4.5.5 Rekursive Typen

Die Definition eines Typen kann rekursiv sein, d.h.
Typ-Konstruktoren dürfen Elemente des zu definierenden Typ erhalten.

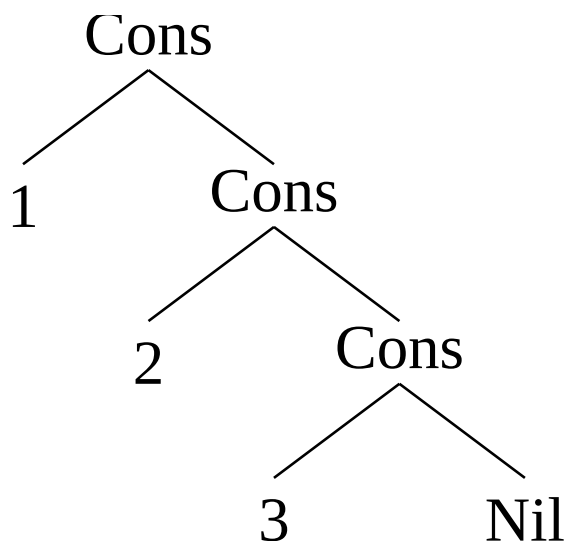
```
datatype IntList = Nil | Cons of (int*IntList);
```

Damit Werte konstruierbar sind, muss midestens ein nullstelliger Konstruktor angegeben werden. Bsp.: Liste mit Elementen 1, 2, 3

```
Cons(1,
=       Cons(2,
=            Cons(3,Nil)));
val it = Cons (1,Cons (2,Cons (3,nil))) : IntList
```

Aufbau einer Liste

```
Cons(1,Cons(2,Cons(3,Nil)));
val it = Cons (1,Cons (2,Cons (3,nil))) : IntList
```



Verarbeitung rekursiver Datentypen

... erfolgt mit Hilfe von Pattern-Matching und rekursive Funktionen

Die Länge einer Liste:

```
fun length l =
  case l of
      Nil => 0
    | Cons(first,rest) => 1 + length rest;
val length = fn : IntList -> int

length (Cons(1,Cons(2,Cons(3,Nil))));
val it = 3 : int
```

4.5.6 Polymorphismus

Polymorphe Typen

Listentypen unterscheiden sich nur in den Typ ihrer Elemente:

```
datatype IntList = Nil | Cons of (int * IntList);
```

```
datatype RealList = Nil | Cons of (real * RealList);
```

Besser als verschiedene Definitionen: parametrisieren:

```
datatype 'a List = Nil | Cons of ('a * 'a List);
datatype 'a List = Cons of 'a * 'a List | Nil
```

'a ist ein Bezeichner für einen Typ (Typ-Variable)

Polymorphe Typen

```
datatype 'a List = Nil | Cons of ('a * 'a List);
```

Der Typ `List` ist parametrisiert ≡ `List` ist ein (postfix) Typ-Operator.

Diese Art von Polymorphismus heißt parametrischer Polymorphismus.

Der Compiler erkennt den Parameter-Typ automatisch:

```
- Cons(1,Cons(2,Cons(3,Nil)));
val it = Cons (1,Cons (2,Cons 3)) : int List
- Cons(1.0,Cons(2.0,Cons(3.0,Nil)));
val it = Cons (1.0,Cons (2.0,Cons 3.0)) : real List
- Cons(true,Cons(true,Cons(false,Nil)));
val it = Cons (true,Cons (true,Cons false)) : bool List
```

Polymorphe Funktionen

```
datatype 'a List = Nil | Cons of ('a * 'a List)
```

Der Compiler leitet den allgemeinst möglichen Typ ab:

```
fun length l =
    case l of
        Nil => 0
      | Cons(first,rest) => 1 + length rest;
val length = fn : 'a List -> int
```

Hier ist 'a als *ein beliebiger Typ* zu lesen

$\Longrightarrow$ `length` is polymorph:

Polymorphe Funktionen

```
length (Cons(1,Cons(2,Cons(3,Nil))));
val it = 3 : int
- length (Cons(1.0,Cons(2.0,Cons(3.0,Nil))));
val it = 3 : int
- length (Cons(true,Cons(true,Cons(false,Nil))));
val it = 3 : int
```

Der polymorphe Typ `list`

Ein polypmorpher Typ `'a list` ist vordefiniert.

Konstruktoren:

- nullstellig: `nil` (entspricht unserem `Nil`)
- zweistellig: `::` (entspricht unseren `Cons`)
  - ▷ steht zwischen seinen Argumenten: `kopf::rest`
  - ▷ `::` ist rechtsassoziativ:
    `1::(2::(3::nil))` ≡ `1::2::3::nil`
- Alternative Klammernotation:
  - ▷ `[]` ≡ `nil`
  - ▷ `[1,2,3]` ≡ `1::(2::(3::nil))` ≡ `1::[2,3]`

Der polymorphe Typ `list`

```
- fun length l = case l of
                    nil => 0
                  | first::rest => 1 + length rest;
val length = fn : 'a list -> int
- length [1,2,3];
val it = 3 : int
- length [true,true,false];
val it = 3 : int
```

- Alle Elemente einer Liste müssen vom selben Typ sein:

  [1,[1]] ist keine Liste!

- Liste von Listen von `int`s: `[[1,2,3],[4,5],[6],[7,8,9]]`

List-Typen: Beispiele

- Alle Elemente einer Liste müssen vom selben Typ sein:

```
- [1,[1]];
stdIn:93.1-93.8 Error: operator and operand don't agree [literal]
 operator domain: int * int list
 operand: int * int list list
 in expression:
 1 :: (1 :: nil) :: nil
```

- Liste von Listen von `int`s:

```
- [[1,2,3],[4,5],[6],[7,8,9]];
val it = [[1,2,3],[4,5],[6],[7,8,9]] : int list list
```

Polymorphe Typen: Beispiele

- Binäre Bäume mit Informationen nur an Blättern:

```
datatype 'a BinTree = Leaf of 'a
                    | Node of 'a BinTree * 'a BinTree
val t2 = Leaf "text1"
val t2 = Leaf "text1" : string BinTree
val t1 = Node(Leaf 1,Leaf 2);
val t1 = Node (Leaf 1,Leaf 2) : int BinTree
fun height t =
  case t of Leaf _ => 1
          | Node(t1,t2) => 1 + max(height t1,height t2);
val height = fn : 'a BinTree -> int
height t2;
val it = 1 : int
height t1;
val it = 2 : int
```

Polymorphe Typen: Beispiele

- Binäre Bäume mit Informationen an Blättern und inneren Knoten:

```
datatype 'a BinTree1 = Leaf1 of 'a
                     | Node1 of 'a * 'a BinTree1 * 'a BinTree1
fun height1 t =
  case t of Leaf1 _ => 1
          | Node1(_,t1,t2) => 1 + max(height t1,height t2);
val height1 = fn : 'a BinTree1 -> int
```

- Bäume beliebiger Stelligkeit:

```
datatype 'a Baum = Blatt of 'a | Knoten of 'a * 'a list;
```

Pattern-Matching: Einschränkungen

Nur Konstrukoren:

```
case "abc" of prefix^suffix => prefix;
stdIn:66.1-66.38 Error: non-constructor applied to argument in pattern:^
stdIn:66.32-66.38 Error: unbound variable or constructor: prefix
```

Höchstens eine Variable mit einem gegeben Namen in einem Pattern (Linearität):

```
fun elimDouble l =
  case l of x::x::rest => x::(elimDouble rest);
stdIn:67.20-67.64 Error: duplicate variable in pattern(s): x
```

4.6 Mehr über Variablen

Eine Variable v ist ein Paar der Form $(name, wert)$ (geschrieben auch: $name \leftarrow wert$).

4.6.1 Variablendefinitionen

- ... bestehen aus *name* und einen Ausdruck für *wert*
- heißen auch Variablen-Bindungen (*variable bindings*)
- können explizit oder implizit sein

Variablendefinitionen

- Explizite Definition:

```
val x = 1*2;
```

- Implizite Definition:
  - ▷ via Funktionsaufrufe

```
fun f x = 42
f (25*4)
```

Der Aufruf `f (25*4)` bindet `x` zu dem Wert von `25*4`.

  - ▷ via Pattern-Matching mit Variablen-Bindungen

4.6.2 Variablengültigkeit

Die Gültigkeit einer Variable (*scope*) ist die Menge der Programmpunkte, an denen ihre Definition gilt.

- Top-level Variablen (definiert mit `val name = expr` in der Interpreter-Umgebung) sind gültig an allen nachfolgenden Programmpunkten:

```
val x = 1*2;
```

Variablengültigkeit

- Parameter-Variablen (Funktionsargumente) sind gültig im Rumpf der Funtkion

```
fun f x = x+1
```

- Pattern-Variablen sind gültig in der entprechenden rechten Seite.

```
val description = case f of
                    Rot => "pure red"
                  | Blau => "pure blue"
                  | RGB(x,y,_) =>
                    if (x=y) then "kind of yellow"
                    else "something else";
```

Benutzer-definierte Gültigkeitsbereiche

... können mit Hilfe des `let`-Ausdrucks eingeführt werden:

```
let
  val name = ausdruck
in
  ausdruck'
end
```

- Der Scope der Variable *name* ist *ausdruck'*.
- Der Wert des let-Ausdrucks ist der Wert von *ausdruck'*.

```
let
   val x = 1 + 1
in
  10 * x
end
val it = 20 : int
```

Der let-Ausdruck

... kann auch den Bereich einer Funktionsdefinitionen einschränken

```
let
  fun square x = x*x
in
  square 2
end;
val it = 4 : int
- square 3;
stdIn:75.1-75.7 Error: unbound variable or constructor: square
```

Geschachtelte let-Ausdrücke

Oft möchte man geschachtelte Gültigkeitsbereiche:

```
let  val  x = 1
in  let  val  y = x+1
    in  x+y
    end
end;
val it = 3 : int
```

Äquivalent kann man schreiben:

```
let
   val  x = 1
   val  y = x+1
in  x+y
end;
val it = 3 : int
```

Der let-Ausdruck

Im Allgemeinen:

```
let
  val name_1 = ausdruck_1
  val name_2 = ausdruck_2
  ..........................
  val name_n = ausdruck_n
in
  ausdruck
end
```

- Der Scope der Variable $name_i$ besteht aus $ausdruck_{i+1}$, ..., $ausdruck_n$ und *ausdruck*.
- Der Wert des let-Ausdrucks ist der Wert von *ausdruck*.

Der let-Ausdruck: Beispiel

```
let
  val increment = 2
  fun add x = x + increment
in
  add 4
end;
val it = 6 : int
```

Statischer Gültigkeitsbereich

Static scoping Der Gültigkeitsbereich einer Variablendefinition in SML und in den meisten modernen Programmiersprachen ist definiert durch die (statische) Struktur des Programmtextes:

An welchen Programmpunkten eine Variablen-Definition (zu bestimmten Zeitpunkten) gültig ist, hängt nur von der (statischen) Struktur des Programmtextes ab.

$\Longrightarrow$ *static/lexical scoping*$\equiv$ *static/lexical variable binding*

Gültige Variablen-Definitionen am Programmpunkt •

```
let
  val separator = ";"
  fun set2String l =
    let fun doit l1 =
          case l1 of
            nil => ""
          |x::nil => Int.toString x
          |x::rest => • (Int.toString x)^separator^(doit rest)
    in "{"^(doit l)^"}" end
in set2String [1,2,3] end
val it = "{1;2;3}": string
```

Gültige Variablen-Definitionen am Programmpunkt •

```
let
  val separator = ";"
  fun set2String l =
    let fun doit l1 =
          case l1 of
            nil => ""
          |x::nil => Int.toString x
          |x::rest => (Int.toString x)^separator^(doit rest)
    in • "{"^(doit l)^"}" end
in set2String [1,2,3] end
val it = "{1;2;3}": string
```

Gültige Variablen-Definitionen am Programmpunkt •

```
let
  val separator = ";"
  fun set2String l =
    let fun doit l1 =
          case l1 of
            nil => ""
          |x::nil => Int.toString x
          |x::rest => (Int.toString x)^separator^(doit rest)
    in  "{"^(doit l)^"}" end
in • set2String [1,2,3] end
val it = "{1;2;3}": string
```

Dynamischer Gültigkeitsbereich

Dynamic scoping An welchen Programmpunkten eine Variablen-Definition gültig ist, hängt davon ab, wie diese bei der Laufzeit erreicht werden.

Dynamic Scoping: Beispiel

Scheme-Beispiel:

```
(define mult (lambda (x y) (* x y)))
(define fact (lambda (n)
                (if (= n 0) 1 (mult (fact (- n 1) n)))
              ))
(fact 3)
6

(define mult (lambda (x y) (y)))
(fact 3)
3
```

Grund: Die Bindung der top-level Variablen in Scheme ist dynamisch

$\Longrightarrow$ Die referentielle Transparenz ist verletzt

4.6.3 Variablen-Sichtbarkeit

Eine Variablen-Definition ist an einem Programmpunkt P zu einem bestimmten Zeitpunkt t sichtbar, wenn:

1. die Variablen-Definition an P zum Zeitpunkt t gültig ist, und
2. alle anderen gültigen Variablen-Definitionen mit dem selben Namen zu einem früheren Zeitpunkt stattgefunden haben.

Variablen-Sichtbarkeit: Beispiel

```
let
  val x = 1
in
  (let
     val x = 2
   in
     •P x+1
   end) + •P1 x
end
```

- Beide $x \leftarrow 1$ und $x \leftarrow 2$ sind gültig an P.
  Nur $x \leftarrow 2$ ist sichtbar an P.
- Nur $x \leftarrow 1$ ist gültig und sichtbar an P_1.

Variablen-Sichtbarkeit: Beispiel

```
fun fact n = •P if n=1 then 1
                  else n * (fact (n-1))
fact 2;
```

Sichtbare Variablen am P:

Zeit	Variablen-Definition (implizit)	gültig	sichtbar
t_1:	`(fact 2)`	$n \leftarrow 2$	$n \leftarrow 2$
t_2:	`(fact 1)`	$n \leftarrow 2$, $n \leftarrow 1$	$n \leftarrow 1$

4.6.4 Kontext

Der Kontext eines Programmpunkts P zu einem bestimmten Zeitpunkt t besteht aus der Menge der Variablen-Definitionen die sichtbar am P zum Zeitpunkt t sind.
$\Longrightarrow$ ist ein dynamischer Konzept: Dem selben Programmpunkt können bei der Laufzeit verschiedene Kontexte zu verschiedenen Zeitpunkten entsprechen.

$$Kontext(P)_t = \{n \leftarrow 1\}$$

$$Kontext(P)_{t+\Delta t} = \{n \leftarrow 2\}$$

Kontext: Beispiel

```
fun fact n = •P if n=1 then 1
                  else n * (fact (n−1))
```

Kontext von P:

Zeit	Kontext
t_1: `(fact 2)`	$n \leftarrow 2$
t_2: `(fact 1)`	$n \leftarrow 1$

4.7 Mehr über Funktionen

4.7.1 Der Typ-Operator für Funktionstypen

$$-> : MT \times MT \mapsto MT$$

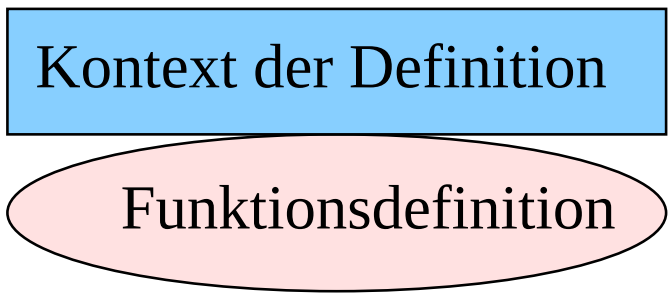
$$\boxed{\alpha \; -> \; \beta} = \{f : \alpha \mapsto \beta \mid \alpha, \beta \in MT\}$$

Der Typ-Operator $->$ ist rechtsassoziativ:

$$\alpha \mapsto \beta \mapsto \gamma \equiv \alpha \mapsto \beta \mapsto \gamma$$

4.7.2 Funktionale Abschlüsse

Eine Funktion besteht aus der Funktionsdefinition und der Kontext des Programmpunktes an welchen die Funktion definiert (≡ konstruiert) wird.



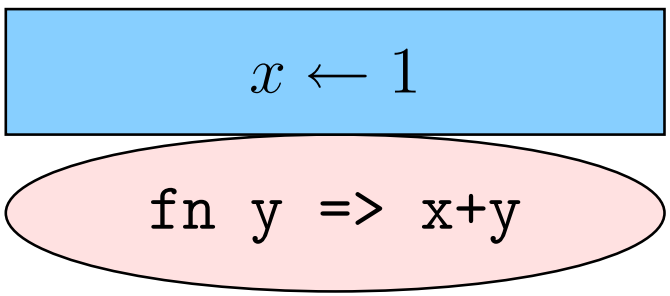
Man sagt, dass Funktionen ihren Kontext zu dem Zeitpunkt ihrer Definition abschließen.

Eine Funktion wird auch funktionaler Abschluss (*closure*) genannt.

Funktionaler Abschluss: Beispiel

```
val x = 1
• val f = fn y => x + y
val v1 = f 3;
val v1 = 4 : int

val x =2
val v2 = f 3;
val v2 = 4 : int
```



4.7.3 Currying

- Currying = Methode mit der man Funktionen von mehreren Argumenten konstruieren kann.
- genannnt nach dem Erfinder, Haskell B. Curry

```
val sum = fn x => (fn y => x+y);
val sum = fn : int -> int -> int
```

`sum` erwartet ein `x` Argument und liefert eine Funktion zurück, die wiederum ein Argument `y` erwartet und `x+y` zurückliefert.

Wegen der Rechtsassoziativität von `=>` kann man auch schreiben:

```
val sum = fn x => fn y => x+y;
val sum = fn : int -> int -> int
```

Curry-Funktionen (*curried functions*)

Funktionsanwendung (Aufruf):

```
val sum = fn x => fn y => x+y;
(sum 2) 3;
val it = 5 : int
```

Die Funktionsanwendung ist linksassoziativ:

`f x1 x2 x3` $\equiv$ `(((f x1) x2) x3)`.

Deshalb kann man auch schreiben:

```
sum 2 3;
val it = 5 : int
```

Verkürzte Syntax

Statt:

```
val sum = fn x => fn y => x+y;
val sum = fn : int -> int -> int
```

kann man mit verkürzter Syntax die Funktion `sum` so definieren:

```
fun sum x y = x+y;
val sum = fn : int -> int -> int
```

I.a. ist:

```
fun f x1 x2 ... xn = expr
```

das selbe wie:

```
val f = fn x1 => fn x2 => ... fn xn => expr
```