

## 15. SML-Modul-Konzept

Bisher: „Programmieren im Kleinen“

Programmieren einer einzelnen Datenstruktur oder eines einzelnen Algorithmus

Jetzt: „Programmieren im Großen“

### Ziele:

- Separation of concerns: Dekomposition von Systemen in Komponenten mit verschiedenen Aufgaben
- Wiederverwendbarkeit (reuse)
- Wartbarkeit

### Methoden und Konzepte:

- Modularisierung
- Information Hiding, Datenabstraktion, Datenkapselung
- Trennung von Spezifikation und Implementierung
- Parametrisierung (Generizität)

### Vorgehensweisen:

- **top down**: Spezifikation gegeben, Implementierung gesucht  
SML:      Signatur:      Spezifikation  
              Struktur:      Implementierung
- **bottom up**: Implementierung vorhanden, spezielle Benutzung  
SML:      Struktur:      Implementierung  
              Signatur:      Spezifikation

<u>Struktur</u>	<u>Signatur</u>
ist Implementierung <u>besteht aus</u> : Deklarationen von <ul style="list-style-type: none"> <li>• Typkonstruktoren</li> <li>• Ausnahmekonstruktoren</li> <li>• Wertbindung</li> </ul>	ist Spezifikation <u>besteht aus</u> : Spezifikation von: <ul style="list-style-type: none"> <li>• Typen</li> <li>• Ausnahmen</li> <li>• Werten</li> </ul>

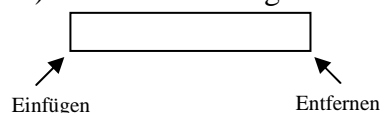
**Frage:** Implementiert eine gegebene Struktur eine gegebene Signatur?

## 15.1 Signatur: Spezifikation

signature sigid = sig specs end

### Schlange (Warteschlange):

Datenstruktur (FIFO): Elemente eintragen und entfernen



**Beispiel für eine Signatur:**

```
signature QUEUE =
  sig
    type 'a queue
    exception Empty
    val empty : 'a queue
    val insert : 'a * 'a queue -> 'a queue
    val remove : 'a queue -> 'a * 'a queue
  end
```

## 15.2 Struktur: Implementierung

Wie sollen wir QUEUE implementieren? Über eine Liste? (Einfüge- und Entfernungsliste)  
 siehe Struktur Queue\_one\_list

Laufzeit:      insert  $O(1)$   
                  remove  $\Theta(n)$

**Invariante:** Schlange = Einfügeliste @ (rev Entfernungsliste)

**Schlange als 2 Listen:**

Einfügeliste       $is = [x_1, \dots, x_n]$   
 Entfernungsliste       $rs = [y_1, \dots, y_m]$   
 Invariante       $q = [x_1, \dots, x_n, y_m, \dots, y_1]$

$insert(x, ([x_1, \dots, x_n], ys)) = ([x, x_1, \dots, x_n], ys)$   
 $remove(xs, [y_1, \dots, y_m]) = (y_1, (xs, [y_2, \dots, y_m]))$   
 $remove([x_1, \dots, x_n], []) = (x_n, ([], [x_{n-1}, \dots, x_1]))$

**Beispiel für eine Struktur:**

```
structure Queue =
  struct
    type 'a queue = 'a list * 'a list
    exception Empty
    val empty = (nil, nil)
    fun insert (x, (il, rl)) = (x::il, rl)
    fun remove (nil, nil) = raise Empty
      | remove (il, nil) = remove (nil, rev il)
      | remove (il, r::rl) = (r, (il, rl))
  end
```

## 15.3 Module, Signaturabgleich

```
structure strid : sigid = strexp
```

Frage : Implementiert die Struktur die Signatur ?

Bedingungen:

- Die Struktur muss alle Komponenten zur Verfügung stellen, die die Signatur fordert
- Die Struktur muss alle Typanforderungen erfüllen, die die Signatur fordert.  
Die implementierten Typen können allgemeiner sein als die spezifizierten Typen
- Ausnahmen müssen „typäquivalent“ sein
- Typkomponenten müssen „typäquivalent“ sein

### Typäquivalenz

type-Definition `type tid = ty`

führt äquivalente Typen ein.

Im Gültigkeitsbereich dieser Deklaration können tid und ty äquivalent verwendet werden

### Prinzipale Signatur einer Struktur

Analog: Prinzipaler Typ eines Ausdrucks ist der allgemeinste Typ, mit dem Typberechnung für das Programm erfolgreich war.

SML: In einem typkorrekten Programm hat jeder Ausdruck einen bis auf Umbenennung der Typvariablen eindeutig bestimmten prinzipalen Typ.

Strukturen nach erfolgreicher Typberechnung: genau eine prinzipale Signatur

Ist eine Struktur einmal typanalysiert, dann muss ihre Definition für die weitere Typberechnung nicht mehr betrachtet werden.

### Definition (prinzipale Signatur)

Die prinzipale Signatur einer Struktur besteht aus Typdefinition, Datentypdefinition, Ausnahmedefinition und den prinzipalen Typen der Wertbindungen.

### Definition

Signatur  $S_2$  **passt zu** Signatur  $S_1$ , i.Z.  $S_2 \succeq S_1$ , genau dann wenn

1. Jeder Typkonstruktor in  $S_1$  muss auch in  $S_2$  definiert sein und zwar mit der gleichen Stelligkeit und äquivalenter Definition.
2. Jeder Datentyp in  $S_1$  muss auch in  $S_2$  definiert sein und zwar mit äquivalenter Definition der Wert-Konstrukturen.
3. Jeder Ausnahme-Konstruktor in  $S_1$  muss auch in  $S_2$  definiert sein und zwar mit äquivalentem Argumenttyp.
4. Jeder Wert in  $S_1$  muss auch in  $S_2$  vorkommen mit einem mindestens so allgemeinen Typ.

Signatur mit **Datentyp-Deklaration**  $\succeq$  Signatur mit **Typdeklaration**

wenn für jede Funktion, die Werte des **Typs** konstruiert, ein **Wert-Konstruktor** mit passenden Argumenttypen existiert.

### Beispiel:

signature RBT\_DT =

sig

datatype 'a rbt =

Empty |

Red of 'a rbt \* 'a \* 'a rbt |

Black of 'a rbt \* 'a \* 'a rbt

end

$\succeq$

signature RBT =

sig

type 'a rbt

val Empty: 'a rbt

val Red: 'a rbt \* 'a \* 'a rbt  $\rightarrow$  'a rbt

end

## 15.4 Antwort

**Eine Struktur St implementiert eine Signatur Si, genau dann wenn die prinzipale Signatur von St zu Si passt.**

## 15.5 Lange Namen

Zugriffe auf Struktur-Komponenten

- Lange Namen: Queue\_1\_liste.empty  
hat den Typ: 'a Queue\_1\_liste.queue
- open Queue\_1\_liste  
macht alle internen Namen verfügbar  
empty statt Queue\_1\_liste.empty  
**Gefahr:** Namen von Komponenten können existierende Namen verdecken
- selektives Umbenennen  
val qinsert = Queue\_one\_list.insert