Modularisierung großer funktionaler Programme

Konzepte zur Definition abstrakter Datentypen

Zusammenfassung von Daten mit den darauf definierten Operationen zu einer Einheit.

- abstype
  Spezifikation der Operationen kann nicht von der Implementierung getrennt werden (*konkreter Typ*)

- structure, signature, functor
  * Spezifikationsteil (signature) und Implementierungsteil (structure) können voneinander getrennt werden.
  * Strukturen können mittel Funktoren parametrisiert werden.

Strukturen

```
structure <Strukturbezeichner> = struct
      <beliebige Definitionen von Typen, Werten und Operationen
        gemäß bekannter ML-Syntax>
end.

Bsp.
- structure S = struct
          type t = int
          val x = 3;
          fun f(x) = if x=0 then 1 else x * f(x-1)
        end;
structure S :
  sig
    type t = int
    val f : int -> int
    val x : int
  end
```

Zugriff auf gekapselte Definitionen

<Strukturbezeichner>**.**<Bezeichner>

```
- S.x;
val it = 3 : int

- S.f(S.x);
val it = 6 : int
```

```
- let open S in f(x) end;
val it = 6 : int

- open S;
opening S
  type t = int
  val f : int -> int
  val x : int

- x:t;
val it = 3 : t
```


Signaturen

signature <Signaturbezeichner> = **sig**
<Folge von Typdeklarationen und Deklarationen von Bezeichnern mit ihrer Funktionalität>
end;

Bsp.

```
- signature SIG =
    sig
      val x : int
      val b : bool
end;
signature SIG =
  sig
    val x : int
    val b : bool
  end
```

Spezifikationsmöglichkeiten:

- Wert- und Funktionsdeklarationen
  val <Bezeichner> : <Typ>
- Typdefinitionen
  type <Typvariablen> = <Typbezeichner>
- Erweiternde Typdefinitionen
  datatype ...

signature matching

structure <Strukturbezeichner> **:** <Signaturbezeichner> = **struct**
<Folge von Typdeklarationen und Deklarationen von Bezeichnern mit ihrer Funktionalität>
end;

Bsp.

```
- structure S: SIG =
    struct
      val x = 2+1
      val b = x=7
      type object=int
    end;
structure S : SIG
```

Matchingregeln

Regel 1: Matching von Bezeichnern.
Zu jedem Bezeichner, der in einer Signatur definiert wird, muß es eine zugehörige Definition in der Struktur geben.

Regel 2: Matching von Typen.
Existiert für einen Bezeichner der Struktur eine entsprechende Definition in der Signatur, dann müssen die Typen übereinstimmen, wobei gilt:
a) Jede Typdefinition für den Typ T in der Struktur paßt zur Spezifikation type T in der Signatur.
b) Nur eine identische datatype-Definition für den Typ T in der Struktur paßt zur Spezifikation datatype T = ... in der Signatur.

Regel 3: Private Objekte.
Jede Deklaration in einer Struktur, zu der es keine entsprechende Definition in der zugehörigen Signatur gibt, ist bzgl. der Struktur privat, also außerhalb nicht sichtbar.

Bsp. Matching Struktur-Signatur

```
signature order = sig
    type T;
    val lesseq: T * T -> bool
  end;
structure intorder: order = struct
    type T=int;
    fun lesseq(x:T,y:T)=x≤y
  end;
structure natorder: order = struct
    datatype T=null | succ of T;
    fun lesseq(null,_) = true | lesseq(_,null) = false |
    lesseq(succ x,succ y) = lesseq(x,y)
  end;
structure natpairorder: order = struct
    type T=natorder.T  *natorder.T;
    fun lesseq((x,y),(x',y'))=natorder.lesseq(x,x') orelse (x=x'
    andalso natorder.lesseq(y,y'))
  end;
```


Funktoren

```
functor <Funktorbezeichner> (structure <Bezeichner 1>: <Signatur 1>;...;
                structure <Bezeichner n>: <Signatur n>): <Signatur>=struct
      <Definitionen wie bei Strukturen>
end;
```

Funktor: Strukturen → Struktur

Analogien
Funktor ↔ Funktion
Struktur ↔ Wert
Signatur ↔ Typ

Bsp. Funktoren

```
fun min [x]=x: real | min (x::y::z)=if x≤y then min(x::z) else min (y::z);

- signature minimum = sig
      type T
      val min: T list -> T
  end;

- functor make_min (structure act_order: order): minimum = struct
      type T = act_order.T;
      fun min[x] = x | min (x::y::z = if act_order.lesseq(x,y) then min (x::z)
      else min (y::z)
  end;

- structure natpairmin = make_min(structure natpairorder:order);

- natpairmin.min [(null,succ null),(succ null,succ(succ null)),...].
```


I/O-System-Struktur

Imperative
Stream-oriented
Primitive

```
structure TextIO : TEXT_IO

signature TEXT_IO

Interface
   include IMPERATIVE_IO
   structure StreamIO : TEXT_STREAM_IO
   val inputLine : instream -> string
   val outputSubstr : (outstream * substring) -> unit
   val openIn : string -> instream
   val openOut : string -> outstream
   val openAppend : string -> outstream
   val openString : string -> instream
   val stdIn : instream
   val stdOut : outstream
   val stdErr : outstream
   val print : string -> unit
   val scanStream : ((Char.char,StreamIO.instream) StringCvt.reader ->
   ('a,StreamIO.instream) StringCvt.reader) -> instream -> 'a option
```

```
open TextIO;
val myfile=openOut("testfile");
output(myfile,"Zeilea\r\n");
output(myfile,"Zeileb\r\n");
flushOut myfile;
closeOut;
```