Abstraktion

- `fn <Argumentmuster> => <Ausdruck>`
- Für `fun f ... = ...`
  können wir auch `val f = fn ... => ...` schreiben.
- Wenn `f` rekursiv: `val rec f = fn ... => ...`

Abstraktion

Achtung Syntax!

- ```
  fn (x:int) => 1.0;
  > val it = fn :  int -> real
  ```
- ```
  fn (x:int):real => 1.0;
  !  Toplevel input:
  !  fn (x:int):real => 1.0;
  !  ^^^^^^^
  !  Type clash:  pattern of type
  !  int
  !  cannot have type
  !  real
  ```
- ```
  fun f (x:int):real = 1.0;
  > val f = fn :  int -> real
  ```
- ```
  fn (x:int):int => 1
  > val it = fn :  int -> int
  ```

Syntaxregeln

- `<Prozedurdeklaration> ::=`
  `fun <Bezeichner> <Argumentmuster> = <Ausdruck>`
  `| fun <Bezeichner> <Argumentmuster> : <Typ> = <Ausdruck>`

aber:

- `fn <Argumentmuster> => <Ausdruck>`
- `<Argumentmuster> ::=`
  `( <Argumentspezifikation> , ... , <Argumentspezifikation> )`
  `| <Argumentmuster> : <Typ>`
- `<Argumentspezifikation> ::= <Bezeichner> : <Typ>`

Damit:

- `fn (x:int) => 1.0;`
  `> val it = fn :  int -> real`
- `fn (x:int) => 1.0:real;`
  `> val it = fn :  int -> real`
- `fn (x:int):int => 1;`
  `> val it = fn :  int -> int`
- `fn (x:int):int:int:int:int => 1;`
  `> val it = fn :  int -> int`
- `fn (x:int, y:real):int*real => 1;`
  `> val it = fn :  int * real -> int`

aber:

- `fn (x:int):real => 1.0;`
  `!  Type clash`

Kaskadierte und höherstufige Prozeduren

- **kaskadierte Prozeduren:**
  Prozeduren, die Prozeduren als Ergebnis liefern:
  - kaskadierte Darstellung:
    ```
    fun mul (x:int) = fn (y:int) => x*y;
    fun mul (x:int) (y:int) = x*y;
    > val mul = fn :  int -> int -> int
    ```
  - kartesische Darstellung:
    ```
    fun mult (x:int, y:int) = x*y;
    > val mult = fn :  int * int -> int
    ```
- **höherstufige Prozeduren:**
  Prozeduren bei denen eines der Argumente eine Prozedur ist
  ```
  fun sum (f:int->int) (n:int):int =
  if n<1 then 0 else sum f (n-1) + f n
  ```

Beispiele

- **Bestimmte iteration: Iter**
  - $\underbrace{f(\ldots(f(f}_{n\text{-mal}}(s))\ldots)$
  - `iter fun iter (n:int) (s:int) (f: int -> int) : int`
    `if n < 1 then s else iter (n-1) (f s) f`

- **Unbestimmte Iteration: first**
  - $\mathit{first}\ s\ p = \min\{x \in \mathrm{Z} \mid x \geq s \text{ und } p\ x = \mathit{true}\}$
  - `fun first (s:int) (p:int-> bool) : int =`
    `if p s then s else first (s+1) p`

Polymorphe Typisierung

- Polymorphes iter: `fun 'a iter (n:int) (s:'a) (f:'a->'a) : 'a =`
  `if n<1 then s else iter (n-1) (f s) f`
- **Typschema:** $\forall\alpha$. `int -> `α` -> (`α` -> `α`) -> `α
- **Instanzen** des Schemas:
  `int -> int -> (int->int) -> int` $(\alpha = \texttt{int})$
  `int -> real -> (real->real) -> int` $(\alpha = \texttt{real})$
  `int -> int*int -> (int*int->int*int) -> int`
  $(\alpha = \texttt{int} * \texttt{int})$

- Bezeichner heisst **polymorph**
  wenn er mit einem Typschema getypt ist
- sonst: **monomorph**