

---

# III. Funktionale Programmierung

- 1. Prinzipien der funktionalen Programmierung
- 2. Deklarationen
- 3. Ausdrücke
- 4. Muster (Patterns)
- 5. Typen und Datenstrukturen
- 6. Funktionale Programmiertechniken



# Parametrische Polymorphie

```
len :: [Bool] -> Int
len []      = 0
len (x : xs) = 1 + len xs
```

```
len :: [Int] -> Int
len []      = 0
len (x : xs) = 1 + len xs
```

stattdessen



```
len :: [a] -> Int
len []      = 0
len (x : xs) = 1 + len xs
```

```
ident :: a -> a
ident x = x
```

```
app :: [a] -> [a] -> [a]
app []      ys = ys
app (x : xs) ys = x : app xs ys
```

Diagram illustrating the components and flow of a parser for a language with types and data constructs.

The components shown are:

- topdecl**: The initial input or state.
- data**: A terminal symbol (represented by a pink oval).
- tyconstr**: A non-terminal symbol (represented by a yellow rectangle).
- var**: A non-terminal symbol (represented by a yellow rectangle).
- =**: A terminal symbol (represented by a pink circle).
- constr**: A non-terminal symbol (represented by a yellow rectangle).
- type**: A non-terminal symbol (represented by a yellow rectangle).

The flow of the parser is indicated by arrows:

- topdecl** leads to **tyconstr**.
- tyconstr** leads to **data**.
- tyconstr** leads to **var**.
- var** leads to **=**.
- =** leads to **constr**.
- constr** leads to **type**.
- type** leads to the final output arrow.
- There are feedback loops from **var** back to **tyconstr** and from **type** back to **constr**, indicating recursive calls.
- A long arrow from **decl** (which is reached from **topdecl**) goes directly to the final output arrow.

Handwritten blue text at the bottom identifies the components:

- Tykonstruktor** (Type constructor) points to **tyconstr**.
- Datenkonstruktoeren** (Data constructors) points to **constr**.

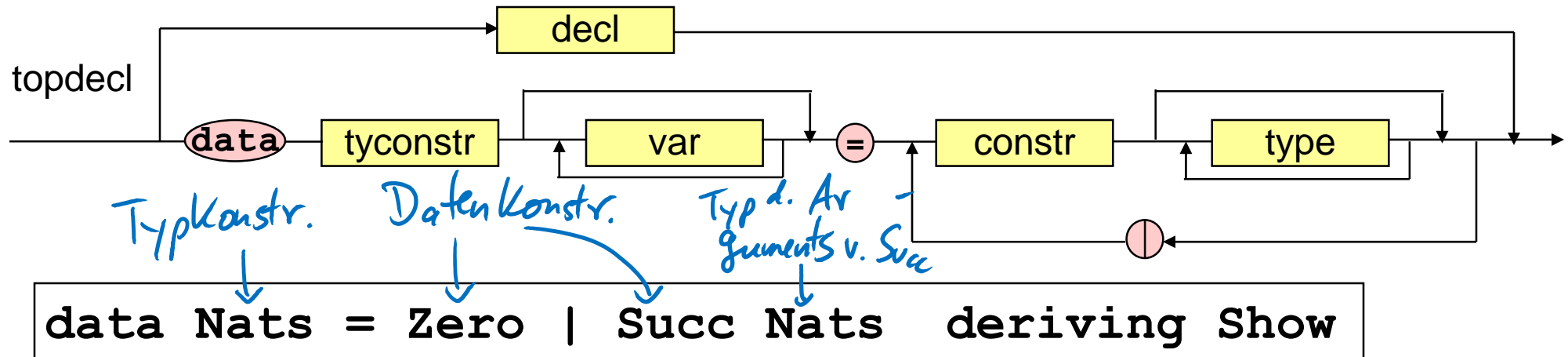
```

ampel :: Color -> Color
ampel Red      = Green
ampel Green    = Yellow
ampel _        = Red

```

III.5 Typen und Datenstrukturen - 4 -

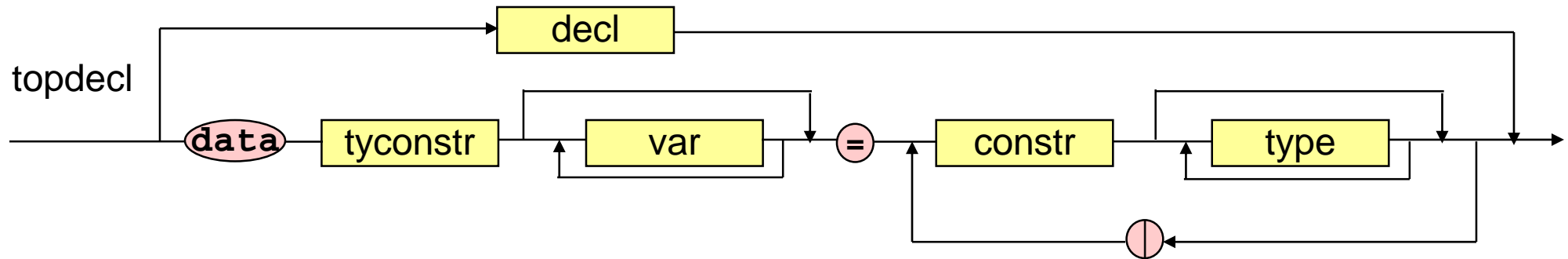
# Deklaration neuer Datentypen



```
plus :: Nats -> Nats -> Nats
plus Zero      y      = y
plus (Succ x)  y      = Succ (plus x y)
```

```
half :: Nats -> Nats
half Zero = Zero
half (Succ Zero) = Zero
half (Succ (Succ x)) = Succ (half x)
```

# Deklaration neuer Datentypen



```
data List a = Nil | Cons a (List a) deriving Show
```

```
len :: List a -> Int
len Nil           = 0
len (Cons x xs)   = 1 + len xs
```

```
app :: List a -> List a -> List a
app Nil          ys   = ys
app (Cons x xs)  ys   = Cons x (app xs ys)
```