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

Module

- Mit *Modulen* kann man größere Programme hierarchisch strukturieren.
- Die Modulbegriffe von SML heißen:
  - Struktur
  - Signatur
  - Funktor

Strukturen

- SML - *Strukturen* sind Teilprogramme.
- In einer SML - Struktur können Werte, Typen und Ausnahmen definiert werden, wobei auch Funktionen erlaubte Werte sind.

Signatur

- Eine SML - *Signatur* teilt mit, was in einer Struktur implementiert wird, ohne die Implementierung selbst preiszugeben.
- Eine Signatur ähnelt insofern dem Typ einer Funktion, der in abgekürzter Form (oder abstrakt) wiedergibt, was die Funktion leistet, ohne die Implementierung der Funktion preiszugeben.

Funktor

- Ein SML - *Funktor* ist ein parametrisiertes Modul.
- Ein SML - Funktor spezifiziert, wie aus Strukturen neue Strukturen gebildet werden können und ähnelt insofern einer Funktion, die aber nicht Werte auf Werte, sondern Strukturen auf Strukturen abbildet.

SML – Strukturen

- Deklarationen können in einer SML - Struktur zusammengefasst werden.
- Beispiel:

  Eine Struktur, die alle Deklarationen, die zur Definition eines Typs „komplexe Zahlen“ benötigt werden, zusammenfasst.

Beispiel – Implementierung: „komplexe Zahlen“

```
structure Complex =
  struct
      type t = real * real;
      val zero = (0.0, 0.0) : t;
      fun sum ((x1,y1):t,(x2,y2):t) = (x1+x2,y1+y2) : t;
      fun difference((x1,y1):t,(x2,y2):t) = (x1-x2,y1-y2) : t;
      fun product ((x1,y1):t,(x2,y2):t) =
            (x1*x2-y1*y2,x1*y2+x2*y1) : t;
      fun reciprocal((x,y) : t) =
            let val r = x * x + y * y
            in
               (x/r, ~y/r) : t
            end;
      fun quotient (z1:t,z2:t) = product(z1,reciprocal z2)
  end;
```


Sichtbarkeit

- Die Namen, die in einer Struktur deklariert sind, sind außerhalb dieser Struktur nicht sichtbar.
- Beispiel:

  Die Funktion `reciprocal` kann in der Struktur verwendet werden (z.B. in der Definition der Funktion `quotient`).

  Außerhalb der Struktur ist sie unbekannt:

```
- reciprocal(1.0, 0.0);
Error: unbound variable or constructor: reciprocal
```

Zugriff von Außen

- Ein Name `N`, der in einer Struktur namens `S` deklariert ist, kann außerhalb dieser Struktur als `S.N` verwendet werden.
- So ist der Zugriff auf die Funktion `reciprocal` außerhalb der Struktur `Complex` durch den Namen `Complex.reciprocal` möglich:

```
- Complex.reciprocal(1.0, 0.0);
val it = (1.0,0.0) : Complex.t
```


Konvention bei Strukturen zur Definition eines Typs

- Dient eine Struktur namens `S` zur Definition eines Typs, so benennt man diesen Typ üblicherweise mit `t` innerhalb der Struktur und `S.t` außerhalb der Struktur:

```
- val i = (0.0, 1.0) : Complex.t;
val i = (0.0,1.0) : Complex.t

- Complex.product(i, i);
val it = (~1.0,0.0) : Complex.t
```

Vorteil eines abstrakten Typs

- Zur Definition des Typs „komplexe Zahlen“ könnte auch ein abstrakter Typ benutzt werden.
- Er hätte gegenüber der besprochenen Struktur den Vorteil, die Implementierung der Werte zu verbergen und so unbeabsichtigte Verwendungen zu verhindern.

Mitteilung von SML

- Die Mitteilung, die SML liefert, wenn die Struktur `Complex` deklariert wird, ist die Signatur dieser Struktur.
- Diese Signatur besteht aus den Mitteilungen, die gegeben würden, wenn die Deklarationen der Struktur einzeln definiert würden:

```
structure Complex :
    sig
          type t = real * real
          val difference : t * t -> t
          val product : t * t -> t
          val quotient : t * t -> t
          val reciprocal : t -> t
          val sum : t * t -> t
          val zero : t
    end
```

SML – Signaturen

- Eine *SML - Signatur* ist eine abstrakte Beschreibung der Deklarationen einer SML - Struktur oder mehrerer SML - Strukturen.
- Eine Signatur wird nicht nur vom SML - System aus einer Strukturdeklaration ermittelt, sondern kann auch vom Programmierer selbst deklariert werden.

Beispiel

- Eine Signatur, die jede Struktur, die einen Typ `t` sowie die grundlegenden arithmetischen Operationen über `t` implementiert.

Beispiel – Implementierung

```
- signature ARITHMETIC =
  sig
       type t
       val zero : t
       val sum : t * t -> t
       val difference : t * t -> t
       val product : t * t -> t
       val reciprocal : t -> t
       val quotient : t * t -> t
  end;
signature ARITHMETIC =
  sig
       type t
       val zero : t
       val sum : t * t -> t
       val difference : t * t -> t
       val product : t * t -> t
       val reciprocal : t -> t
       val quotient : t * t -> t
end
```

Die Ausdrücke einer Signatur, die zwischen den reservierten Wörtern `sig` und `end` vorkommen, heißen *(Signatur-) Spezifikationen*.

Signatur – Constraints

- Die Signatur `ARITHMETIC` kann in sogenannten *Signatur - Constraints* verwendet werden, wenn eine Struktur definiert wird, die alle in der Signatur spezifizierten Komponenten deklariert.

Beispiel (1)

```
- structure Rational : ARITHMETIC =
      struct
         type t = int * int;
         val zero = (0, 1) : t;
         fun sum ((x1,y1):t,(x2,y2):t) =
             (x1*y2+x2*y1,y1*y2) :t;
         fun difference((x1,y1):t,(x2,y2):t) =
             (x1*y2 - x2*y1, y1*y2) :t;
         fun product ((x1,y1):t,(x2,y2):t) =
             (x1 * x2, y1 * y2) : t;
         fun reciprocal((x,y) : t) = (y,x) : t
         fun quotient (z1 : t, z2 : t) =
             product(z1, reciprocal z2)
      end;
structure Rational : ARITHMETIC
```


Beispiel (2)

```
- structure Complex : ARITHMETIC =
      struct
         type t = real * real;
         val zero = (0.0, 0.0) : t;
         fun sum ((x1,y1):t,(x2,y2):t) = (x1 + x2, y1 + y2):t;
         fun difference((x1,y1):t,(x2,y2):t) =
             (x1 - x2, y1 - y2) : t;
         fun product ((x1,y1):t, (x2,y2):t) =
             (x1 * x2 - y1 * y2, x1 * y2 + x2 * y1) : t;
         fun reciprocal((x,y) : t) =
             let val r = x * x + y * y
             in (x/r, ~y/r) : t
             end;
         fun quotient (z1 : t, z2 : t) =
             product(z1, reciprocal z2)
      end;
structure Complex : ARITHMETIC
```

Gleiche Signatur unterschiedliche Typen

- Die Typabkürzungen `Rational.t` und `Complex.t` bezeichnen unterschiedliche Typen:
  - `int * int`
  - `real * real`,

  obwohl die beiden Strukturen `Rational` und `Complex` dieselbe Signatur `ARITHMETIC` haben.

Spezifikation versus Deklaration in SML - Signaturen

- Eine Signatur besteht aus *Spezifikationen*, nicht aus *Deklarationen*.
- Beispiel:

  `type t` ist eine Spezifikation:
  - Diese Spezifikation teilt mit, dass in der Signatur der Name `t` einen Typ bezeichnet.
  - Sie teilt NICHT mit, wie der Typ `t` definiert ist.
  - Insbesondere teilt sie NICHT mit, welche (Wert-) Konstruktoren der Typ `t` hat.

Spezifikationen und Deklarationen

- Um den Unterschied zwischen Spezifikationen und Deklarationen zu unterstreichen, darf das reservierte Wort `fun` in einer Signatur nicht vorkommen, sondern nur das reservierte Wort `val` .
- Beispiel:

```
val sum : t * t -> t
```


`eqtype` – Spezifikationen in SML - Signaturen

- In einer SML - Signatur kann die Spezifikation eines Typs `t` mit dem reservierten Wort `eqtype` statt `type` eingeführt werden, wenn die Gleichheit über `t` definiert sein muss.
- Erinnerung:

  Über Funktionstypen ist keine Gleichheit definiert.

`datatype` - Spezifikationen in SML-Signaturen

- In einer SML - Signatur kann die Spezifikation eines Typs `t` auch in Form einer `datatype` - Deklaration erfolgen.
- In diesem Fall ist nicht nur spezifiziert, dass `t` ein Typ ist, sondern auch, welche (Wert-) Konstruktoren `t` hat.
- Ein Beispiel dafür kommt in der Signatur `LIST` vor (vgl. Kapitel 11.5).

Angleich einer Struktur an eine Signatur - Strukturansichten

- Eine Struktur `Struk` kann an eine Signatur `Sig` angeglichen werden, wenn alle Komponenten, die in der Signatur `Sig` spezifiziert werden, in der Struktur `Struk` deklariert sind.
- Man kann eine Struktur an eine Signatur angleichen, die weniger Komponenten spezifiziert, als die Struktur deklariert.
- So können eingeschränkte Sichten (*views*) auf eine Struktur definiert werden.

Beispiel

- Die Signatur spezifiziert nur einen Teil der Namen, die in der Struktur `Complex` deklariert sind:

```
- signature RESTRICTED_ARITHMETIC =
      sig
            type t
            val zero : t
            val sum : t * t -> t
            val difference : t * t -> t
      end;
signature RESTRICTED_ARITHMETIC =
      sig
            type t
            val zero : t
            val sum : t * t -> t
            val difference : t * t -> t
      end
```


RESTRICTED_ARITHMETIC (1)

- Unter Verwendung der Signatur `RESTRICTED_ARITHMETIC` erzeugt die folgende Deklaration (nächste Folie) eine Einschränkung der Struktur `Complex`, die nur die Namen zur Verfügung stellt, die in der Signatur `RESTRICTED_ARITHMETIC` vorkommen.
- Die Definition dieser Namen ist diejenige aus der (uneingeschränkten) Struktur `Complex`.

RESTRICTED_ARITHMETIC (2)

```
- structure RestrictedComplex : RESTRICTED_ARITHMETIC =
    Complex;
structure RestrictedComplex : RESTRICTED_ARITHMETIC

- val i = (0.0, 1.0) : RestrictedComplex.t;
val i = (0.0,1.0) : Complex.t

- RestrictedComplex.sum(RestrictedComplex.zero, i);
val it = (0.0,1.0) : Complex.t

- RestrictedComplex.product(RestrictedComplex.zero, i);
Error: unbound variable or constructor: product in path
    RestrictedComplex.product

- Complex.product(RestrictedComplex.zero, i);
val it = (0.0,0.0) : Complex.t
```


Signatur – Constraint

- In der Deklaration der Struktur `RestrictedComplex` ist

  `: RESTRICTED_ARITHMETIC`

  ein *Signatur - Constraint*.
- Mit Signatur - Constraints kann also eine Form des Verbergens (*information hiding*) erreicht werden.

Parametrisierte Module in SML: SML – Funktoren (1)

- Beide Strukturen, `Rational` und `Complex`, können um eine Funktion `square` erweitert werden, die eine rationale bzw. komplexe Zahl auf ihr Quadrat abbildet.
- Die Deklaration einer solchen Funktion `square` wäre in den beiden Strukturen identisch:

```
fun square z = product(z, z)
```


Parametrisierte Module in SML: SML – Funktoren (2)

- Wäre jede der beiden Strukturen `Rational` und `Complex` um diese Deklaration ergänzt, so wären die Funktionen `Rational.square` und `Complex.square` trotzdem unterschiedlich definiert:
  - Jede Deklaration bezieht sich auf die Definition der Funktion `product` aus der eigenen Struktur.
  - `Rational.product` und `Complex.product` sind unterschiedlich definiert.
- Es ist kein Zufall, dass beide identisch deklariert werden können, auch wenn sie unterschiedlich definiert sind:

  Jeder Zahlentyp, der über ein Produkt verfügt, ermöglicht dieselbe Definition des Quadrats in Bezug auf dieses Produkt.

SML-Funktoren (1)

- Die Erweiterung beider Strukturen um eine Funktion `square` wäre allerdings nachteilig.
- Sie würde die Identität der Deklarationen der beiden Funktionen `square` verbergen und so etwaige Veränderungen dieser Funktionen erschweren.
- Genauso wie es sich anbietet, eine Teilberechnung, die in einem Algorithmus mehrfach vorkommt, als Prozedur zu definieren, so bietet es sich an, Deklarationen, die in verschiedenen Strukturen identisch sind, nur einmal für die verschiedenen Strukturen anzugeben.
- Dazu dienen *SML - Funktoren*.

SML-Funktoren (2)

- Ein *SML - Funktor* ist eine SML - Struktur, die andere SML - Strukturen als Parameter erhält, also eine parametrisierte Struktur.

Beispiel

- Die Funktion `square` kann für alle Strukturen mit der Signatur `ARITHMETIC` definiert werden:

```
- functor Extended(S : ARITHMETIC) =
      struct
            fun square z = S.product(z, z)
      end;
functor Extended : <sig>
```

- Der Funktor `Extended` stellt erst dann eine benutzbare Struktur dar, wenn der Name `S` an eine Struktur mit Signatur `ARITHMETIC` gebunden wird, also eine Struktur wie `Rational` oder `Complex`.

Beispiel

```
- structure ExtComplex = Extended(Complex);
structure ExtComplex : sig val square : S.t -> S.t end

- val i = (0.0, 1.0) : Complex.t;
val i = (0.0,1.0) : Complex.t

- ExtComplex.square(i);
val it = (~1.0,0.0) : Complex.t

- structure ExtRational = Extended(Rational);
structure ExtRational : sig val square : S.t -> S.t end

- val minusone = (~1, 1) : Rational.t;
val minusone = (~1,1) : Rational.t

- ExtRational.square(minusone);
val it = (1,1) : Rational.t
```

Extended

- Die Quadratfunktionen der beiden Strukturen `ExtComplex` und `ExtRational` sind, wie gewünscht, unterschiedliche Funktionen.
- Ihre Deklarationen im Funktor `Extended` sind allerdings syntaktisch identisch :

```
- ExtComplex.square(0.0, 1.0);
val it = (~1.0,0.0) : Complex.t

- ExtRational.square(0, 1);
val it = (0,1) : Rational.t
```


Typprüfung eines Funktors

- Während der (statischen) Typprüfung eines Funktors wird ausschließlich die Information verwendet, die die Signatur liefert.
- So entsteht die Abstraktion, die ermöglicht, dass zur Laufzeit ein Funktor auf unterschiedliche Strukturen angewendet wird.
- Sobald ein Funktor keine Typfehler enthält, kann er auf jede Struktur angewendet werden, die mit der Signatur des Parameters des Funktors angeglichen werden kann.
- Es wäre übrigens naheliegend, die Signatur `ARITHMETIC` ohne die Funktion `quotient` zu definieren und diese Funktion in generischer Weise durch einen Funktor auf `product` und `reciprocal` zurückzuführen.

Generative und nichtgenerative Strukturdeklarationen

- Zwei Formen von Strukturdeklarationen sind möglich:
  - *generative* Strukturdeklaration
  - *nichtgenerative* Strukturdeklaration

Generative Strukturdeklaration

```
- structure RestrictedComplex =
    struct
      type t = real * real;
      val zero = (0.0, 0.0) : t;
      fun sum ((x1,y1), (x2,y2)) =
                  (x1 + x2, y1 + y2) : t;
      fun difference((x1,y1), (x2,y2)) =
                  (x1 - x2, y1 - y2) : t;
    end;
```

- Diese Strukturdeklaration deklariert ihre Komponenten explizit.

Nichtgenerative Strukturdeklaration

```
- structure RestrictedComplex : RESTRICTED_ARITHMETIC =
  Complex;
structure RestrictedComplex : RESTRICTED_ARITHMETIC
```

- In solchen Strukturdeklarationen werden die Komponenten der Struktur nicht explizit, sondern implizit in Bezug auf bereits vorgenommene Deklarationen definiert.

Weiteres über Module in SML

- Geschachtelte Strukturen
- Lange Namen (oder „lange Bezeichner“)
- Teilen von Deklarationen in Strukturen

Beispiel: Geschachtelte Strukturen

```
- structure Nb =
    struct
      structure ComplexNb = Complex;
      structure RationalNb = Rational
    end;
structure Nb :
    sig
      structure ComplexNb : <sig>
      structure RationalNb : <sig>
    end
```


Anwendungen

```
- val i = (0.0, 1.0) : Nb.ComplexNb.t;
val i = (0.0,1.0) : Complex.t

- Nb.ComplexNb.product(i, Nb.ComplexNb.zero);
val it = (0.0,0.0) : Complex.t

- val minusone = (~1, 1) : Nb.RationalNb.t;
val minusone = (~1,1) : Rational.t

- Nb.RationalNb.sum(minusone, Nb.RationalNb.zero);
val it = (~1,1) : Rational.t
```

Lange Namen

- Namen wie:
  - `Complex.t`,
  - `Complex.product` oder
  - `Nb.RationalNb.sum`

  werden „*lange Namen*“ oder „*lange Bezeichner*“ genannt.
- Da geschachtelte Strukturen möglich sind, können lange Namen aus mehr als zwei (nichtlangen) Namen bestehen.
- Beispiel:

  `Nb.RationalNb.sum`.

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

Standardbibliothek von SML

- Die Standardbibliothek von SML, die unter der URI http://www.smlnj.org//doc/basis/ verfügbar ist, besteht aus Modulen.
- In ihr kommen Signaturdeklarationen und Spezifikationen vor.
- Ein Beispiel stellen die Spezifikationen der Struktur `List` dar:

  siehe http://www.smlnj.org//doc/basis/pages/list.html

Zugehörige Signaturen herausfinden

- Wenn man den Namen einer Struktur wie `List` kennt, kann man im SML - System die zugehörige Signatur herausfinden, indem man einen neuen (irrelevanten) Strukturnamen deklariert:

```
- structure Struct = List;
structure Struct : LIST
```

Schnittstelle einer Struktur herausfinden

- Die Mitteilung des SML - Systems enthält also die Information, dass die Struktur `List` die Signatur `LIST` hat.
- Deklariert man dafür wiederum einen neuen (irrelevanten) Signaturnamen, erfährt man aus der Mitteilung des SML - Systems die Schnittstelle der Struktur.

Beispiel `LIST` (1)

```
- signature SIG = LIST;
signature LIST =
  sig
      datatype 'a list = :: of 'a * 'a list | nil
      exception Empty
      val null : 'a list -> bool
      val hd : 'a list -> 'a
      val tl : 'a list -> 'a list
      val last : 'a list -> 'a
      val getItem : 'a list -> ('a * 'a list) option
      val nth : 'a list * int -> 'a
      val take : 'a list * int -> 'a list
      val drop : 'a list * int -> 'a list
      val length : 'a list -> int
      val rev : 'a list -> 'a list
      val @ : 'a list * 'a list -> 'a list
```

Beispiel `LIST` (2)

```
    val concat : 'a list list -> 'a list
    val revAppend : 'a list * 'a list -> 'a list
    val app : ('a -> unit) -> 'a list -> unit
    val map : ('a -> 'a) -> 'a list -> 'b list
    val mapPartial : ('a -> 'b option) -> 'a list -> 'b list
    val find : ('a -> bool) -> 'a list -> 'a option
    val filter : ('a -> bool) -> 'a list -> 'a list
    val partition : ('a -> bool) -> 'a list -> 'a list * 'a list
    val foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
    val foldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
    val exists : ('a -> bool) -> 'a list -> bool
    val all : ('a -> bool) -> 'a list -> bool
    val tabulate : int * (int -> 'a ) -> 'a list
end
```


Hinweis

- Mit einer gewissen Erfahrung in der Listenverarbeitung reicht diese Information oft aus, um die geeignete Funktion für das gerade bearbeitete Problem zu finden.
- Die erwähnte Seite der Standardbibliothek enthält im wesentlichen diese Signatur sowie eine kurze verbale Beschreibung aller Funktionen der Signatur.