Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung
Informatik I
Wintersemester 2006

Kapitel 11: Bildung von Abstraktionsbarrieren mit abstrakten Typen und Modulen

Vorlesung: Prof. Dr. Christian Böhm
Übungen: Elke Achtert, Arthur Zimek

http://www.dbs.ifi.lmu.de/Lehre/Info1

Database Systems

Inhalt

1. Vorzüge des Verbergens
2. Abstrakte Typen in SML
3. Beispiel : Abstrakte Typen zur Implementierung der Datenstruktur „Menge“
4. Module in SML
5. Hinweis auf die Standardbibliothek von SML

Überblick

1. Vorzüge des Verbergens
2. Abstrakte Typen in SML
3. Beispiel : Abstrakte Typen zur Implementierung der Datenstruktur „Menge“
4. Module in SML
5. Hinweis auf die Standardbibliothek von SML

Strukturierung

- Damit Programme für die menschliche Leserschaft übersichtlich und verständlich bleiben, sollte man (große) Programme strukturieren.
- Programmiersprachen bieten zwei komplementäre Mittel zur Strukturierung von Programmen:
  - die Bildung von Abstraktionsbarrieren zum Verbergen von Definitionen
  - die Gruppierung von Teilprogrammen
- In SML können Abstraktionsbarrieren durch abstrakte Datentypen und Module geschaffen werden.
- SML bietet Module („Strukturen“), zur Gruppierung von Teilprogrammen.

Verborgene Definitionen

- Es gibt komplexe Programme mit kleinen und übersichtlichen Kernen - meistens erfordern praktische Anwendungen aber die Erweiterung der Programmkerne.
- Wenn eine Aufteilung in Teilprogramme nicht ausreicht, um Programme übersichtlich zu machen, dann ist es sinnvoll Definitionen (von Prozeduren oder Typen z.B.) zu verwenden, die außerhalb dieses Teilprogramms nicht sichtbar und nicht verwendbar sind.
- Dieser Ansatz liegt auch den lokalen Deklarationen einer Prozedur zugrunde (vgl. Kapitel 4).

Vor- und Nachteil

- Nachträgliche Änderungen werden leichter, da diese sich nur innerhalb des Teilprogramms auswirken und nicht im größeren Gesamtprogramm.
- Teilprogramme müssen aber häufig mehrere Prozeduren und Typen umfassen, so dass lokale Deklarationen von Prozeduren unzureichend sind.

Abstrakter Typ

- Ein *abstrakter Typ* ergänzt eine Typdefinition (in SML: `datatype` – Deklaration) um Namen von Prozeduren und anderen Werten, die abgerufen werden können, deren Implementierung aber außerhalb der Definition des abstrakten Typs unsichtbar ist.
- So können zusammen mit einem Typ grundlegende Namen und Operationen zur Verwendung des Typs zur Verfügung gestellt werden, ohne die interne Repräsentation von Werten dieses Typs zugänglich machen zu müssen.

Modul

- Ein *Modul* ist ein Teilprogramm, das lokale Definitionen beinhalten kann.
- Mit einem Modul wird festgelegt:
  - Die Definitionen des Moduls, die außerhalb des Moduls verwendet werden können
  - Die Definitionen des Moduls, die nur innerhalb des Moduls verwendet werden können.

Information hiding

- Man kann bei abstrakten Typen und Modulen sicherstellen, dass nur die zur Verfügung gestellten Operationen verwendet werden können.
- So kann die Datenverarbeitung auf die Verwendung dieser Operationen eingeschränkt werden.
- Unter „*information hiding*“ versteht man das Prinzip, nur die für die Verarbeitung notwendigen Namen und Operationen zur Verfügung zu stellen, aber die interne Repräsentation von Werten zu verbergen.
- Information hiding ist eines der wichtigsten Hilfsmittel bei der Softwareentwicklung.

Überblick

1. Vorzüge des Verbergens
2. Abstrakte Typen in SML
3. Beispiel : Abstrakte Typen zur Implementierung der Datenstruktur „Menge“
4. Module in SML
5. Hinweis auf die Standardbibliothek von SML

Motivationsbeispiel: Das Farbmodell von HTML

- Die Markup-Sprache (oder Auszeichnungssprache) HTML sowie die Style-Sheet-Sprache (oder Formatierungssprache) CSS1 repräsentieren Farben nach einem einfachen Modell.
- In diesem Modell ist eine Farbe durch Anteile an den drei Grundfarben Rot, Grün und Blau (daher die Bezeichnung RGB) definiert, wobei diese Anteile natürliche Zahlen zwischen einschließlich 0 und 255 sind:
  - Das RGB-Tripel mit den kleinstmöglichen Komponenten (0, 0, 0) entspricht der Farbe Schwarz
  - Das RGB-Tripel mit den größtmöglichen Komponenten (255, 255, 255) entspricht der Farbe Weiß.
- In HTML 4.01 sind 16 Farbennamen (vor-) definiert.

Ein SML – Typ zur Definition des Farbmodells von HTML

- Unter Verwendung einer `datatype` - Deklaration (vgl. Kapitel 8.3) kann in SML ein Typ `color` definiert werden:

```
- datatype color = rgb of int * int * int;
datatype color = rgb of int * int * int
```

Deklaration von HTML - Farbnamen

```
- val Black = rgb( 0, 0, 0);
val Black = rgb (0,0,0) : color
- val Gray = rgb(128, 128, 128);
val Gray = rgb (128,128,128) : color
- val Red = rgb(255, 0, 0);
val Red = rgb (255,0,0) : color
- val Green = rgb( 0, 128, 0);
val Green = rgb (0,128,0) : color
- val Lime = rgb( 0, 255, 0);
val Lime = rgb (0,255,0) : color
- val Blue = rgb( 0, 0, 255);
val Blue = rgb (0,0,255) : color
- val White = rgb(255, 255, 255);
val White = rgb (255,255,255) : color
```

Die grüne Farbe mit *maximalem* Grünanteil heißt `Lime`.
`Green` bezeichnet die grüne Farbe mit *halbem* Grünanteil.

Definition einer Funktion zum Mischen zweier Farben

```
- exception negative_part;
- fun mix(color1, part1, color2, part2) =
      if part1 < 0 orelse part2 < 0
      then raise negative_part
      else  let val parts = part1 + part2;
                val rgb(r1, g1, b1) = color1;
                val rgb(r2, g2, b2) = color2;
                val r = (part1*r1 + part2*r2) div parts;
                val g = (part1*g1 + part2*g2) div parts;
                val b = (part1*b1 + part2*b2) div parts
            in
                rgb(r, g, b)
            end;
val mix = fn : color * int * color * int -> color
```

Beispiel – Anwendungen

```
- mix(Red, 1, mix(Lime, 1, Blue, 1), 2);
val it = rgb (85,84,84) : color

- mix(rgb(1,1,1), 1, rgb(128,128,128), 1);
val it = rgb (64,64,64) : color

- mix(rgb(1,1,1), 1, White, 1);
val it = rgb (128,128,128) : color
```


Sichtbarkeit

- Werden Farben in SML durch die vorangehende `datatype` - Deklaration definiert, so sind der Typkonstruktor `rgb` und die Tripel-Gestalt einer Farbe sichtbar.
- Die Funktion `mix` kann z.B. nicht nur auf Farbennamen wie im Ausdruck

  ```
  mix(Red, 1, mix(Lime, 1, Blue, 1), 2)
  ```

  angewendet werden, sondern auch auf `rgb` - Tripel wie im Ausdruck

  ```
  mix(rgb(1,1,1), 1, rgb(128,128,128), 1).
  ```

Kritik

- Unsere Definition des Farbmodells verhindert nicht, dass ungültige `rgb` – Tripel mit negativen Komponenten oder mit Komponenten größer als 255 vom Programmierer verwendet werden:

```
- val a = rgb(~1,0,999);
val a = rgb (~1,0,999) : color
- mix(rgb(~12,~12,1200),5, rgb(~20,~20,2000),3);
val it = rgb (~15,~15,1500) : color
```

- Die Deklarationen von Farbennamen wie `Black`, `Red`, `Blue`, `Lime` und der Funktion `mix` stellen außerdem keinen abgeschlossenen Teil im Programm dar.
- Es fehlt eine syntaktische Einheit, die alle Deklarationen zusammenfasst, die mit RBG-Farben zu tun haben.

Ein abstrakter Typ zur Definition der Farben von HTML (1)

```
abstype color = rgb of int * int * int
with
  val Black = rgb( 0, 0, 0);
  val Silver = rgb(192, 192, 192);
  val Gray = rgb(128, 128, 128);
  val White = rgb(255, 255, 255);
  val Maroon = rgb(128, 0, 0);
  val Red = rgb(255, 0, 0);
  val Purple = rgb(128, 0, 128);
  val Fuchsia = rgb(255, 0, 255);
  val Green = rgb( 0, 128, 0);
  val Lime = rgb( 0, 255, 0);
  val Olive = rgb(128, 128, 0);
  val Yellow = rgb(255, 255, 0);
  val Navy = rgb( 0, 0, 128);
  val Blue = rgb( 0, 0, 255);
  val Teal = rgb( 0, 128, 128);
  val Aqua = rgb( 0, 255, 255);
```

Die 16 vordefinierten Farbnamen von HTML.

Ein abstrakter Typ zur Definition der Farben von HTML (2)

```
  exception negative_part;
  fun mix(color1, part1, color2, part2) =
      if part1 < 0 orelse part2 < 0
      then raise negative_part
      else   let val parts = part1 + part2;
                 val rgb(r1, g1, b1) = color1;
                 val rgb(r2, g2, b2) = color2;
                 val r = (part1*r1 + part2*r2) div parts;
                 val g = (part1*g1 + part2*g2) div parts;
                 val b = (part1*b1 + part2*b2) div parts
             in
                 rgb(r, g, b)
             end;
  fun display(rgb(r,g,b)) = print(Int.toString r ^ " " ^
                                  Int.toString g ^ " " ^
                                  Int.toString b ^ "\n");
end;
```


Abgeschlossenes Teilprogramm

- Diese `abstype` - Deklaration bildet eine syntaktische Einheit, in der folgende Deklarationen eingeschlossen sind:
  - Die Deklaration der Farbennamen
  - Die Deklaration der Ausnahme `negative_part`
  - Die Deklarationen der Funktionen `mix` zur Farbmischung und `display` zum Drucken eines Farbwerts
- So wird ein abgeschlossenes Teilprogramm gebildet, das alle Deklarationen umfasst, die zu dem neuen Typ `color` gehören und zu dessen Definition beitragen.

SML Mitteilung (1)

- Die Mitteilung des SML-Systems als Reaktion auf die Deklaration von `color` ist:

```
type color
val Black = - : color
val Silver = - : color
val Gray = - : color
val White = - : color
val Maroon = - : color
val Red = - : color
val Purple = - : color
val Fuchsia = - : color
val Green = - : color
val Lime = - : color
val Olive = - : color
val Yellow = - : color
```


SML Mitteilung (2)

```
val Navy = - : color
val Blue = - : color
val Teal = - : color
val Aqua = - : color
exception negative_part
val mix = fn : color * int * color * int -> color
val display = fn : color -> unit
```

- Die innere Gestalt einer Farbe (die (Wert -)Konstruktoren des neuen Typs `color`) wird in dieser Mitteilung nirgends erwähnt!

Verwendbarkeit (1)

- Die innere Gestalt der Werte des Typs ist außerhalb der `abstype`- Deklaration nicht mehr verwendbar.
- Beispiele:

```
- val mein_lieblingsblau = rgb(0, 85, 255);
Error: unbound variable or constructor: rgb

- mix(rgb(1, 1, 1), 1, rgb(128, 128, 128), 1);
Error: unbound variable or constructor: rgb
Error: unbound variable or constructor: rgb
```


Verwendbarkeit (2)

- Die *Namen* der Farben und Funktionen der `abstype` - Deklaration sind außerhalb der `abstype` - Deklaration verwendbar (sonst wäre ihre Deklaration auch nutzlos).
- Die innere Gestalt ihrer *Werte* bleibt verborgen:

```
- Red;
val it = - : color
- mix(Red, 1, Red, 1);
val it = - : color
- mix(Red, 1, mix(Lime, 1, Blue, 1), 2);
val it = - : color
- display(mix(Red, 1, mix(Lime, 1, Blue, 1), 2));
85 84 84
val it = () : unit
```

Vor- und Nachteil

- Ein Programm, das den Typ `color` verwendet, kann:
  - auf die dort definierten Farben über die zur Verfügung gestellten Namen (z.B. `Red` oder `Lime`) zugreifen und
  - neue Farben mit der zur Verfügung gestellten Funktion `mix` erzeugen.
- Es kann aber keine Farben auf andere Weise bilden.
- Insbesondere kann es nicht den Konstruktor `rgb` benutzen, um beliebige Zahlentripel zur Darstellung von Farben zu verwenden.

→ Ungültige `rgb` - Tripel mit negativen Komponenten oder mit Komponenten größer als 255 können weder vom Programmierer verwendet, noch erzeugt werden (dank der sorgfältigen Definition der Funktion `mix`).

Implementierungstyp und Schnittstelle eines abstrakten Typs

- Die Definition eines abstrakten Typs besteht aus einer Typdefinition und Wertdefinitionen, die auch Funktionsdefinitionen sein können.
- Erinnerung:
  In SML und vielen anderen funktionalen Programmiersprachen sind Funktionsdefinitionen Wertdefinitionen (vgl. Kapitel 2.4 und 7.1).
- Der Typ, der in einer `abstype` - Deklaration definiert wird, wird *Implementierungstyp* des abstrakten Typs genannt.
- Die Deklarationen, einschließlich der Deklarationen von Funktionen, die in einer `abstype` - Deklaration vorkommen, bilden die sogenannte *Schnittstelle* des abstrakten Typs.
- Ein Programm, das einen abstrakten Typ verwendet, d.h., sich darauf bezieht, wird manchmal „*Klient*“ dieses abstrakten Typs genannt.

Vorteile des Verbergens mit abstrakten Typen

- Wird ein abstrakter Typ in einem Programm verwendet, so verlangt eine Veränderung seiner Definition keine Veränderung seiner Klienten, so lange die Schnittstelle des abstrakten Typs unverändert bleibt.

→ Die Klienten sind unabhängig von der Implementierung eines Typs.

Beispiel (1)

- Das Farbmodell von HTML soll verfeinert werden, so dass für die drei Anteile der Grundfarben 1024 Abstufungen erlaubt sind.
- In diesem verfeinerten Modell können die Komponenten der Tripel Werte von 0 bis 1023 annehmen.
- Jedes Tripel des ursprünglichen Modells muss komponentenweise mit 4 multipliziert werden, um das Tripel für dieselbe Farbe im verfeinerten Modell zu erhalten.

Beispiel (2)

- Wird das HTML-Farbmodell mit dem abstrakten Typ `color` implementiert, so reicht diese Vervierfachung der Zahlen in den Deklarationen der (benannten) Farben `Black`, `Silver`, usw., um den Klienten ohne weitere Änderungen auf das verfeinerte Farbmodell umzustellen.
- Könnte der Klient dagegen direkt die interne Tripel-Gestalt der Farben verwenden, müsste für eine solche Umstellung der gesamte (möglicherweise sehr große) Klient daraufhin durchsucht werden, wo ein Zahlentripel erzeugt wird, das zur Darstellung einer Farbe dient und deshalb vervierfacht werden muss.

Vorteil abstrakter Typen

- Die Verwendung von abstrakten Typen erleichtert wesentlich die stufenweise Entwicklung und die Wartung von komplexen Programmen, weil sie lokale Veränderungen ermöglichen.

Einkapselung zur sicheren Wertverarbeitung

- Zudem können Werte eines abstrakten Datentyps nur unter Verwendung der zu diesem Zweck in der Definition des abstrakten Typs zur Verfügung gestellten Namen und Funktionen erzeugt, verändert und im Allgemeinen manipuliert werden.
- Diese *Einkapselung* verhindert eine Verarbeitung der Werte eines abstrakten Typs, die bei der Implementierung des Typs nicht beabsichtigt wurde.
- In der Praxis entstehen häufig Fehler, wenn Werte eines Typs in einer Weise verarbeitet werden, die bei der Implementierung des Typs nicht beabsichtigt wurde.
- Die Einkapselungstechnik stellt einen wesentlichen Beitrag zur Erstellung sicherer Software dar.

Überblick

1. Vorzüge des Verbergens
2. Abstrakte Typen in SML
3. Beispiel : Abstrakte Typen zur Implementierung der Datenstruktur „Menge“
4. Module in SML
5. Hinweis auf die Standardbibliothek von SML

Beispiel: Menge (1)

- Im Folgenden werden endliche Mengen von ganzen Zahlen, die extensional definiert sind (vgl. Kapitel 8.6) betrachtet.
- Der Übersichtlichkeit halber werden nur die folgenden Namen und Mengenoperationen definiert:
  - der Name `empty_set` zur Definition der leeren Menge;
  - das Prädikat `is_empty` zur Überprüfung, ob eine Menge leer ist;
  - das Prädikat `is_element` zur Überprüfung der Elementbeziehung;
  - die Funktion `insert` zum Einfügen eines Elements in eine Menge;
  - die Funktion `remove` zum Entfernen eines Elements aus einer Menge;
  - die Funktion `union` zur Vereinigung von zwei Mengen;
  - die Funktion `display` zum Drucken der Elemente einer Menge.

Beispiel: Menge (2)

- Zwei unterschiedliche abstrakte Typen mit derselben Schnittstelle werden implementiert:
  - Der erste abstrakte Typ stellt Mengen als unsortierte Listen ohne Wiederholungen von Elementen dar.
  - Der zweite abstrakte Typ verfeinert diese erste Darstellung mit der Berücksichtigung der kleinsten und größten Mengenelemente zur Beschleunigung der Funktionen `is_element`, `insert` und `remove`.

Einschub

- Ein realistisches Szenario ist, dass im Rahmen eines größeren Programmierprojekts zunächst die naheliegende, einfache erste Implementierung realisiert wird.
- Bei der Benutzung des Programms stellt sich heraus, dass es für die Anwendung zu langsam ist.
- Eine Analyse ergibt, dass die Elemente meistens in aufsteigender oder absteigender Reihenfolge sortiert in die Mengen eingefügt werden.
- Dadurch entsteht die Idee für die zweite Implementierung.
- Wenn die Mengen als abstrakte Typen realisiert sind, ist die Umstellung auf die zweite Implementierung möglich, ohne den Rest des Programms zu verändern.

Erster abstrakter Typ zur Implementierung der Menge als Liste

- Die Mengen von ganzen Zahlen werden als unsortierte Listen ohne Wiederholung von Elementen dargestellt.

emty_set, is_emty, is_element, insert

```
- abstype set = Set of int list
    with
      val empty_set = Set nil;
      fun is_empty x = (x = Set nil);
      fun is_element(_, Set nil) = false
      |   is_element(e, Set(h::t)) = (e = h)
          orelse is_element(e, Set t);
      fun insert(e, Set nil) = Set(e::nil)
      |   insert(e, Set(h::t)) = if e = h
            then Set(h::t)
            else
               let val Set L = insert(e, Set t)
               in
                  Set(h::L)
               end;
```


remove, union

```
      fun remove(e, Set nil) = Set nil
      |   remove(e, Set(h::t)) =
            if e = h
            then Set t
            else  let val Set L = remove(e, Set t)
                  in
                     Set(h::L)
                  end;
      fun union(Set nil, M) = M
      |   union(Set(h::t), Set L) =
            if is_element(h, Set L)
            then union(Set t, Set L)
            else  let val Set L1 = union(Set t,Set L)
                  in
                     Set(h::L1)
                  end;
```

display

```
      fun display(Set nil) = print "\n"
      |   display(Set(h::t)) =
          (print(Int.toString (h) ^ " ");display(Set t));

  end; (*end von with*)
```


Wert der Definition in SML

```
type set
val empty_set = - : set
val is_empty = fn : set -> bool
val is_element = fn : int * set -> bool
val insert = fn : int * set -> set
val remove = fn : int * set -> set
val union = fn : set * set -> set
val display = fn : set -> unit
```

Anwendungen (1)

```
- is_element(1, insert(1, empty_set));
val it = true : bool

- is_empty(remove(1, insert(1, empty_set)));
val it = true : bool

- insert(3, insert(2, insert(1, empty_set)));
val it = - : set

- display(insert(3, insert(2, insert(1, empty_set))));
1 2 3
val it = () : unit

- display(remove(2, insert(3, insert(2, insert(1,
  empty_set)))));
1 3
val it = () : unit
```


Anwendungen (2)

```
- is_element(1, insert(1, empty_set));
val it = true : bool

- val set1 = insert(3, insert(2, insert(1,
  empty_set)));
val set1 = - : set

- val set2 = insert(5, insert(4, insert(3,
  empty_set)));
val set2 = - : set

- display(union(set1, set2));
1 2 3 4 5
val it = () : unit
```

Zweiter abstrakter Typ zur Implementierung der Menge als Liste

- Eine Menge von ganzen Zahlen wird nun als Tripel `(k,g,L)` dargestellt, wobei:
  - k das kleinste Element der Menge ist,
  - g das größte Element der Menge ist,
  - L die unsortierte Liste (ohne Wiederholung) der Mengenelemente ist.

empty_set, is_emty, is_element

```
- abstype set = Set of int * int * int list | EmptySet
    with
      val empty_set = EmptySet;
      fun is_empty x = (x = EmptySet);
      fun is_element(_, EmptySet) = false
      |   is_element(e, Set(k, g, h::t)) =
            let
               fun member(_, nil) = false
               |   member(m, h::t) =
                     m = h orelse member(m,t);
            in
               k <= e andalso e <= g andalso member(e, h::t)
            end;
```

insert

```
fun insert(e, EmptySet) = Set(e, e, e::nil)
|   insert(e, Set(k, g, L)) =
      if e < k
      then Set(e, g, e::L)
      else  if e > g
            then Set(k, e, e::L)
            else
               let
                  fun add(e, nil) = e::nil
                  | add(e, h::t) =
                        if e = h
                        then h::t
                        else h::add(e, t)
               in
                  Set(k, g, add(e, L))
               end;
```


remove

```
fun remove(e, EmptySet) = EmptySet
 |  remove(e, Set(k, g, L)) = if e < k orelse e > g
      then Set(k, g, L)
      else   let fun remove(e, nil) = nil
                   |  remove(e, h::t) = if e = h
                         then t
                         else h::remove(e,t);
                 fun min(h::nil) = h
                  |  min(h::t) = Int.min(h, min t);
                 fun max(h::nil) = h
                  |  max(h::t) = Int.max(h, max t);
                 val L´ = remove(e, L)
             in
                if L´ = nil then EmptySet
                else Set(min L´, max L´ , L´ )
             end;
```

union

```
fun union(s, EmptySet) = s
  | union(EmptySet, s) = s
  | union(Set(k1, g1, L1), Set(k2, g2, L2)) =
      let fun member(_, nil) = false
            | member(m, h::t) =
              m = h orelse member(m,t);
          fun merge(nil, L) = L
            | merge(h::t, L) =
                if member(h,L)
                then merge(t,L)
                else h::merge(t,L)
      in
          Set(Int.min(k1,k2),
              Int.max(g1,g2),
              merge(L1,L2))
      end;
```


display

```
fun display(EmptySet) = print "\n"
  | display(Set(_, _, h::t)) =
      let
            fun display_list(nil) = print "\n"
              | display_list(h::t) =
                  (
                  print(Int.toString(h) ^ " ");
                  display_list(t)
                  );
      in
            display_list(h::t)
      end
end; (*end von with*)
```

Wert der Definition in SML

```
type set
val is_empty = fn : set -> bool
val is_element = fn : int * set -> bool
val insert = fn : int * set -> set
val remove = fn : int * set -> set
val union = fn : set * set -> set
val display = fn : set -> unit
```


Anwendungen (1)

```
- is_element(1, insert(1, empty_set));
val it = true : bool

- is_empty(remove(1, insert(1, empty_set)));
val it = true : bool

- insert(3, insert(2, insert(1, empty_set)));
val it = - : set
```

Anwendungen (2)

```
- display(insert(3, insert(2, insert(1, empty_set))));
3 2 1
val it = () : unit
- display(remove(2,insert(3,insert(2,insert(1, empty_set)))));
3 1
val it = () : unit
- is_element(1, insert(1, empty_set));
val it = true : bool
- val set1 = insert(3,insert(2,insert(1,empty_set)));
val set1 = - : set
- val set2 = insert(5,insert(4,insert(3,empty_set)));
val set2 = - : set
- display(union(set1,set2));
2 1 5 4 3
val it = () : unit
```


Vergleich

- Die Auswertung der gleichen Ausdrücke wie beim ersten Typ ergibt genau die gleichen Werte.
- Der einzige Unterschied ist die Reihenfolge der Mengenelemente in der Liste, die die Menge darstellt.
- Dieser Unterschied kommt daher, dass die Operationen `insert` und `remove` zum Einfügen bzw. Entfernen von Elementen in unterschiedlicher Weise implementiert sind.
- Dieser Unterschied ist aber nur mit Hilfe der Druckprozedur sichtbar und hat keine Auswirkungen auf die Ergebnisse der Mengenoperationen.