Interpreter für Operationen auf Fuzzy-Mengen (und scharfe Mengen)

Wolfgang Bongartz, Sommer 2015

Das Programm erlaubt die Definition von Mengen und die Anwendung von Operationen auf diese Mengen. Die Kommandos werden entweder direkt von der Standardeingabe gelesen und sofort ausgeführt oder zeilenweise aus einer Textdatei entnommen, deren Dateiname beim Programmstart als Parameter angegeben wird. Jede Eingabe schließt mit einem Semikolon und dem Zeilenende ab. Die Ausgaben des Programms erfolgen über die Standardausgabe.

1 Benutzung des Programms

Start des Programms:

```
java -jar FuzzySetInterpreter.jar
```

Start des Programms mit einem Skriptfile:

```
java -jar FuzzySetInterpreter.jar test.set
```

Es handelt sich um ein Terminalprogramm, dass jede vollständig eingegebene Anweisung verarbeitet, sobald die Return-Taste betätigt wird. Alternativ kann als Kommandozeilenparameter auch eine Datei angegeben werden, deren Inhalt dann verarbeitet wird. Die möglichen Anweisungen werden im Abschnitt 3 erläutert. Mit der Anweisung `exit;` wird das Programm verlassen.

2 Die Kommandos der Skriptsprache

Es gelten die folgenden Konventionen:

- Groß- und Kleinschreibweise werden unterschieden
- Alle Kommandos werden durchgängig klein geschrieben
- Bezeichner für Mengen werden in Großbuchstaben geschrieben

2.1 Datentypen

Das Programm unterstützt die im Folgenden aufgeführten Datentypen, die in Anweisungen verwendet werden können.

2.1.1 Zeichenketten

Zeichenketten werden von Anführungszeichen eingeschlossen. Eine Zeichenkette kann alle Zeichen außer dem Anführungszeichen und dem Backslash enthalten.

Beispiel: `„Hallo Welt!“`

2.1.2 Ganze Zahlen

Positive ganze Zahlen einschließlich der Null.

2.1.3 Reelle Zahlen

Positive reelle Zahlen einschließlich der Null. Achtung: Zur Trennung zwischen Vor- und Nachkommastellen wird ein Punkt verwendet.

Beispiel: `1507.45`

2.1.4 Mengen

Eine Menge kann durch Angabe ihrer Elemente definiert werden. Elemente scharfer Mengen werden durch Kommata getrennt in geschweiften Klammern aufgelistet. Zur Kennzeichnung von Fuzzy-Mengen wird einer solchen Auflistung das Zeichen ‚#' vorangestellt.

Elemente scharfe Mengen:

- Element-Bezeichner (in Kleinbuchstaben)
  ```
  C := { a, d, e, fa, gb };
  ```
- Ganze Zahlen
  ```
  N := { 1, 2, 4, 5};
  ```
- Reele Zahlen
  ```
  R := { 5.1, 3.2, 3.14, 5.0};
  ```
- Wertepaare
  ```
  P := { (a,a), (a,b), (a,c) };
  ```

Elemente von Fuzzy-Mengen:

- Wertepaare aus einem der oben aufgeführten Element einer scharfen Menge und dem dazugehörigen µ-Wert
  ```
  F := #{ [a,0.1], [b,0.75], [c,1] };
  ```
- Falls für eine Fuzzy-Menge nur Elemente für scharfe Mengen angegeben werden so wird jeweils ein „µ-" von 1 angenommen
  ```
  G := #{a,b,c};
  ```

Im Programm müssen Mengen die folgenden Bedingungen erfüllen:

- „Sortenreinheit": Eine Menge darf nur Elemente eines Typs enthalten. Gemischte Mengen sind also nicht möglich[1]
- „Eindeutigkeit": Elemente dürfen in einer Menge nicht mehrmals vorkommen[2]

2.2 Ausdrücke

Für alle Beispiele in diesem Abschnitt werden die folgenden Variablendefinitionen verwendet:

```
A  := true;
B  := false;
$A := #{ [a,0.1], [b,0.75] };
$B := #{ [a,0.9], [b,0.25] };
```

---

[1] Die folgende Definition würde im Programm also zu einem Fehler führen: `G := {a,1,3.14};`

[2] Die Definition `F := {1,1,2,3,5,8};` ist zwar möglich, da das Element ‚1' doppelt vorkommt ist die Definition aber äquivalent zu : `F := {1,2,3,5,8};`

2.2.1 logische Ausdrücke

Operanden können sein:

- Die booleschen Konstanten `true` und `false`
- Das Ergebnis eines logischen Ausdrucks (Ausdrücke können also geschachtelt werden)

Folgende logische Operationen werden unterstützt:

- and
  Beispiel: `print A and true;`
- or
  Beispiel: `print A or B;`
- xor
  Beispiel: `print A xor B;`
- not
  Beispiel: `print not B;`

Folgende Vergleichsoperationen werden unterstützt (die Operanden müssen arithmetische Ausdrücke sein):

- „größer als“ (>)
  Beispiel: `print 7 > 5;`
- „kleiner als“ (<)
  Beispiel: `print 5 < 7;`
- „gleich“ (==)
  Beispiel: `print 7 == ( 5 + 2 );`
- „größer als oder gleich“ (>=)
  Beispiel: `print 7 >= 5;`
- „kleiner als oder gleich“ (<=)
  Beispiel: `print 5 <= 7;`

Folgende Mengenoperationen werden unterstützt:

- „<operand> ist leere Menge“
  Syntax: `set <operand> is empty`
  Beispiel: `print set { } is empty;`
- „<op1> ist Schnittmenge von <op2> und <op3>“
  Syntax: `set <op1> is intersection of <op2> with <op3>`
  Beispiel: `print set #{ [a,0.1], [b,0.25] } is intersection of $A with $B;`
- „<op1> ist Vereinigungsmenge von <op2> und <op3>“
  Syntax: `set <op1> is union of <op2> with <op3>`
  Beispiel: `print set #{ [a,0.9], [b,0.75] } is union of $A with $B;`
- „<op1> ist Teilmenge von <op2>“
  Syntax: `set <op1> is subset of <op2>`
  Beispiel: `print set #{ [a,0.05] } is subset of $A;`

- „<op1> ist das symmetrische Komplement von <op2> und <op3>“
  Syntax: `set <op1> is symmetric complement of <op2> with <op3>`
  Beispiel: `print set #{ [a,0.8],[b,0.5] } is symmetric complement of $A with $B;`
- „<op1> ist das Komplement von <op2>“
  Syntax: `set <op1> is complement of <op2>`
  Beispiel: `print set $A is complement of $B;`
- „<op1> ist das Komplement von <op2> und <op3>“
  Syntax: `set <op1> is complement of <op2> with <op3>`
  Beispiel: `print set #{ [a,0], [b,0.50] } is complement of $A with $B;`
- „<op1> ist gleich <op2>“
  Syntax: `set <op1> equal to <op2>`
  Beispiel: `print set $A is equal to $A;`

2.2.2 arithmetische Ausdrücke

Operanden können sein:

- Ganze Zahlen
- Reelle Zahlen
- Variablen
- Das Ergebnis eines arithmetischen Ausdrucks (Ausdrücke können also geschachtelt werden)

Folgende Operationen werden unterstützt:

- Addition (+)
- Subtraktion (−)
- Multiplikation (*)
- Division (/)
- Negation (−)
- Runden
  Syntax: `round(<wert>, <anzahl_nachkommastellen>)`

Folgende Mengenoperationen werden unterstützt:

- Kardinalzahl
  Syntax: `get cardinal value of <operand>`
  Beispiel: `print get cardinal value of $A;` → 2
- Betrag (existiert nur für Fuzzy-Mengen)
  Syntax: `get absolute value of <operand>`
  Beispiel: `print get absolute value of $A;` → 0.85
- relativer Betrag (existiert nur für Fuzzy-Mengen)
  Syntax: `get average absolute value of <operand>`
  Beispiel: `print get average absolute value of $A;` → 0.425

2.2.3 komplexe Ausdrücke

Es können beliebig komplexe Ausdrücke gebildet werden. Die Priorität von Teilausdrücken lässt sich mit Klammern festzulegen.

Beispiele:

```
print B or A xor B and B;                                   → true
print B or ( A xor B ) and B;                               → false
print ( get average absolute value of $A * 15 - 3 ) / 8;    → 0.421
```

2.3 Anweisungen

Ein Skript besteht aus mindestens einer Anweisung. Die letzte Anweisung muss die Anweisung ‚`exit;`' sein.

2.3.1 Ausgaben mit `print`

Ausgaben erfolgen mit dem Kommando `print`. Folgende Ausgaben sind möglich:

- Ausgabe einer Zeichenkette
  `print „Hallo Welt!";`
  → Hallo Welt!
- Ausgabe des Ergebnisses eines arithmetischen Ausdrucks
  `print 3 + 7;`
  → 10
- Ausgabe des Ergebnisses eines boolschen Ausdrucks
  `print true and false;`
  → false
- Ausgabe einer scharfen Menge
  `print { 1, 2, 5, 4711 };`
  → { 1, 2, 5, 4711 }
- Ausgabe einer Fuzzy-Menge
  `print #{[a,0.1], [b,0.75], [c,1]};`
  → #{ [a,0.1], [b,0.75], [c,1] }
- Ausgabe des Ergebnisses einer Mengenoperation
  `print create complement of #{ [a,0.1], [b,0.75], [c,1] };`
  → #{ [a,0.9], [b,0.25], [c,0] }
- Ausgabe des aktuellen Inhalts eines Bezeichners
  `print $A;`

Mehrere Ausgaben können mit ‚&' verknüpft werden.

Beispiel:

```
print $A & „ geschnitten „ & $B & „ ergibt: " & create intersection
of $A and $B;
```

Jede Ausgabe erfolgt in einer neuen Zeile.

2.3.2 Variablendefinition

Es gibt zwei Arten von Variablen:

- Wertevariablen
  Werden mit einem Bezeichner in Großbuchstaben angesprochen: `A`

- Mengenvariablen
  Werden mit einem Bezeichner in Großbuchstaben angesprochen, dem ein Dollar-Zeichen vorangestellt ist: `$A`

Variablen werden einfach durch die Zuweisung eines Wertes definiert.

Syntax: `<bezeichner> := <wert>;`

Variablen werden im internen Speicher des Programms abgelegt und können später in Ausdrücken verwendet oder ausgegeben werden.

Auch das Ergebnis einer Mengenoperation oder eines Ausdrucks kann einer Variablen zugewiesen werden. Die Mengenoperation muss allerdings in Klammern stehen.

Beispiele:

```
$CS := ( create complement of $C );

Y := 3 + 7;

Z := A xor B;
```

2.3.3 Löschen einer Variablen

Einer schon definierten Variablen kann nicht einfach ein neuer Wert zugewiesen werden. Sie muss zuvor mit dem Kommando ‚delete' wieder freigegeben werden.

Beispiel:

```
delete CS;
```

2.3.4 For-Each-Schleife

Mit der For-Each-Schleife lässt sich eine Anweisung bzw. ein Anweisungsblock für jedes Element einer Menge ausführen.

Syntax: `for each <identifier> from <set> do <statement>`

Beispiele:

```
XSUM:=0;
$A := { 1, 2, 4, 6, 5, 0, 14, 28};
print "$A:=" & $A;
for each X from $A do XSUM:=XSUM+X;
print "Die Summe über alle Elemente von $A ist " & XSUM;
```

Zusätzlich kann eine Bedingung angegeben werden. Die Anweisung wird dann nur ausgeführt, wenn die Bedingung erfüllt ist.

Syntax: `for each <identifier> from <set> where <boolean_expression> do <statement>`

Beispiel:

```
XSUM:=0;
for each X from $A where X>5 and X<20 do
begin
```

```
      XSUM:=XSUM+X;
end
print "Die Summe über alle Elemente von $A, die >5 und <20 sind, ist
" & XSUM;
```

2.3.5 If-Then-Else

Führt eine Anweisung bzw. einen Anweisungsblock nur dann aus, wenn eine Bedingung erfüllt ist.

Syntax: `if <boolean_expression> then begin <statement> end`

Beispiel:

```
A := 4712;
if not (A==4711) then
begin
      print "Das ist nicht 4711!";
end
```

Zusätzlich kann eine Anweisung angegeben werden, die ausgeführt wird, wenn die Bedingung nicht zutrifft.

Syntax:

```
if <boolean_expression> then
begin
      <statement>
end else begin
      <statement>
end
```

Beispiel:

```
if (A==4711)
then begin
      print "Das ist 4711!";
end
else begin
      print "Das ist nicht 4711!";
end
```

2.3.6 Add-Element

Diese Anweisung fügt einer bestehenden Menge ein weiteres Element hinzu.

Syntax: `add <element> as element to set <set>;`

Beispiel:

```
$A := { a, c };
print "$A vorher " & $A;                → $A vorher { a, c }
add b as element to set $A;
print "$A nachher " & $A;               → $A nachher { a, b, c }
```

2.3.7 Anweisungsblöcke

Mehrere Anweisungen können in einem Anweisungsblock zusammengefasst werden. Ein Block beginnt mit dem Schlüsselwort ‚`begin`' und endet mit dem Schlüsselwort ‚`end`' (jeweils ohne nachfolgendes Semikolon)[3].

Beispiel:

```
begin
     print "erste Anweisung im Block";
     print "zweite Anweisung im Block";
end
```

2.3.8 Exit

Mit der Anweisung ‚`exit`' wird das Programm beendet.

2.4 Mengenoperationen

Die im Folgenden aufgeführten Operationen sind sowohl auf scharfe Mengen als auch auf Fuzzy-Mengen anwendbar[4]. Als Operanden können Mengendefinitionen (wie `{ a, d, e }`), Bezeichner und Mengenoperationen (`create complement of C`) verwendet werden, wobei letztere in Klammern gesetzt werden müssen, um als Operator verwendet werden zu können. Mengenoperationen können überall verwendet werden, wo auch Mengen benutzt werden können.

Den Beispielen in diesem Abschnitt liegen die folgenden Mengendefinitionen zugrunde:

```
$A := { a, b };
$B := { b, c };
$C := #{ [a,0.1], [b,0.75] };
$D := #{ [a,0.9], [b,0.25] };
```

2.4.1 Vereinigungsmenge

Liefert die Vereinigungsmenge zweier Mengen.

Syntax: `create union of <op1> and <op2>`

Beispiel:

```
print create union of $A and $B; → { a, b, c }
print create union of $C and $D; → #{ [a,0.9], [b,0.75] }
```

2.4.2 Schnittmenge

Liefert die Schnittmenge zweier Mengen.

Syntax: `create intersection of <op1> and <op2>`

Beispiel:

```
print create intersection of $A and $B;     → { b }
print create intersection of $C and $D;     → #{ [a,0.1], [b,0.25] }
```

---

[3] Lokale Variablen werden nicht unterstützt. Eine in einem Anweisungsblock definierte Variable ist also eine globale Variable.

[4] Allerdings bin ich beim Komplement einer Fuzzy-Menge mit einer anderen Fuzzy-Menge (create complement of X using Y) und beim symmetrischen Komplement zweier Fuzzy-Mengen nicht sicher, ob die von mir verwendete Definitionen wirklich korrekt sind.

2.4.3 Kartesisches Produkt

Liefert das kartesische Produkt zweier Mengen.

Syntax: `create cartesian product of <op1> and <op2>`

Beispiel:

```
print create cartesian product of $A and $B;
```
→ { (a,b), (a,c), (b,b), (b,c) }

```
print create cartesian product of $C and $D;
```
→ #{ [(a,a),0.1], [(a,b),0.1], [(b,a),0.75], [(b,b),0.25] }

2.4.4 unäres Komplement

Liefert das Komplement einer Menge. Bei scharfen Mengen ist das Ergebnis immer die leere Menge, da die Angabe einer Grundmenge im Programm derzeit nicht möglich ist.

Syntax: `create complement of <op>`

Beispiel:

```
print create complement of $A;
```
→ { <EMPTY> }
```
print create complement of $C;
```
→ #{ [a,0.9], [b,0.25], [c,0] }

2.4.5 binäres Komplement

Liefert das Komplement einer Menge bezogen auf eine Grundmenge („X ohne Y“).

Syntax: `create complement of <op1> using <op2> as basic set`

Beispiele:

```
print create complement of $A using $B as basic set;
```
→ { c }
```
print create complement of $C using $D as basic set;
```
→ #{ [a,0], [b,0.50] }

2.4.6 symmetrisches Komplement

Liefert das symmetrische Komplement zweier Mengen.

Syntax: `create symmetric complement of <op1> and <op2>`

Beispiele:
```
print create symmetric complement of $A and $B;
```
→ { a, c }
```
print create symmetric complement of $C and $D;
```
→ #{ [a,0.8],[b,0.50] }

2.4.7 verschachtelte Mengenoperationen

Mengenoperationen lassen sich durch Verschachtelung miteinander kombinieren.

Beispiel:

```
print create union of ( create intersection of $A and $B ) and { d,
f };
```
→ { b, d, f }

2.5 Operationen auf Mengenelemente

Syntax	Beschreibung
`left(<pair_element>)`	Liefert das linke Element aus einem Paar-Element.
`right(<pair_element>)`	Liefert das rechte Element aus einem Paar-Element.
`pair_element(<left>,<right>)`	Erzeugt ein neues Paar-Element.
`fuzzy_element(<e>,<value>)`	Erzeugt ein neues Fuzzy-Element.

3 Die Programmstruktur

Das Hauptprogramm (also die main-Methode) befindet sich im File ‚FuzzySetInterpreter.jj' (`src/main/javacc`), das als Input für den Parser-Generator JavaCC dient. Im Wesentlichen besteht das Programm aus einem Interpreter für die im Package ‚parser' definierte Skriptsprache.

Der Interpreter sucht zunächst nach vollständige Anweisungen bzw. Anweisungsblöcken und „übersetzt" diese in eine Objektstruktur im Speicher, die aus Objekten besteht, die das Interface ExecutableItem bzw. ExecutableBlock implementieren. Ist die Objektstruktur vollständig aufgebaut wird sie durch Aufruf der Methode , ExecutableItem.execute()' ausgeführt.

Mit den Parametern der Anweisungen wird auf die gleiche Weise verfahren. Der Interpreter baut für alle Parameter (Ausdrücke, Variablen, ...) zunächst eine Objektstruktur auf, die aus Objekten besteht, die das Interface Evaluable implementieren. Erst wenn die Anweisungen ausgeführt werden, werden auch die Parameter ausgewertet. Dies geschieht durch Aufruf der Methode ‚Evaluate.evaluate()'.

Das Programm besteht aus den folgenden Packages, die in den folgenden Abschnitten kurz erläutert werden.

- evaluable
  - evaluable.arithmetic
  - evaluable.bool
  - evaluable.compare
  - evaluable.converter
  - evaluable.element
  - evaluable.set
- fuzzyset
- parser
- set
- statements
- unittests

Der Sourcecode und eine mit dem Tool JavaDoc generierte HTML-Dokumentation der Sourcen finden sich im beigefügten Eclipse-Projekt sowie in den Unterverzeichnissen src bzw. doc.

3.1 Das Package ‚evaluable'

Hier sind alle Klassen enthalten, die benötigt werden, um die Ausdrücke der Skriptsprache zur Laufzeit auswerten zu können. Die Klassen implementieren das Entwurfsmuster „Kommando"[5].

[5] Siehe https://de.wikipedia.org/wiki/Kommando_(Entwurfsmuster)

3.1.1 Interfaces

Interface	Funktion
Evaluable	Typisiertes Interface. Stellt die Methodensignatur ‚evaluate' bereit. Alle Klassen, die zur Laufzeit des Skripts auswertbare Sprachelemente repräsentieren implementieren dieses Interface.
UnaryOperator	Typisiertes Interface. Stellt die Methodensignatur ‚evaluate' bereit. Alle Klassen, die zur Laufzeit des Skripts auswertbare unäre Operatoren repräsentieren implementieren dieses Interface.
BinaryOperator	Typisiertes Interface. Stellt die Methodensignatur ‚evaluate' bereit. Alle Klassen, die zur Laufzeit des Skripts auswertbare binäre Operatoren repräsentieren implementieren dieses Interface.
TernaryOperator	Typisiertes Interface. Stellt die Methodensignatur ‚evaluate' bereit. Alle Klassen, die zur Laufzeit des Skripts auswertbare ternäre Operatoren repräsentieren implementieren dieses Interface.

3.1.2 Klassen

Klasse	Funktion
UnaryOperation	Implementiert unäre Operationen. Implementiert seinerseits wiederum das Entwurfsmuster „Kommando". Implementiert das Interface Evaluable und verwendet Klassen, die das Interface UnaryOperator implementieren als Operator.
BinaryOperation	Implementiert binäre Operationen. Implementiert seinerseits wiederum das Entwurfsmuster „Kommando". Implementiert das Interface Evaluable und verwendet Klassen, die das Interface BinaryOperator implementieren als Operator.
TernaryOperation	Implementiert ternäre Operationen. Implementiert seinerseits wiederum das Entwurfsmuster „Kommando". Implementiert das Interface Evaluable und verwendet Klassen, die das Interface TernaryOperator implementieren als Operator.
ResolveConstant	Liefert den Wert einer Konstanten zur Laufzeit. Implementiert das Interface Evaluable.
ResolveVariable	Evaluiert den Inhalt einer Variablen zur Laufzeit. Implementiert das Interface Evaluable.

3.2 Das Package ‚evaluable.arithmetic'

Enthält alle konkreten Kommandos, die als Ergebnis etwas liefern, dass in arithmetischen Ausdrücken verwendet werden kann.

3.2.1 Klassen

Klasse	Funktion
Addition	Repräsentiert die Addition zweier Zahlen. Implementiert das Interface BinaryOperator und kann daher in Objekten der Klasse BinaryOperation verwendet werden.
Division	Repräsentiert die Addition zweier Zahlen. Implementiert das Interface BinaryOperator und kann daher in Objekten der Klasse BinaryOperation verwendet werden.
GetAbsoluteValue	Repräsentiert die Ermittlung des Betrags einer Fuzzy-Menge. Implementiert das Interface UnaryOperator und kann daher in Objekten der Klasse UnaryOperation verwendet werden.
GetAverageValue	Repräsentiert die Ermittlung des relativen Betrags einer Fuzzy-

	Menge. Implementiert das Interface UnaryOperator und kann daher in Objekten der Klasse UnaryOperation verwendet werden.
GetCardinalValue	Repräsentiert die Ermittlung der Kardinalzahl einer Menge. Implementiert das Interface UnaryOperator und kann daher in Objekten der Klasse UnaryOperation verwendet werden.
Negate	Repräsentiert die Negation einer Zahl. Implementiert das Interface UnaryOperator und kann daher in Objekten der Klasse UnaryOperation verwendet werden.
Product	Repräsentiert die Multiplikation zweier Zahlen. Implementiert das Interface BinaryOperator und kann daher in Objekten der Klasse BinaryOperation verwendet werden.
Round	Repräsentiert das Runden einer Zahl. Als zweiter Operand wird die Anzahl der Nachkommastellen erwartet. Implementiert das Interface BinaryOperator und kann daher in Objekten der Klasse BinaryOperation verwendet werden.
Subtraction	Repräsentiert die Subtraktion zweier Zahlen. Implementiert das Interface BinaryOperator und kann daher in Objekten der Klasse BinaryOperation verwendet werden.

3.3 Das Package ‚evaluable.bool'

Enthält alle konkreten Kommandos, die als Ergebnis etwas liefern, dass in booleschen Ausdrücken verwendet werden kann.

3.3.1 Klassen

Klasse	**Funktion**
And	Repräsentiert die Konjunktion zweier boolescher Ausdrücke. Implementiert das Interface BinaryOperator und kann daher in Objekten der Klasse BinaryOperation verwendet werden.
IsBinaryComplement	Prüft, ob eine Menge das binäre Komplement zweier anderer Mengen ist. Implementiert das Interface TernaryOperator und kann daher in Objekten der Klasse TernaryOperation verwendet werden.
IsEmpty	Prüft, ob eine Menge leer ist. Implementiert das Interface UnaryOperator und kann daher in Objekten der Klasse UnaryOperation verwendet werden.
IsEqual	Prüft, ob zwei Mengen äquivalent sind. Implementiert das Interface BinaryOperator und kann daher in Objekten der Klasse BinaryOperation verwendet werden.
IsIntersection	Prüft, ob eine Menge die Schnittmenge zweier anderer Mengen ist. Implementiert das Interface TernaryOperator und kann daher in Objekten der Klasse TernaryOperation verwendet werden.
IsSubset	Prüft, ob eine Menge die Teilmenge einer andere Menge ist. Implementiert das Interface BinaryOperator und kann daher in Objekten der Klasse BinaryOperation verwendet werden.
IsSymmetricComplement	Prüft, ob eine Menge das symmetrische Komplement zweier anderer Mengen ist. Implementiert das Interface TernaryOperator und kann daher in Objekten der Klasse TernaryOperation verwendet werden.
IsUnaryComplement	Prüft, ob eine Menge das Komplement einer andere Menge ist. Implementiert das Interface BinaryOperator und kann daher in

	Objekten der Klasse BinaryOperation verwendet werden.
IsUnion	Prüft, ob eine Menge die Vereinigungsmenge zweier anderer Mengen ist. Implementiert das Interface TernaryOperator und kann daher in Objekten der Klasse TernaryOperation verwendet werden.
Not	Repräsentiert die Negation eines booleschen Ausdrucks. Implementiert das Interface UnaryOperator und kann daher in Objekten der Klasse UnaryOperation verwendet werden.
Or	Repräsentiert die Disjunktion zweier boolescher Ausdrücke. Implementiert das Interface BinaryOperator und kann daher in Objekten der Klasse BinaryOperation verwendet werden.
Xor	Repräsentiert die Antivalenz zweier boolescher Ausdrücke. Implementiert das Interface BinaryOperator und kann daher in Objekten der Klasse BinaryOperation verwendet werden.

3.4 Das Package ‚evaluable.compare'

Enthält alle konkreten Kommandos, die zwei arithmetische Ausdrücke miteinander vergleichen und als Ergebnis etwas liefern, dass in booleschen Ausdrücken verwendet werden kann.

Alle Klassen implementieren das Interface BinaryOperator und können daher in Objekten der Klasse BinaryOperation verwendet werden.

3.4.1 Klassen

Klasse	**Funktion**
Equal	„ist gleich" (==).
GreaterThan	„ist größer als" (>).
GreaterThanOrEqual	„ist größer als oder gleich" (>=).
LessThan	„ist kleiner als" (<).
LessThanOrEqual	„ist kleiner als oder gleich" (<=).

3.5 Das Package ‚evaluable.converter'

Enthält alle konkreten Kommandos, die eine Typkonvertierung durchführen.

Alle Klassen implementieren das Interface UnaryOperator und können daher in Objekten der Klasse UnaryOperation verwendet werden.

3.5.1 Klassen

Klasse	**Funktion**
ConvertSetElement2String	Konvertiert das Element einer Menge in eine Zeichenkette.
ConvertSet2String	Konvertiert eine Menge in eine Zeichenkette.

3.6 Das Package ‚evaluable.element'

Enthält alle konkreten Kommandos, die als Ergebnis ein Mengenelement liefern.

3.6.1 Klassen

Klasse	**Funktion**
CreateFuzzyElement	Erzeugt ein neues Fuzzy-Element. Implementiert das Interface BinaryOperator und kann daher in Objekten der Klasse BinaryOperation verwendet werden.
CreatePairElement	Erzeugt ein neues Element-Paar.

	Implementiert das Interface BinaryOperator und kann daher in Objekten der Klasse BinaryOperation verwendet werden.
GetLeftPairElement	Liefert das linke Element eines Element-Paars. Implementiert das Interface UnaryOperator und kann daher in Objekten der Klasse UnaryOperation verwendet werden.
GetRightPairElement	Liefert das rechte Element eines Element-Paars. Implementiert das Interface UnaryOperator und kann daher in Objekten der Klasse UnaryOperation verwendet werden.

3.7 Das Package ‚evaluable.set'

Enthält alle konkreten Kommandos, die als Ergebnis eine Menge liefern.

Alle Klassen außer UnaryComplement implementieren das Interface BinaryOperator und können daher in Objekten der Klasse BinaryOperation verwendet werden. UnaryComplement implementiert UnaryOperator und kann daher in Objekten der Klasse UnaryOperation verwendet werden.

3.7.1 Klassen

Klasse	Funktion
AddElement	Erzeugt eine Menge, der ein weiteres Element hinzugefügt wurde.
BinaryComplement	Erzeugt das Komplement einer Menge gegen eine Basismenge.
CartesianProduct	Erzeugt das kartesische Produkt zweier Mengen.
Intersection	Erzeugt die Schnittmenge zweier Mengen.
SymmetricComplement	Erzeugt das symmetrische Komplement zweier Mengen.
UnaryComplement	Erzeugt das Komplement einer Menge.
Union	Erzeugt die Vereinigungsmenge zweier Mengen.

3.8 Das Package ‚fuzzyset'

Die für den Umgang mit Fuzzy-Mengen nötige Logik ist in diesem Package implementiert. Das Package setzt auf dem Package ‚set' auf.

3.8.1 Klassen

Klasse	Funktion
FuzzyElement	Ein Element einer Fuzzy-Menge. Beinhaltet einen µ-Wert[6] und ein Element (also etwas, dass das Interface ‚Set' implementiert). Ist von ‚SetElement' abgeleitet und kann daher auch in scharfen Mengen als Element genutzt werden[7].
FuzzySet	Stellt Methoden für die innerhalb des Programms auf Fuzzy-Mengen anwendbaren Operationen bereit. Die Elemente der Menge werden in einer sortierten Liste verwaltet[8]. Die Klasse implementiert das Interface ‚Set', sodass alle dort definierten Mengenoperationen unterstützt werden.
FuzzySetIterator	Stellt einen Iterator für FuzzyElement-Objekte bereit. Sorgt also u.a. dafür, dass Objekte der Klasse ‚FuzzySet' in For-Each-Schleifen verwendet werden können.

[6] Weil Berechnungen mit den Datentypen ‚float' und ‚double' in Java sehr schnell ungenau werden hat der µ-Wert" den Java-Standard-Datentypen ‚BigDecimal'.

[7] Dies wird vom Parser derzeit aber nicht unterstützt. Zumindest erbt ‚FuzzyElement' aber die in ‚SetElement' implementierte Methode ‚compareTo'.

[8] Dadurch werden die Elemente der Menge nicht unbedingt in der Reihenfolge ausgegeben, in der sie definiert wurden.

3.8.2 Interfaces

Interface	Funktion
MappingFunction	Stellt die Methodensignatur ‚compute' bereit. Klassen, die dieses Interface implementieren müssen in der Lage sein, zu einem übergebenen Element einen µ-Werten zu berechnen. Wird verwendet, um Fuzzy-Mengen mit berechneten µ-Werten definieren zu können. Derzeit wird diese Funktionalität jedoch nicht benutzt[9].

3.9 Das Package ‚parser'

Dieses Package enthält den Code des Parsers, der die Eingaben bzw. die Zeilen des Eingabefiles interpretiert. Dieser Parser wird mit dem Tool JavaCC[10] auf Basis der im File , FuzzySetInterpreter.jj' definierten Grammatik generiert. Lediglich die Files , FuzzySetInterpreter.jj' und ‚SemanticException.java' werden nicht generiert.

Die Klasse FuzzySetInterpreter enthält die main-Methode, also das Hauptprogramm und die globalen Speicher für Variablen. Außerdem liest sie ein evtl. übergebenes Skriptfile ein bzw. erwartet die Benutzer-Eingaben.

3.10 Das Package ‚set'

Das Package enthält den Code, der für die Definition von scharfen Mengen und für Operationen auf diese Mengen nötig ist.

3.10.1 Interfaces

Interface	Funktion
Set	Definiert die Signaturen aller Methoden, die eine Mengen-Klasse innerhalb des Programms unterstützen muss. Bspw. für die Bildung von Vereinigungsmengen oder des Komplements einer Menge.

3.10.2 Klassen

Klasse	Funktion
BooleanElement	Nimmt boolesche Werte auf und ist von ‚SetElement' abgeleitet.
StringElement	Nimmt Zeichenketten auf und ist von ‚SetElement' abgeleitet.
NaturalNumberElement	Nimmt ganze Zahlen auf und ist von ‚SetElement' abgeleitet.
DecimalElement	Nimmt Gleitkommazahlen auf und ist von ‚SetElement' abgeleitet.
PairElement	Nimmt Element-Paare auf und ist von ‚SetElement' abgeleitet.
SetElement	Abstrakte Basisklasse aller Mengenelemente.
SharpSet	Stellt Methoden für die innerhalb des Programms auf scharfe Mengen anwendbaren Operationen bereit. Die Klasse implementiert das Interface ‚Set', sodass alle dort definierten Mengenoperationen unterstützt werden. Wie in ‚FuzzySet' werden die Elemente auch hier in einer ungeordneten Liste gespeichert.

3.11 Das Package ‚statements'

Das Package enthält alle Klassen, die das Interface ExecutableItem implementieren und die ausführbare Anweisungen repräsentieren. Auch die Klassen in diesem Package implementieren das Kommando-Entwurfsmuster.

---

[9] Die Klasse FuzzySet enthält aber bereits einen entsprechenden Konstruktor.

[10] „Java Compiler Compiler": Ein Werkzeug zum Generieren von Parsern (ähnlich den Unix-Tools lex und yacc).

3.11.1 Interfaces

Interface	Funktion
ExecutableItem	Stellt die Methodensignatur ‚executable()' bereit. Wird von allen Klassen implementiert, die ausführbare Anweisungen repräsentieren.

3.11.2 Klassen

Klasse	Funktion
AddElementStatement	Repräsentiert die Add-Element-Anweisung.
ExecutableBlock	Repräsentiert einen mit den Schlüsselworten ‚begin' und ‚end' eingeschlossenen Anweisungsblock. Letztlich handelt es sich um eine Liste von ExecutableItem-Objekten.
ForEachLoop	Repräsentiert die For-Each-Schleife.
IfThenElseStatement	Repräsentiert die If-Then-Else-Anweisung.
Print	Repräsentiert die Print-Anweisung.
VariableDefinition	Repräsentiert die Definition einer Variablen. Fügt dem Speicher also eine neue Variable hinzu.
VariableDelete	Repräsentiert das Löschen einer zuvor definierten Variablen. Entfernt eine Variable also aus dem Speicher.