Aufgabe 1: Summe von Termen (8)

```
fun sum (T(x,ts)) = foldl (fn (t,s) => sum t + s) x ts
```

Aufgabe 2: Sortieren und Doppelauftreten (8+8+8+8)

```
fun split xs = foldl (fn (x, (ys,zs)) => (zs, x::ys))
                     (nil, nil) xs

fun merge (nil  , ys   ) = ys
  | merge (xs   , nil  ) = xs
  | merge (x::xr, y::yr) = if x<=y then x::merge(xr,y::yr)
                           else y::merge(x::xr,yr)
fun sort nil      = nil
  | sort (x::nil) = x::nil
  | sort xs       = let
                      val (ys,zs) = split xs
                    in
                      merge(sort ys, sort zs)
                    end

fun elim' (x::y::zs) = if x=y then elim'(y::zs)
                       else x::elim'(y::zs)
  | elim' zs         = zs

fun elim xs = elim'(sort xs)
```

Aufgabe 3: Funktor für endliche Mengen mit Suchbäumen (3+1+8+8)

```
functor Set
  (type elem
   val  compare : elem * elem -> order)
  :>
  sig
    type set
    val empty  : set
    val insert : elem * set -> set
    val member : elem * set -> bool
  end
  =
  struct
    datatype set = E | N of set * elem * set

    val  empty = E

    fun member (y, E)        = false
      | member (y, N(a,x,b)) = case compare(y,x) of
                                 LESS    => member(y,a)
                               | EQUAL   => true
                               | GREATER => member(y,b)

    fun insert (y, E)        = N(E,y,E)
      | insert (y, N(a,x,b)) = case compare(y,x) of
                                 LESS    => N(insert(y,a), x, b)
                               | EQUAL   => N(a,y,b)
                               | GREATER => N(a, x, insert(y,b))
  end
```

Aufgabe 4: Statische Semantik (10+6+10)

```
fun closed' xs (Id y)       = List.exists (fn x => x=y) xs
  | closed' xs (Abs(x,_,e)) = closed' (x::xs) e
  | closed' xs (App(e,e'))  = closed' xs e andalso closed' xs e'

fun closed e = closed' nil e

fun adjoin env x t y = if x=y then t else env y

fun elab env (Id y)       = env y
  | elab env (Abs(x,t,e)) = Arrow(t, elab (adjoin env x t) e)
  | elab env (App(e,e'))  = (case elab env e of
                               Arrow(t,t') => if elab env e' = t then t'
                                              else raise Error
                             | _ => raise Error)
```

Aufgabe 5: Kontextfreie Syntax von applikativen Ausdrücken (5+10+6+15)

(a)

$$exp = [\ exp\ ]\ atexp$$
$$atexp = identifier \mid \texttt{"("}\ exp\ \texttt{")"}$$

(b)
```
fun exp (App(e1, e2)) = exp e1 ^ " " ^ atexp e2
 |  exp e             = atexp e
and atexp (Id s)      = s
 |  atexp e           = "(" ^ exp e ^ ")"
```

(c)

$$exp = atexp\ \ exp'$$
$$exp' = [\ atexp\ \ exp'\ ]$$
$$atexp = identifier \mid \texttt{"("}\ exp\ \texttt{")"}$$

(d)
```
fun atexp (ID _::ts) = ts
 |  atexp (LPAR::ts) = (case exp ts of RPAR::tr => tr | _ => raise Error)
 |  atexp _          = raise Error
and exp ts           = exp' (atexp ts)
and exp' (ID _::tr)  = exp' tr
 |  exp' (LPAR::tr)  = exp'(atexp (LPAR::tr))
 |  exp' tr          = tr
and test ts = (exp ts = nil) handle Error => false
```

Aufgabe 6: Übersetzung und Rückübersetzung (8+12)

```
fun compile'(Con i)      = [con i]
  | compile'(Add(e1,e2)) = compile' e2 @ compile' e1 @ [add]
  | compile'(Mul(e1,e2)) = compile' e2 @ compile' e1 @ [mul]

fun compile e = compile' e @ [halt]

fun decompile' (con n::is, es)      = decompile'(is, Con n::es)
  | decompile' (add::is, e::e'::es) = decompile'(is, Add(e,e')::es)
  | decompile' (mul::is, e::e'::es) = decompile'(is, Mul(e,e')::es)
  | decompile' ([halt], [e])        = e
  | decompile'     _                = raise Error "cannot decompile"

fun decompile code = decompile'(code,nil)
```

Aufgabe 7: V-Programmierung (10+8+10)

(a)
```
[proc(2, 21),
 getF ~2, getF ~1, sub, cbranch 15,
 getF ~2, getF ~1, leq, cbranch 6,
 getF ~1, getF ~2, sub, getF ~1, callR 0, (* gcd(x, y-x) *)
 getF ~2, getF ~2, getF ~1, sub, callR 0, (* gcd(x-y, y) *)
 getF ~1, return (* x zurückliefern *)
]
```

(b)
```
fun gcdi (x0, y0) =
    let
        val x = ref x0
        val y = ref y0
    in
        while !x <> !y do
          if !x <= !y then y:= !y - !x else x:= !x - !y ;
        !x
    end
```

(c)
```
[getS 1, getS 0, sub, cbranch 15,          (* x <> y ? *)
 getS 1, getS 0, leq, cbranch 6,           (* ja, x <= y?  *)
 getS 0, getS 1, sub, putS 1, branch ~12, (* ja, y := !y - !x und Schleife *)
 getS 1, getS 0, sub, putS 0, branch ~17, (* nein, x := !x - !y und Schleife *)
 getS 0
]
```

Aufgabe 8: Imperative Listen (10)

```
fun circle’ r k n =
  Cons(k, if k>=n then r else ref(circle’ r (k+1) n))


fun circle n = let
                 val r = ref Nil
                 val f = circle’ r 1 n
               in
                 r:=f ; f
               end
```