an ob die Variable *data* auf einen Namen zeigt, oder ob es sich bei dem Element um eine ganze Domäne handelt. Der Wert den *type* annehmen kann ist in der Datei *domain.h* definiert.

Auch bei RF_ELEMENT wird die Kapselung genutzt. Mit folgenden Funktionen kann mit der Struktur gearbeitet werden:

- rf_element_copy
- rf_element_create
- rf_element_destroy
- rf_element_get_data
- rf_element_get_name
- rf_element_get_type

Mit *rf_element_create* und *rf_element_destroy* wird die Struktur angelegt und zerstört. *rf_element_copy* erstellt eine Kopie. Hier ist anzumerken, dass sollte es sich um eine Domäne handeln, diese nicht kopiert wird! Das Original und die Kopie zeigen also auf die selbe Domäne. Mit *rf_element_get_data*, bekommt man einen Zeiger zu den Daten. Den Type ermittelt man mit *rf_element_type* und den Bezeichner mit *rf_element_name*. Handelt es sich um eine Domäne, so liefert *rf_element_name* den Namen der Domäne.

### Domain

Eine Domäne auch Menge genannt, kann Elemente und andere Domänen enthalten. In RelaFix hat eine Domäne auch einen Namen. Die später erläuterten Objekte *Relation*, *Operation* und *Negation* sind von ihr abhängig. Der strukturelle Aufbau ist aus Abbildung 9 zu entnehmen.

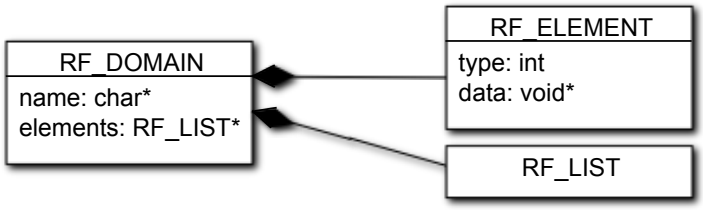



Abbildung 9: Struktur der Domäne

*name* ist ein Zeiger auf den Bezeichner dieser Domäne. Die Elemente einer Domäne sind in der Liste *elements* gespeichert. Des weiteren besitzt auch die Domäne Funktionen mit denen man auf der Struktur arbeiten kann. Die wichtigsten sind

- rf_domain_check_elements
- rf_domain_create
- rf_domain_destroy
- rf_domain_has_element

Die Domäne wird mit *rf_domain_create* angelegt und mit *rf_domain_destroy* wieder zerstört. *rf_domain_has_element* erlaubt es zu erfragen ob ein Bezeichner Element der Domäne ist. Die Funktion *rf_check_elements* überprüft eine Liste mit Elementen ob diese für den Einsatz in einer Domäne in Ordnung ist. Es wird zum Beispiel geprüft ob Elemente doppelt vorkommen. Der komplette Satz Funktionen ist in der Datei *domain.h* ersichtlich.

### Relation

Eine Relation beschreibt die Beziehung zwischen zwei Elementen. In RelaFix wird eine Relation durch die Struktur RF_RELATION beschrieben. Im Vergleich zur mathematischen Relation beschreibt RF_RELATION alle Relationen von allen Elementen einer Domäne zu allen Elementen einer zweiten Domäne. Wobei erste und zweite Domäne identisch sein dürfen. Abbildung 10 zeigt den genauen Aufbau von RF_RELATION. Des weiteren wird in einem 2 dimensionalen Array die Information gespeichert ob eine Relation zwischen zwei Elementen existiert. Dies geschieht platzsparend als Binärwert.

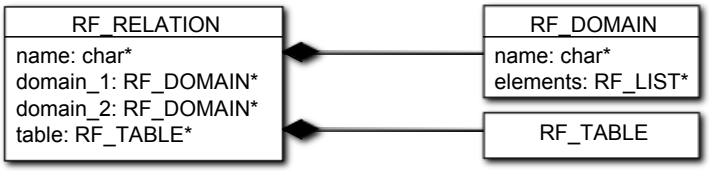



Abbildung 10: Struktur der Relation

*name* ist der Bezeichner der Relation. Über ihn kann in der RelaFix -Sprache auf die Relation zugegriffen werden. Betrachten wir die Relation $aRb$ dann ist $a$ ein Element aus *domain_1* und $b$ ist Element aus *domain_2*. In *table* sind alle Relationen zwischen Domäne 1 und Domäne 2 gespeichert.

Auch hier gilt wieder das Prinzip der Kapselung und es gibt eine Reihe an Schnittstellenfunktionen um auf Relationen arbeiten zu können. Folgend die wichtigsten:

- rf_relation_calc
- rf_relation_copy
- rf_relation_create
- rf_relation_destroy

Mit *rf_relation_create* und *rf_relation_destroy* wird wieder die Struktur angelegt und abgebaut. *rf_relation_calc* wird dazu genutzt um zu testen ob eine Relation existiert oder nicht. Des weiteren ist hier der richtige Platz um die Funktionen zu definieren, die die Eigenschaften einer Relation testen oder basierend auf einer Relation eine neue erzeugen.

- rf_relation_create_complement
- rf_relation_create_converse
- rf_relation_create_intersection
- rf_relation_create_union
- rf_relation_is_antisymmetric
- rf_relation_is_asymmetric
- rf_relation_is_difunctional
- rf_relation_is_equivalent
- rf_relation_is_irreflexive
- rf_relation_is_poset
- rf_relation_is_preorder
- rf_relation_is_reflexive
- rf_relation_is_symmetric
- rf_relation_is_transitive

Die vollständige Liste aller Funktionen im Bezug zur Relation findet man in der Datei *relation.h*.

### Operation

Eine Operation ist ähnlich einer Relation. Während die Relation nur angibt das eine Beziehung zwischen zwei Elementen besteht, gibt die Operation für zwei Elemente ein Element aus einer dritten Domäne zurück. Daher wird auch der Aufbau von RF_OPERATION dem von RF_RELATION sehr ähnlich. Es wird lediglich eine dritte Domäne hinzugefügt. Ein weiterer Unterschied ist, das im zweidimensionalen Array statt Binärwerten, Zeiger zu Bezeichnern gespeichert werden. Diese Bezeichner müssen aus der dritten Domäne stammen. Der Aufbau von RF_OPERATION ist in Abbildung 11 zu sehen.

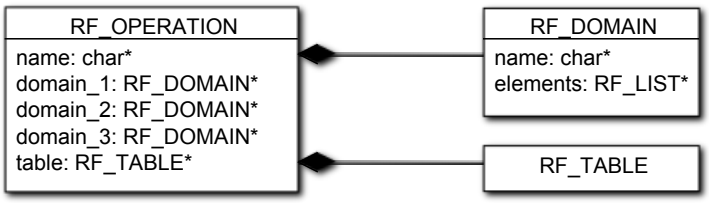



Abbildung 11: Struktur der Operation

*name* enthält den Bezeichner mit dem aus der RelaFix -Sprache die Operation aufgerufen werden kann. Wie bei der Relation $aRb$ ist auch hier $a$ ein Element aus *domain_1* und $b$ ein Element aus *domain_2*. Das Ergebnis der Operation ist in einem 2 dimensionalen Array in *table* gespeichert. Das Ergebnis ist ein Zeiger auf einen Bezeichner, der ein Element aus *domain_3* sein muss.

Folgende Funktionen dienen dem Arbeiten mit Operationen:

- rf_operation_calc
- rf_operation_copy
- rf_operation_create
- rf_operation_destroy
- rf_operation_create_meet
- rf_operation_create_join

Die ersten 4 Funktionen verhalten sich äquivalent zu den Funktionen der Relation. Mit *rf_operation_create_meet* ist es auf Basis einer Kleinergleichrelation möglich eine Operation zu erzeugen, welche als Rückgabewerte das jeweilige *meet* gespeichert hat. Selbiges ist auch mit *rf_operation_create_join* möglich. Auch hier ist eine komplette Liste der Funktionen in der passenden Header-Datei (*operation.h*) zu finden.

### Negation

Eine Negation stellt in diesem Programm eine spezielle Funktion dar, welche als Rückgabewert den negierten Eingabewert liefert. Sie wird gesondert angelegt, da Sie vom Nutzer in der RelaFix -Sprache definiert werden kann. In RelaFix wird eine Negation mit der Struktur *RF_NEGATION* beschrieben.

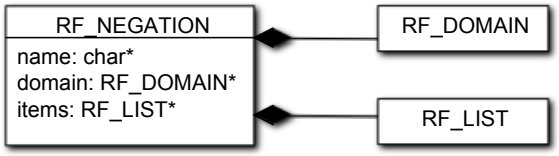



Abbildung 12: Struktur der Negation

*name* ist der Bezeichner der Negation. Dieser Bezeichner wird vom Nutzer in der RelaFix -Sprache definiert und dort zum Aufrufen dieser Negation genutzt. Da sich eine Negation immer auf eine Domäne bezieht, müssen wir auch einen Verweis auf eine solche Speichern. Dies geschieht in *domain*. Die Informationen wie ein Element dieser Domäne zu negieren ist wird in *items* festgehalten. Diese Liste hat ein spezielles Format! Auf ein Element der Domäne folgt immer das negierte Element. Es folgen also immer Pärchen.

$$Liste = (element1, negation1, element2, negation2, ..., elementN, negationN)$$

Es werden wieder die Standardfunktionen zum Erzeugen und Entfernen definiert, sowie einige „Geter-/Seter“ Methoden um Daten zu bearbeiten.

- rf_negation_calc
- rf_negation_create
- rf_negation_destroy
- rf_negation_get_domain
- rf_negation_get_items
- rf_negation_get_name
- rf_negation_has_name
- rf_negation_set_items

### Table

Wie zuvor in der Beschreibung der *Operation* und *Relation* schon angesprochen, benötigen wir eine Tabelle in der die Ergebnisse der *Operation* bzw. die Relationen gespeichert sind. Damit bei der Implementierung von Operationsfunktionen nicht auf den Aufbau und die Implementierung der Tabelle geachtet werden muss, wird diese in eine eigene Struktur (RF_TABLE) ausgelagert. Zudem ermöglicht dies die Strategie wie die Tabelle im Speicher angelegt ist im nach hinein zu ändern ohne andere Programmteile die auf diese Tabelle zurückgreifen, antasten zu müssen. Nachfolgend in Abbildung 13 der Aufbau der Struktur RF_TABLE.

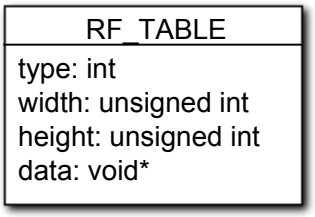



Abbildung 13: Struktur der Tabelle

*type* beschreibt welche Art von Daten unter *data* gespeichert sind. In dieser Version des Programms wird es 2 Arten von Daten geben, die die Tabelle handhaben kann. Dies sind Zeiger auf Zeichenketten und Bits. Die Bits werden benutzt um Relationen platzsparend im Speicher zu halten. Bei der Implementierung wird dann jedes Bit eines Bytes genutzt werden. Im Vergleich zu einem *char*, dem kleinsten Datentypen in C, ergibt sich dann ein 8-fach geringerer Speicherbedarf. *data* stellt ein zweidimensionales Array dar, dessen Ausdehnung in *width* und *height* beschrieben sind.

Mit folgenden Funktionen können Daten geschrieben und gelesen werden. Zudem werden auch Funktionen zur Ermittlung von Eigenschaften aufgelistet:

- rf_table_get_height
- rf_table_get_bit
- rf_table_get_string
- rf_table_get_type
- rf_table_get_width
- rf_table_set_bit
- rf_table_set_string

### Funktionen

Wie in den meisten Computersprachen, gibt es auch in der RelaFix -Sprache die Möglichkeit Funktionen aufzurufen. Diese werden Programmintern mit der Struktur RF_FUNCTION abgebildet. Funktionen werden programmintern beschrieben und können daher nicht vom Nutzer definiert werden. Neben ihrem Namen enthalten Sie einen Zeiger auf die zugehörige C-Funktion und eine Angabe wie viele Argumente diese Funktion übergeben bekommt. Zudem enthält diese Struktur auch eine Beschreibung der Funktion, welche in der RelaFix -Sprache durch die Angabe des Funktionsnamens angezeigt werden kann.

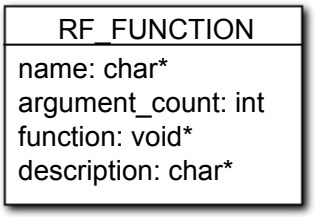



Abbildung 14: Struktur für Funktionen

Auch für diese Struktur existieren wieder Funktionen zum Erzeugen und Zerstören, sowie „Geter- / Seter-“ Methoden um auf die Daten zugreifen zu können.

### Formula

Ein wichtiger Teil des Programms ist das Auswerten von Ausdrücken die der Nutzer angegeben hat. Dazu gibt es zwei Möglichkeiten. Es kann der Ausdruck während des parsens schrittweise berechnet werden oder man parst den kompletten Ausdruck und speichert ihn in einer geeigneten Struktur ab bevor man ihn auswertet. Da es in der RelaFix -Sprache die Möglichkeit gibt Operationen als eine Formel mit Variablen zu definieren, muss es möglich sein diese Formel für alle Variablenkombinationen ausrechnen zu können. Dies ist mit dem Auswerten während des parsens nicht machbar. Daher wird bei dieser Implementierung ein Ausdruck in einer speziellen Struktur zwischengespeichert und im Anschluss ausgewertet.

Diese spezielle Struktur stellt eine Art Baum dar. Jeder Knoten (RF_FORMULA) in diesem Baum enthält ein Element, welches zum Beispiel eine *Operation* sein kann. Blätter des Baumes sind Knoten, welche einen Bezeichner (ID) beinhalten. Argumente einer Operation, Funktion, ... werden in einem Array festgehalten. Die Argumente in dem Array sind wiederum auch Knoten (RF_FORMULA), somit sind sie die Kindknoten.

Um bei der Fehlerbehandlung die Position an der der Fehler aufgetreten ist ermitteln zu können, wird beim Parsen in jedem Knoten die Position festgehalten. Die Struktur RF_FORMULA ist wie in Abbildung 15 definiert:

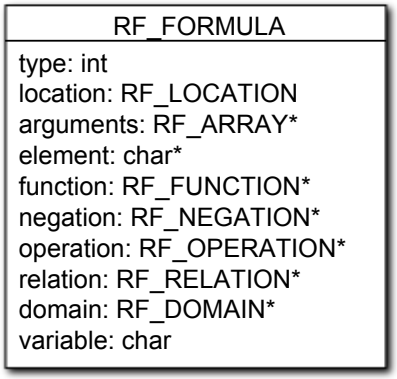



Abbildung 15: Struktur eines Knoten des Formulabaums

Wenn im Programm ein Formula-Baum vorliegt, kann dieser mit der Funktion *rf_formula_calc()* ausgewertet werden. Abbildung 16 zeigt wie ein Ausdruck im Baum abgebildet wird.

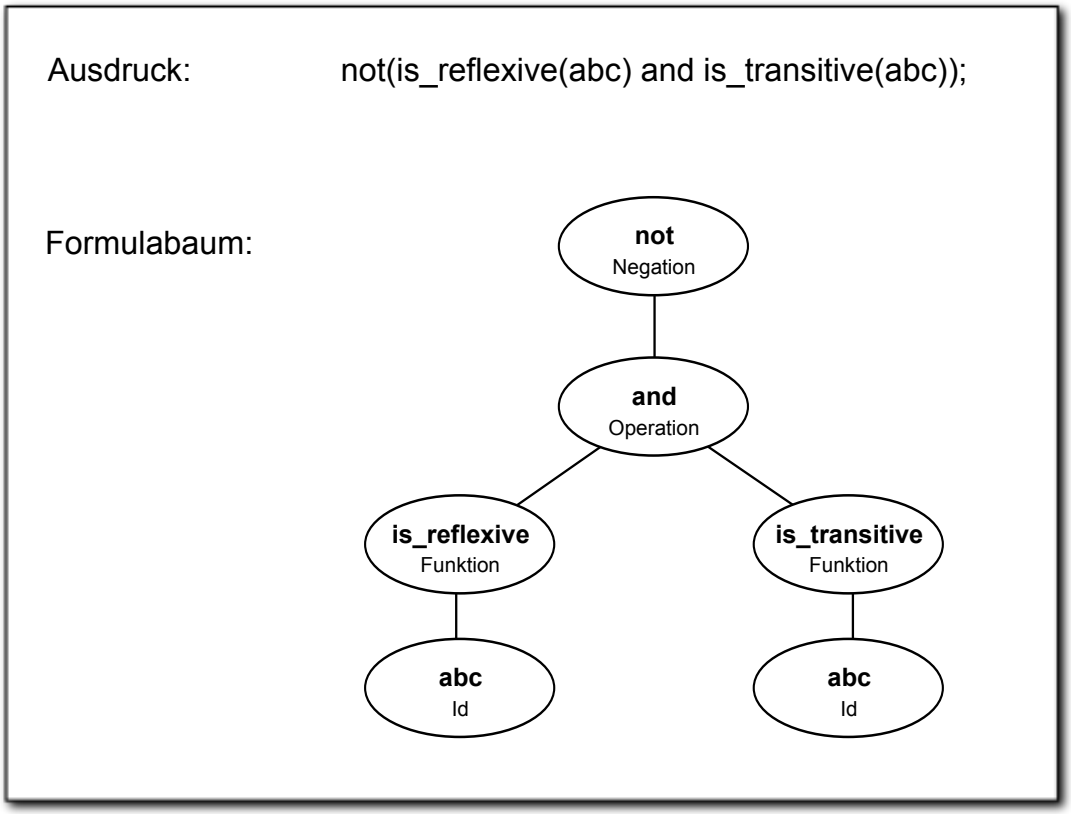



Abbildung 16: Abbildung eines Ausdrucks im Formulabaum

## 4.3 Algorithmen

### Meet

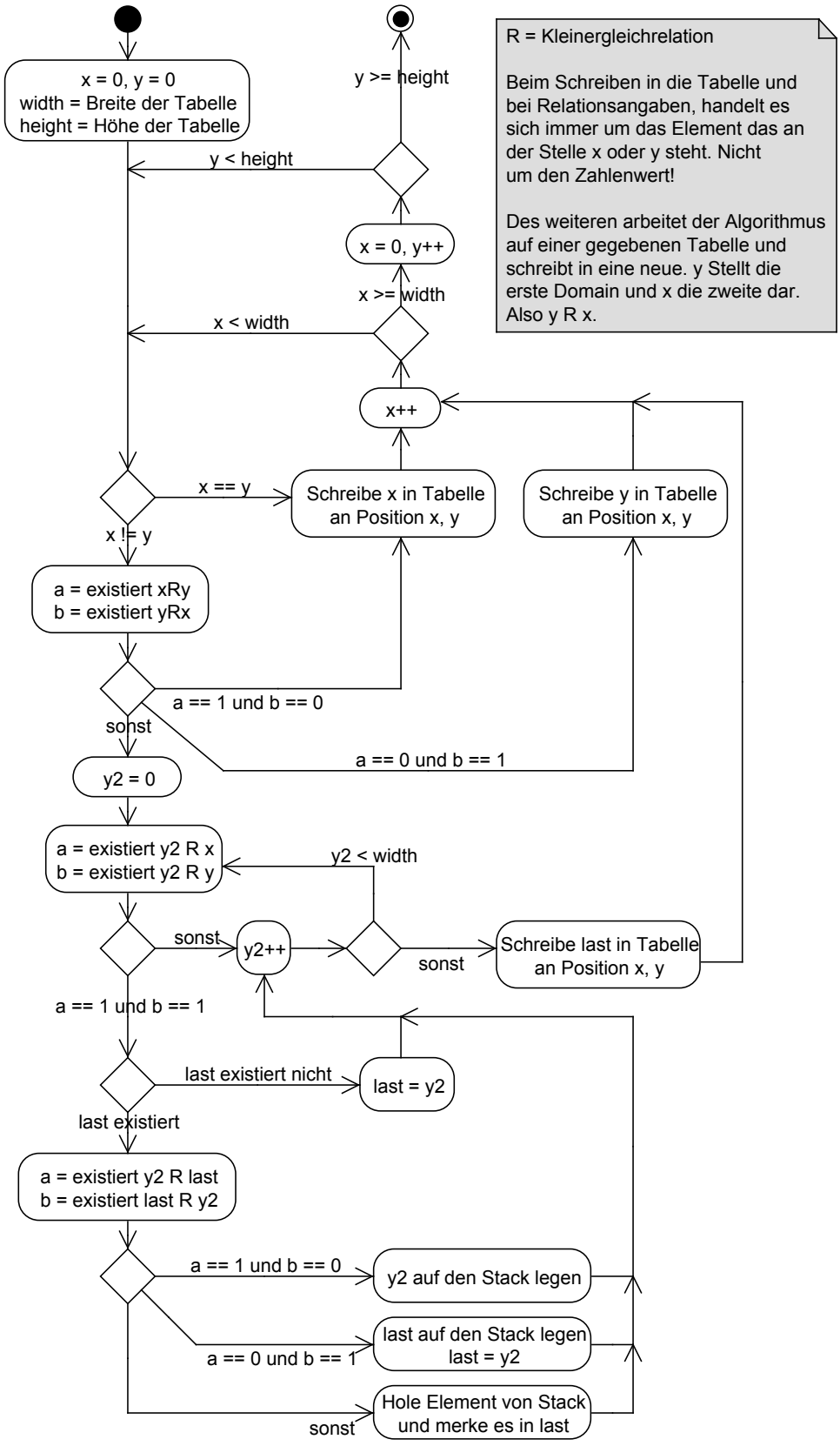



Abbildung 17: Algorithmus zum Erzeugen der meet-Operation

### Join

Der join-Algorithmus ist im Aufbau mit dem meet-Algorithmus identisch. Es muss lediglich an paar Stellen die Verzweigung vertauscht werden.

### Difunktional

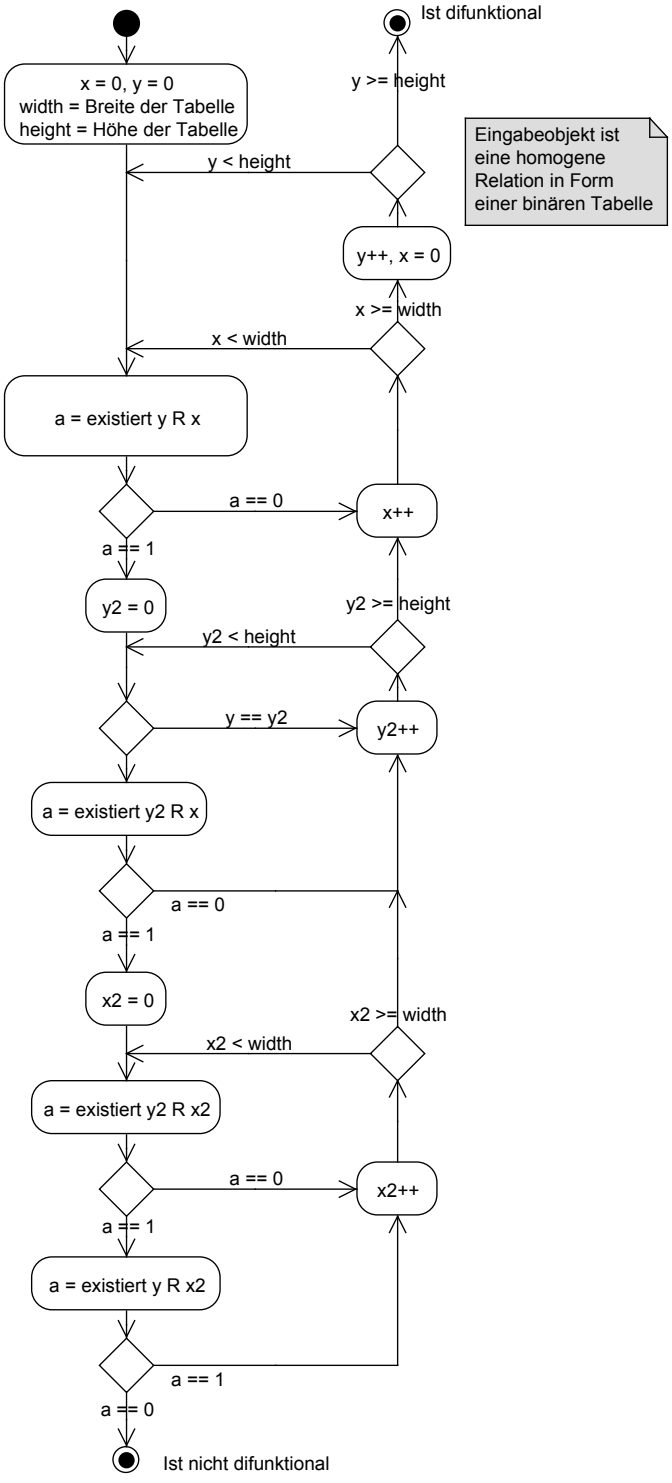



Abbildung 18: Algorithmus zum Testen auf Difunktionalität

### Transitivität

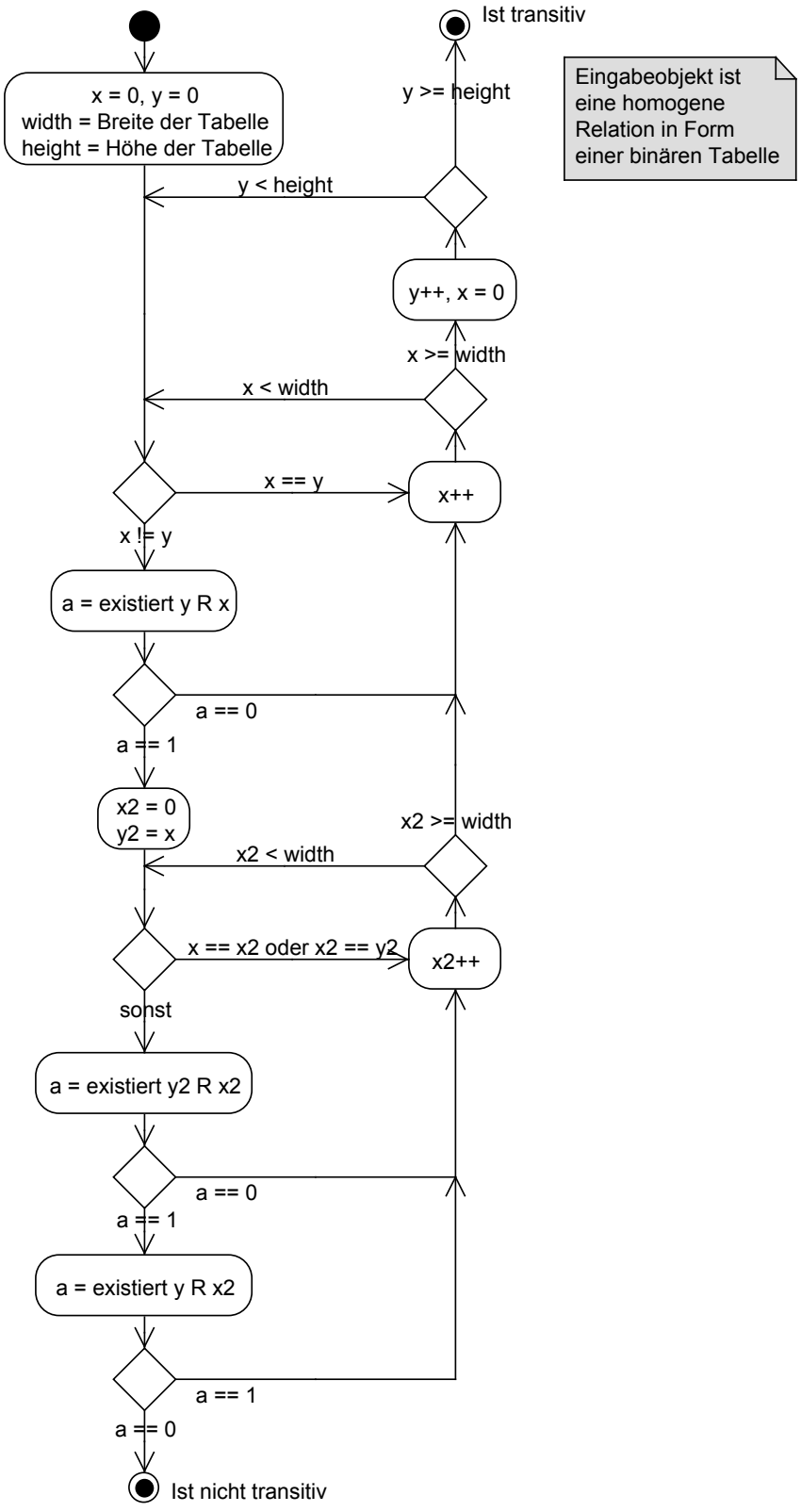



Abbildung 19: Algorithmus zum Testen auf Transitivität

## Konkatenation

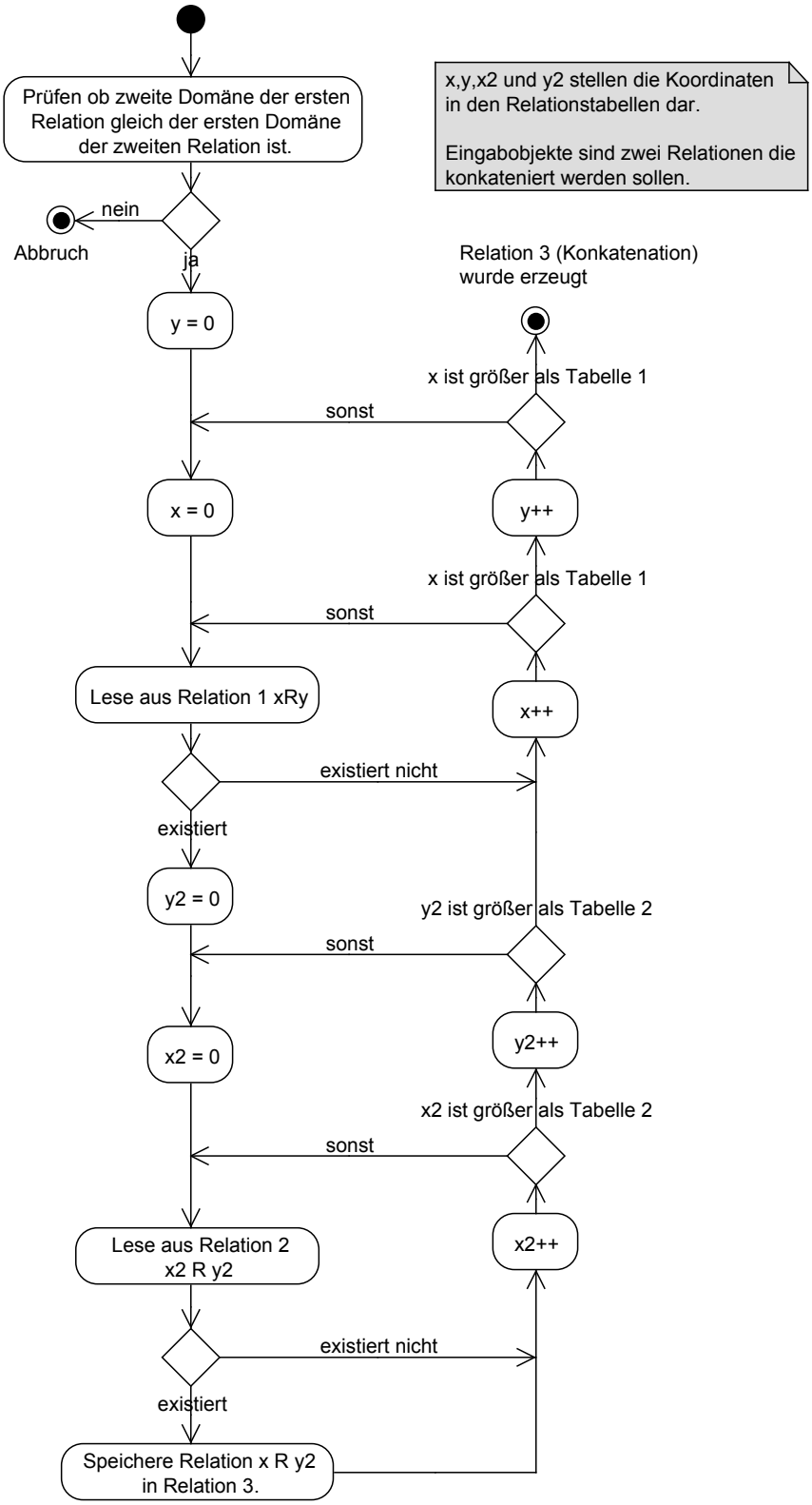



Abbildung 20: Algorithmus zum Konkatenieren zweier Relationen

# 5 Test / Evaluation

In diesem Abschnitt der Arbeit wird die Funktionalität getestet und überprüft ob die gegebenen Anforderungen erfüllt sind. Getestet wird ob das Programm die Sprache korrekt einliest. Des weiteren muss auch überprüft werden ob die angewandten mathematischen Funktionen die korrekten Ergebnisse liefern.

Zur Überprüfung ob die Sprache richtig eingelesen wird, wird für jedes Element der Sprache eine Testdatei angelegt und mit ihr das Programm getestet. Zudem werden auch Testdateien angelegt die Fehler enthalten um zu überprüfen, ob diese auch korrekt erkannt werden. Die Dateien finden sich im Ordner *testfiles* und sind dort jeweils in die Ordner *ok* und *error* aufgeteilt.

In späteren Versionen könnten diese Tests mit Hilfe eines Shellscripts automatisiert werden.

## 5.1 Funktionstest

Im folgenden werden die Ergebnisse der mathematischen Funktionen überprüft:

### is_reflexive

Eine Relation $R : A \to A$ ist reflexiv, wenn gilt: $\forall x \in A, xRx$

Die Ausgabe in Abbildung 21 zeigt das die Funktion korrekt funktioniert.

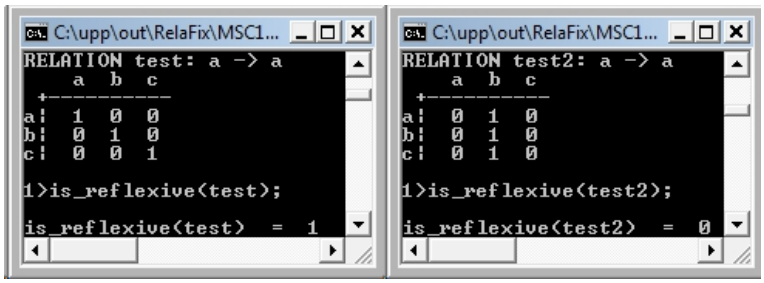



Abbildung 21: Test auf Reflexivität

### is_irreflexive

Eine Relation $R : A \to A$ ist irreflexiv, wenn gilt: $Id_A \cap R = \emptyset$
Test in Abbildung 22.

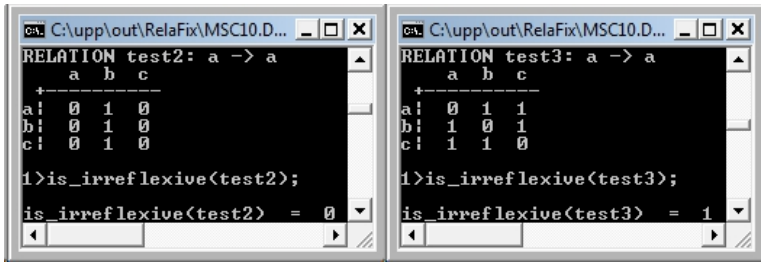



Abbildung 22: Test auf Irreflexivität

### is_symmetric

Eine Relation $R : A \to A$ ist symmetrisch, wenn gilt: $\forall x, y \in A, xRy \to yRx$
Test in Abbildung 23.

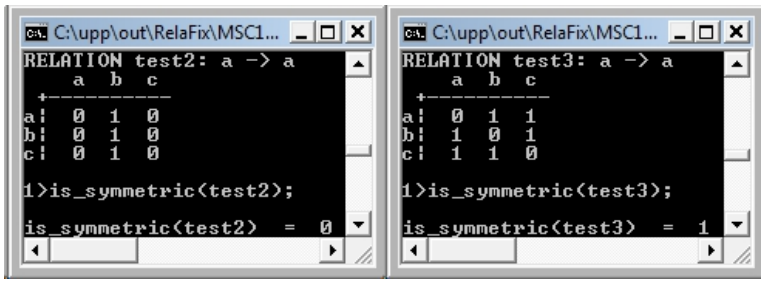



Abbildung 23: Test auf Symmetrie

### is_transitive

Eine Relation $R : A \to A$ ist transitiv, wenn gilt: $\forall x, y, z \in A, xRy \wedge yRz \to xRz$
Test in Abbildung 24.

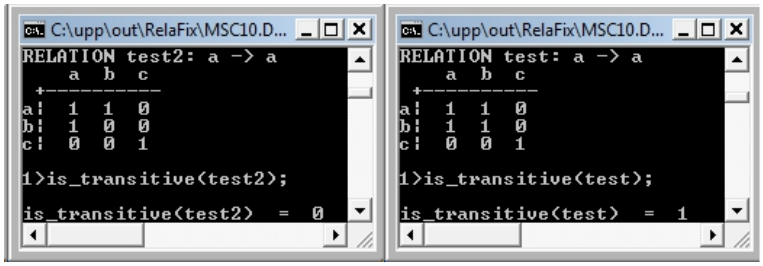



Abbildung 24: Test auf Transitivität

### is_antisymmetric

Eine Relation $R : A \to A$ ist antisymmetrisch, wenn gilt: $\forall x, y \in A, x \neq y, xRy \to \neg yRx$
Test in Abbildung 25.

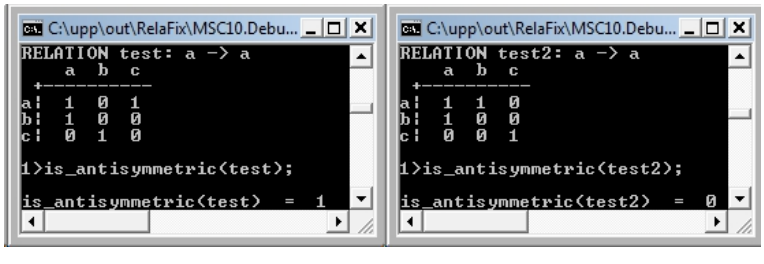



Abbildung 25: Test auf Antisymmetrie

### is_asymmetric

Eine Relation $R : A \to A$ ist asymmetrisch, wenn sie antisymmetrisch und irreflexiv ist.
Test in Abbildung 26.

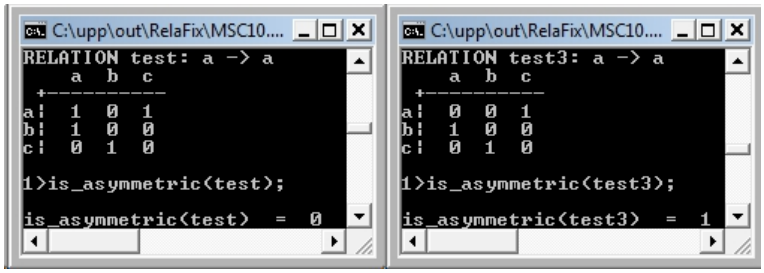



Abbildung 26: Test auf Asymmetrie

### is_equivalent

Eine Relation $R : A \to A$ ist äquivalent, wenn sie reflexiv, symmetrisch und transitiv ist.
Test in Abbildung 27.

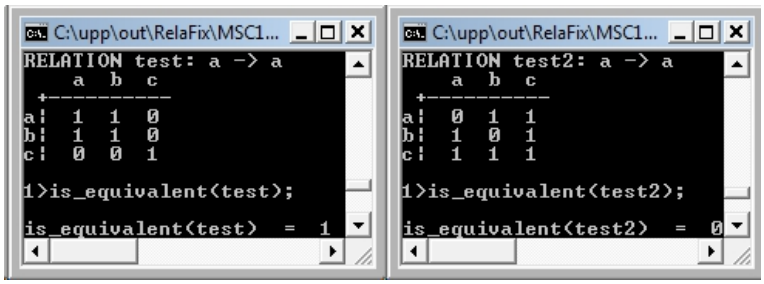



Abbildung 27: Test auf Äquivalenz

### is_difunctional

Eine Relation $R : A \to A$ ist difunktional, wenn gilt: $\forall x, y, z, w \in A, xRy \wedge zRy \wedge zRw \to xRw$
Test in Abbildung 28.

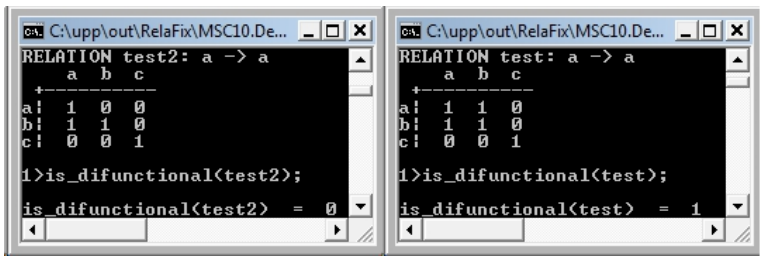



Abbildung 28: Test auf Difunktionalität

### is_preorder

Eine Relation $R : A \to A$ hat die Eigenschaft „preorder“ wenn sie reflexiv und transitiv ist.
Test in Abbildung 29.

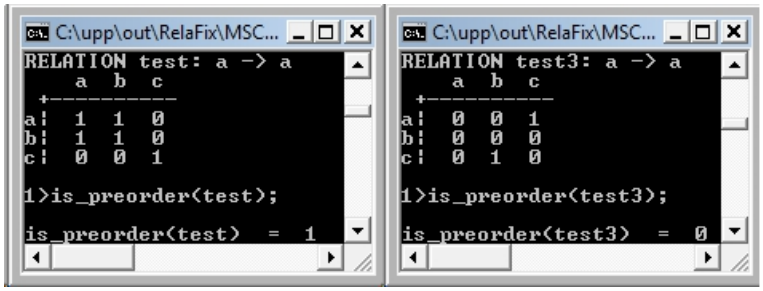



Abbildung 29: Test auf Preorder-Eigenschaft

### is_poset

Eine Relation $R : A \to A$ hat die Eigenschaft „poset“ wenn sie reflexiv, antisymmetrisch und transitiv ist.
Test in Abbildung 30.

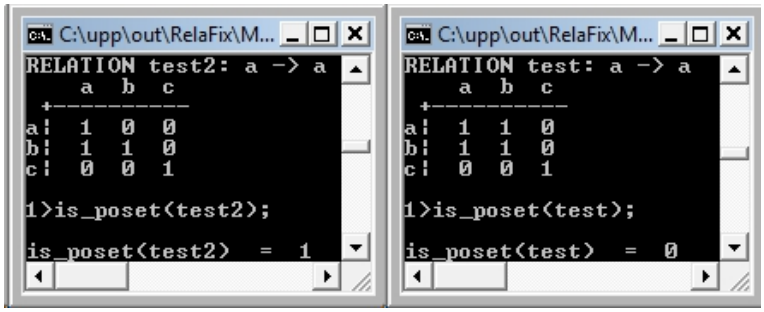



Abbildung 30: Test auf Poset-Eigenschaft

### id

Erzeugt eine neue Relation für die gilt: $xRx \forall x \in Domain$
Test in Abbildung 31.

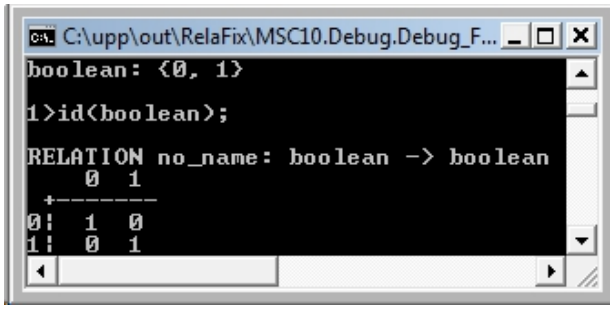



Abbildung 31: Test der Funktion id

### empty

Erzeugt eine neue leere Relation basierend auf Domänen
Test in Abbildung 32.

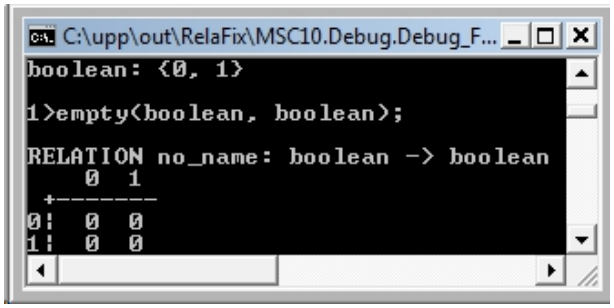



Abbildung 32: Test der Funktion empty

### full

Erzeugt eine neue Relation basierend auf Domänen, für die alle $xRy$ existieren Test in Abbildung 33.

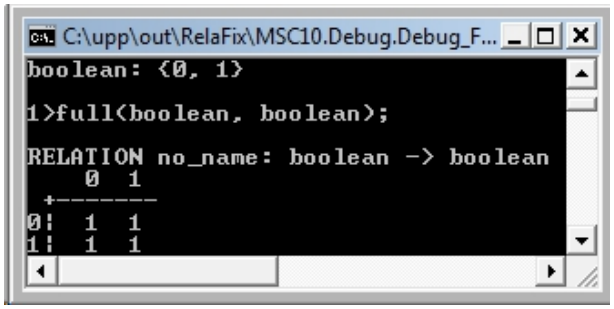



Abbildung 33: Test der Funktion full

### complement

Das Komplement einer Relation $R : A \to A$ ist gegeben wenn gilt: $\bar{R} = A \times A \setminus R$ Test in Abbildung 34.

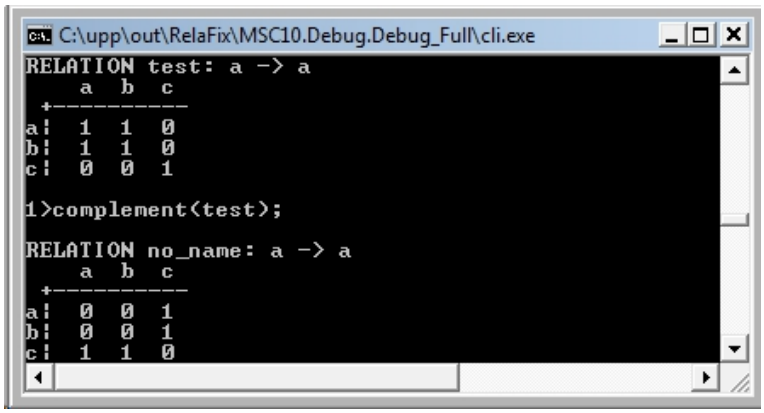



Abbildung 34: Test der Funktion „complement“

### converse

Das Inverse einer Relation $R : A \to A$ ist gegeben wenn gilt: $\forall x, y \in A, xRy \to yR^{-1}x$ Test in Abbildung 35.

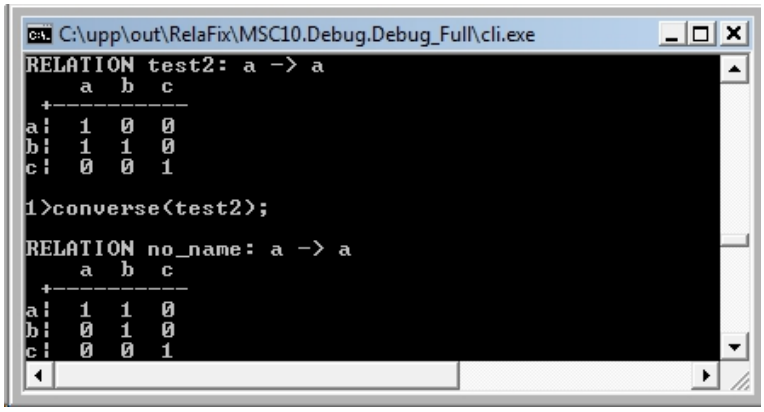



Abbildung 35: Test der Funktion „converse“

### union

Die Vereinigung zweier Relationsmengen enthält alle Relationen aus der ersten und aus der zweiten Menge.
Test in Abbildung 36.

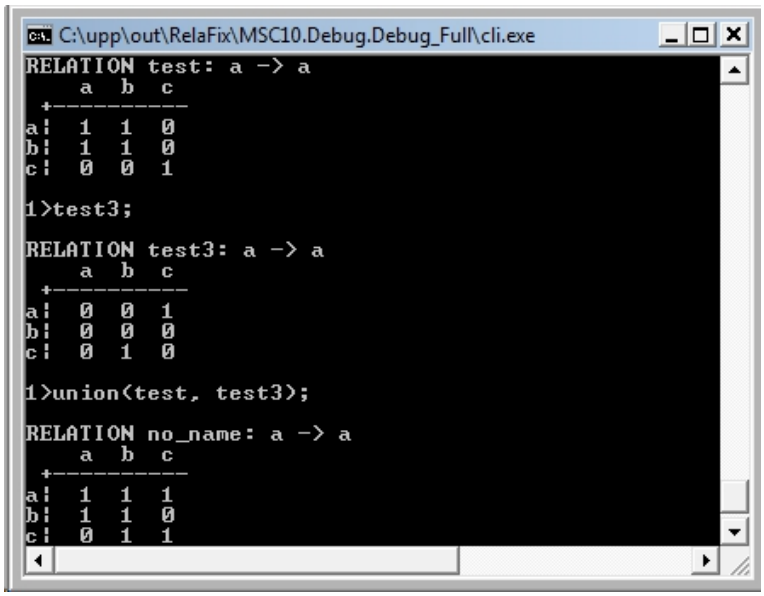



Abbildung 36: Test der Funktion „union“

### intersection

Der Schnitt zweier Relationsmengen enthält nur die Relationen, die sowohl in der ersten Menge als auch in der zweiten Menge enthalten sind.
Test in Abbildung 37.

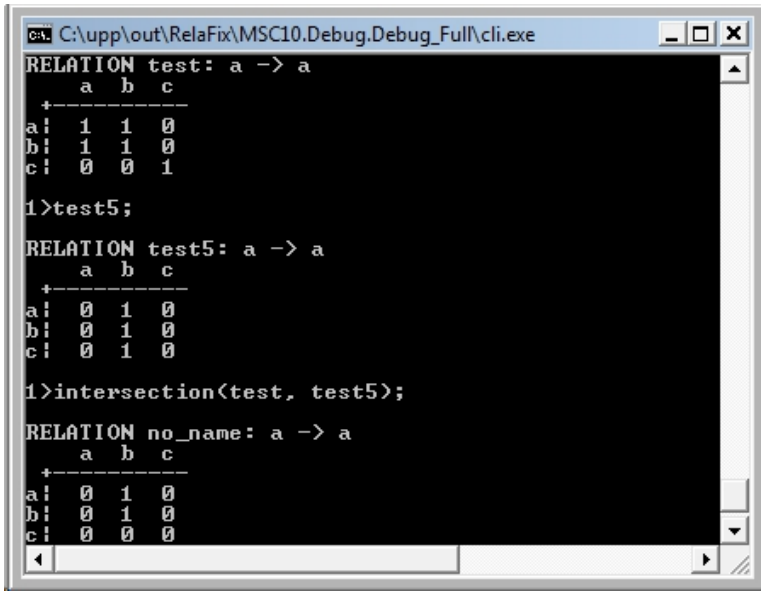



Abbildung 37: Test der Funktion „intersection“

### concat

$R : A \to B$
$RR : B \to C$
$S := R \circ RR = \{< x, z > | x \in A \wedge z \in C \wedge \exists y \in B \wedge xRy \wedge yRz\}$
Test in Abbildung 38.

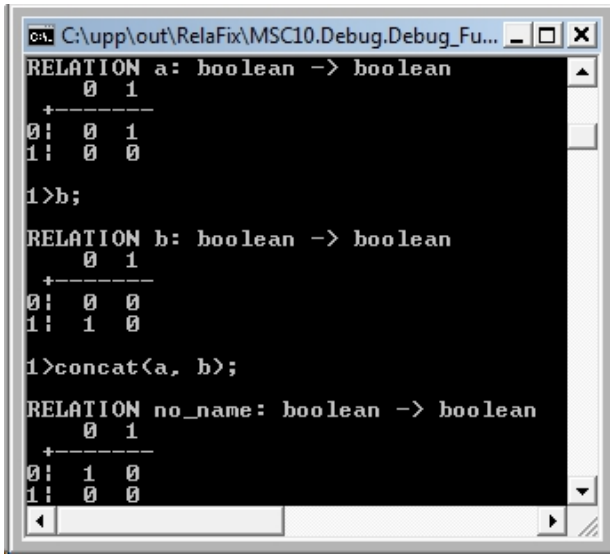



Abbildung 38: Test der Funktion „concat“

### max

Diese Funktion findet das größte Element einer $\leq$ Relation.
Test in Abbildung 39.

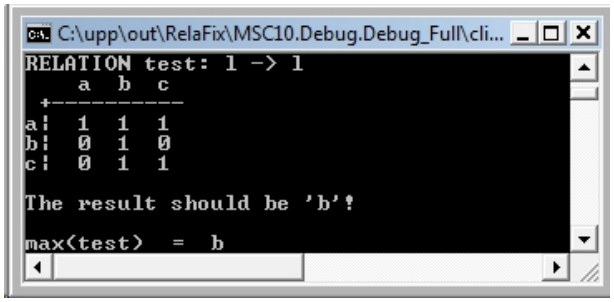



Abbildung 39: Test der Funktion „max“

### min

Diese Funktion findet das kleinste Element einer $\leq$ Relation.
Test in Abbildung 40.

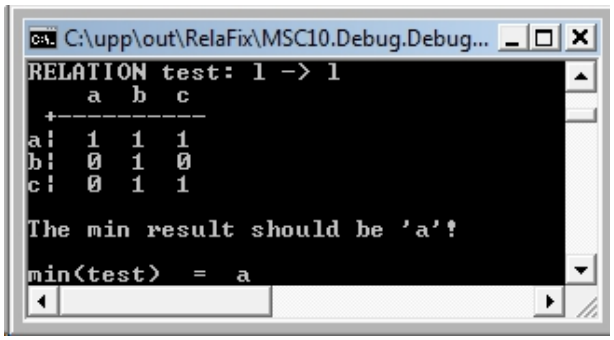



Abbildung 40: Test der Funktion „min“

### upperbound

Diese Funktion findet alle oberen Schranken für gegebene Elemente.
Test in Abbildung 41.

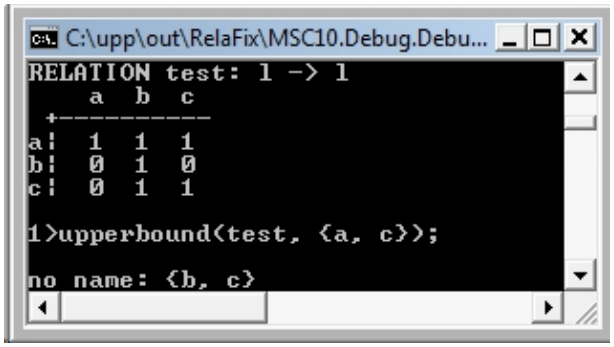



Abbildung 41: Test der Funktion „upperbound“

### lowerbound

Diese Funktion findet alle unteren Schranken für gegebene Elemente.
Test in Abbildung 42.

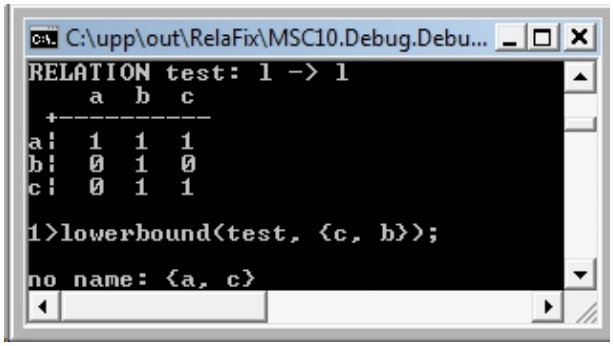



Abbildung 42: Test der Funktion „lowerbound“

### supremum

Diese Funktion findet das kleinste Element einer gegebenen Teilmenge.
Test in Abbildung 43.

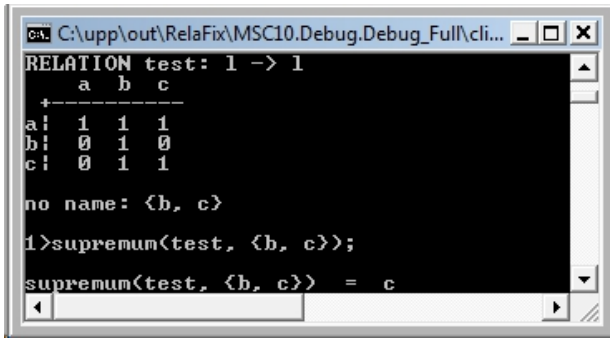



Abbildung 43: Test der Funktion „supremum“

### infimum

Diese Funktion findet das größte Element einer gegebenen Teilmenge.
Test in Abbildung 44.

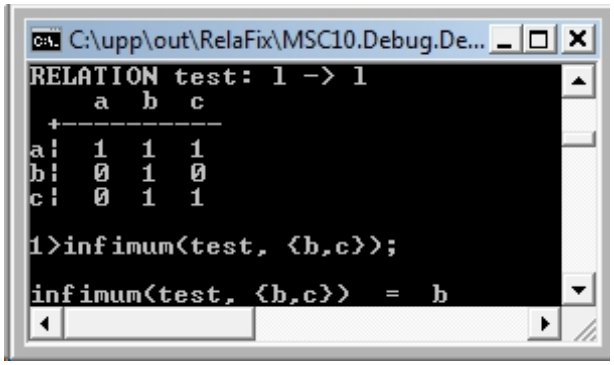



Abbildung 44: Test der Funktion „infimum“

### generate_meet

Diese Funktion generiert auf Basis einer $\leq$ Relation eine Funktion, welche das größte niedrigste gemeinsame eindeutige Element zweier Eingabeelemente liefert.

Test in Abbildung 45.

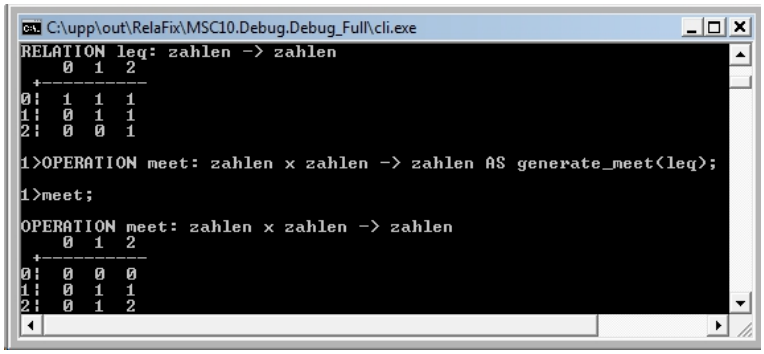



Abbildung 45: Test der Funktion „generate_meet“

### generate_join

Diese Funktion generiert auf Basis einer $\leq$ Relation eine Funktion, welche das kleinste größte gemeinsame eindeutige Element zweier Eingabeelemente liefert.
Test in Abbildung 46.

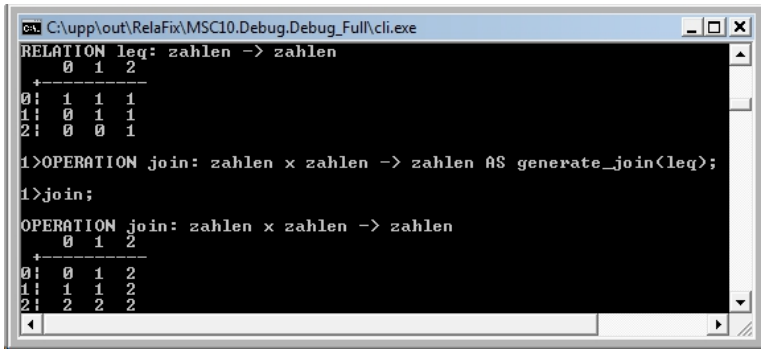



Abbildung 46: Test der Funktion „generate_join“

## 5.2 Überprüfung der Anforderungen

| Nr. | ✓/✗ | Beschreibung |
|---|---|---|
| 1 | ✓ | Alles in C |
| 2 | ✓ | Es wird nur ein externes Programm genutzt. Dieses ist OpenSource und in C. |
| 3 | ✓ | |
| 4 | ✓ | Sowohl über Dateien als auch in Konsole |
| 5 | ✓ | |
| 6 | ✓ | Es werden Position, Datei und Namen zur Fehlerbehandlung angegeben |
| 7 | ✓ | |
| 8 | ✓ | `DOMAIN a AS {a, b, c};` |
| 9 | ✓ | `DELETE a;` |
| 10 | ✓ | `RELATION leq: ... AS TABLE ...` |
| 11 | ✓ | |

| | | |
|---|---|---|
| 12 | ✓ | `OPERATION plus: ... AS TABLE ...` |
| 13 | ✓ | `OPERATION plus: ... AS FORMULA ...` |
| 14 | ✓ | |
| 15 | ✓ | `NEGATION not: boolean AS 0/1, 1/0;` |
| 16 | ✓ | |
| 17 | ✓ | `LOAD test.rfc;` |
| 18 | ✓ | `PATH c:\abc\;` |
| 19 | ✓ | Relationen und Operationen sind fest mit Domänen verknüpft, daher muss bei der Berechnung keine Domäne angegeben werden. |
| 20 | ✓ | `a + (b - c);` |
| 21 | ✓ | `a * is_reflexive(leq);` |
| 22 | ✓ | Durch Angabe des Funktionsnamens wird diese Angezeigt. |
| 23 | ✓ | `a leq b;` |
| 24 | ✓ | `is_antisymmetric();` |
| 25 | ✓ | `is_asymmetric();` |
| 26 | ✓ | `is_difunctional();` |
| 27 | ✓ | `is_equivalent();` |
| 28 | ✓ | `is_irreflexive();` |
| 29 | ✓ | `is_poset();` |
| 30 | ✓ | `is_preorder();` |
| 31 | ✓ | `is_reflexive();` |
| 32 | ✓ | `is_symmetric();` |
| 33 | ✓ | `is_transitive();` |
| 34 | ✓ | `complement(leq);` |
| 35 | ✓ | `converse(leq);` |
| 36 | ✓ | `union(leq, id);` |
| 37 | ✓ | `intersection(leq, id);` |
| 38 | ✓ | `generate_meet(leq);` |
| 39 | ✓ | `generate_join(leq);` |
| 40 | ✓ | `id();` |
| 41 | ✓ | `empty();` |
| 42 | ✓ | `full();` |
| 43 | ✓ | `concat();` |
| 44 | ✓ | `max();` |
| 45 | ✓ | `min();` |
| 46 | ✓ | `upperbound();` |
| 47 | ✓ | `lowerbound();` |
| 48 | ✓ | `supremum();` |
| 49 | ✓ | `infimum();` |
| 50 | ✓ | |
| 51 | ✓ | Sind mit Hilfe von Doxygen dokumentiert und steht als HTML-Dokumentation zur Verfügung. |
| 52 | ✓ | In diesem Dokument. Kapitel 3.2 |
| 53 | ✓ | |

# 6 Dokumentation

Neben den Kommentaren im Quellcode, wurden alle Funktionen und Strukturen dokumentiert. Diese Codedokumentation steht auch direkt im Quellcode. Es werden spezielle Kommentare genutzt, aus denen mit Hilfe des Programms Doxygen (`www.doxygen.org/`) eine HTML-Dokumentation erstellt werden kann. Die Einstellungen für Doxygen sind im *source* Verzeichnis in der Datei *Doxyfile* gespeichert.

Als Spezifaktion der domänenspezifischen Sprache dient Kapitel 3.2 in dem die Sprache geplant wurde. Dort sind alle Informationen enthalten.

Zum Kompilieren des Programms wird nur ein C-Compiler und das Makekommando benötigt. Ein Aufruf des Makefiles im Hauptverzeichnis sollte einen neuen Ordner *release* erzeugen, welcher das Programm mit den benötigten Dateien enthält. Gegebenenfalls muss das Makefile auf das jeweilige System angepasst werden (Compiler, Make, Dateipfade). Wenn Änderungen am Parser vorgenommen werden sollen, wird noch das Programm *bison* benötigt, mit dem die Datei *parser.y* in einen Parser umgewandelt werden muss. Dies geschieht mit dem Aufruf: `bison -d parser.y`.

Das fertige Programm kann in zwei Modi aufgerufen werden. Im ersten Modus, wenn das Programm ohne Parameter aufgerufen wird, startet es im Kommandozeilenmodus und nimmt eingaben vom Nutzer entgegen. Wird das Programm hingegen mit einer RelaFix -Datei als Parameter gestartet, so befindet es sich im zweiten Modus in dem es die Befehle in dieser Datei abarbeitet und sich danach wieder beendet.

`rfcli.exe` oder `rfcli.exe <file>`

Die mit dem Programm mitgelieferten Algebren wie zum Beispiel Heyting, können mit dem Befehl LOAD geladen werden. Beispiel: `LOAD heyting.rfc;`. Bei der Definition eigener Dateien ist darauf zu achten, dass sie nur Zeichen aus dem ASCII-Zeichensatz enthalten. Möchte man Befehle bei Programmstart immer automatisch ausführen lassen, so kann man diese in die Datei *system.rfc* schreiben. Die vorhandenen Befehle dürfen aber nicht verändert werden!

# 7 Zusammenfassung

In dieser Arbeit wurde eine erste Version eines Programms und einer Sprache geschaffen, die dem Mathematiker oder Informatiker beim Analysieren von Relationen hilft. Es ist möglich Domänen, Relationen und Operationen zu erzeugen und mit ihnen zu Rechnen oder ihre Eigenschaften zu analysieren. Des weiteren können auch Dateien hinzu geladen werden.

Zur Vorgehensweise wurde das Phasenmodell gewählt in dem das Projekt in verschiedene Phasen eingeteilt wurde, welche zu einem bestimmten Datum abgeschlossen sein müssen. Anhand dieser Phasen erfolgten dann die Analye, der Entwurf, die Implementierung und die Evaluation.

In der Analysephase wurden die Elemente der Anforderungsliste näher betrachtet. Es wurden mathematische Begriffe erläutert und auf die Eigenheiten einer domänenspezifischen Sprache eingegangen. Auch wurde betrachtet welcher Parser für dieses Projekt geeignet ist.

Beim Entwurf und der Implementierung wurde darauf geachtet, dass das Programm portabel bleibt. So wurde zum Beispiel auf den Einsatz einer externen Programmbibliothek für Listen und Arrays verzichtet. Auch wurde das Programm so konzipiert, dass die Benutzerschnittstelle vom restlichen Programm abgetrennt ist und somit leicht zu ersetzen ist. Auf diese Weise kann zum Beispiel schnell eine graphische Benutzeroberfläche geschrieben werden, ohne am restlichen Programm Änderungen vornehmen zu müssen.

Eine Codedokumentation des Projekts wurde mit Hilfe des Programms Doxygen erstellt. Dadurch ist es möglich die Dokumentation direkt in den Quellcode zu integrieren.

Zum Abschluß des Projekts wurde das Programm noch getestet und überprüft ob alle Anforderungen erfüllt wurden.

Mit Blick in die Zukunft, könnte das Programm mit einer graphischen Benutzerschnittstelle ausgestattet werden. Oder aber auch eine RelaFix zu Latex Variante implementiert werden. Des weiteren können unkompliziert komplexere Funktionen zur Sprache hinzugefügt werden, da dafür am Grundgerüst der Sprache keine Änderungen vorgenommen werden müssen.

# 8 Literaturverzeichnis


| | |
|---|---|
| Berghammer, R. | „Ordnungen, Verbände und Relationen mit Anwendungen”. Vieweg + Teubner, Wiesbaden 2008 |
| Fowler, M. | „DomainSpecificLanguage”. `http://www.martinfowler.com/bliki/DomainSpecificLanguage.html` 24.01.2012 |
| Free Software Foundation Inc. | „Bison - GNU parser generator”. `http://www.gnu.org/software/bison/` 24.01.2012 |
| Louis, D. | „C/C++ KOMPENDIUM”. Markt + Technik Verlag 2004 |
| Müller, M. E. | „Relational Knowledge Discovery”. Cambridge University Press 2012 |
| Witt, K. U. | „Mathematische Grundlagen für die Informatik”. Vorlesungsskript, Hochschule Bonn-Rhein-Sieg 2007 |

# Anhang

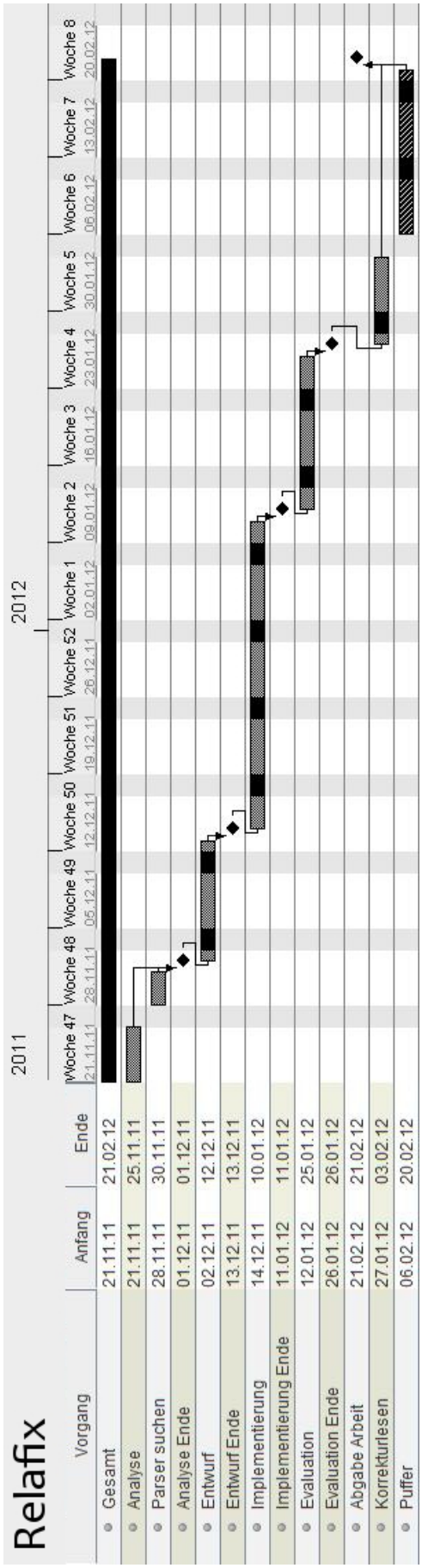



Abbildung 47: Zeitplan