III. Funktionale Programmierung

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

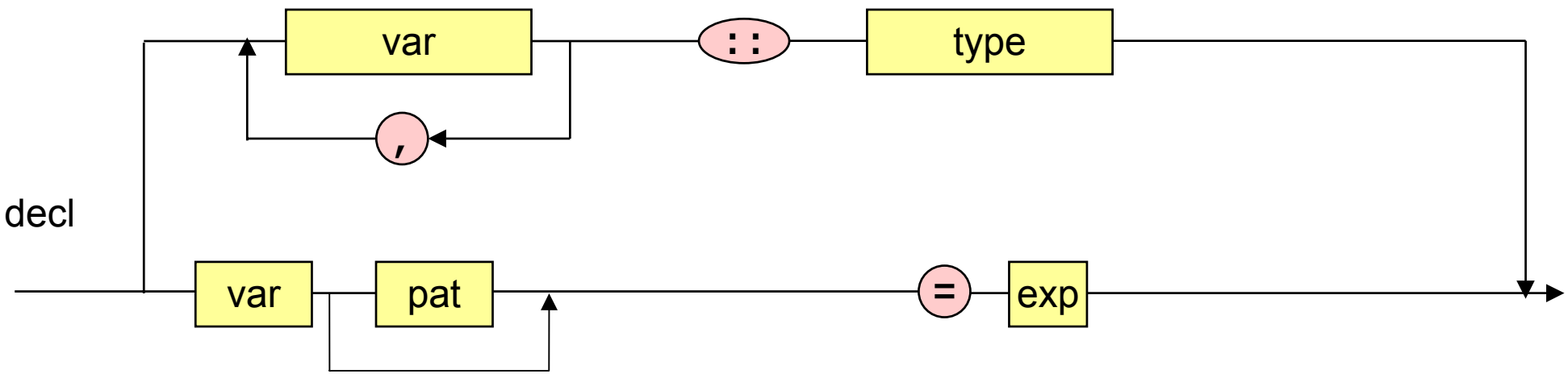
Deklarationen

```
len :: [Int] -> Int
len []                  =    0
len (kopf : rest)       =    1 + len rest

square :: Int -> Int
square x = x * x
```

Programm in *Haskell*: Folge von linksbündig untereinander stehenden Deklarationen

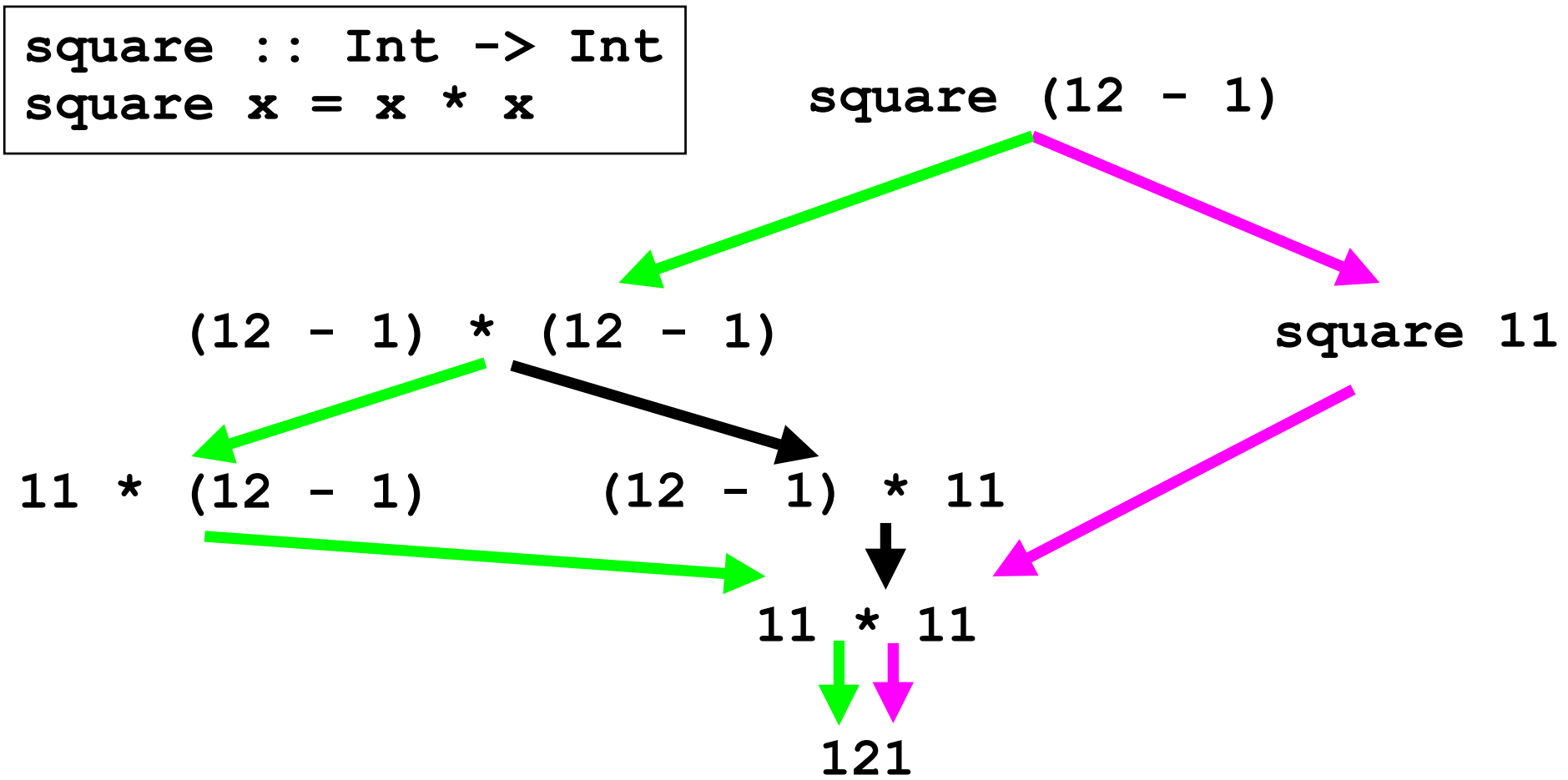
Deklarationen

```
len :: [Int] -> Int
len []                  =    0
len (kopf : rest)       =    1 + len rest

square :: Int -> Int
square x = x * x
```



Auswertungsstrategie

```
square :: Int -> Int
square x = x * x
```

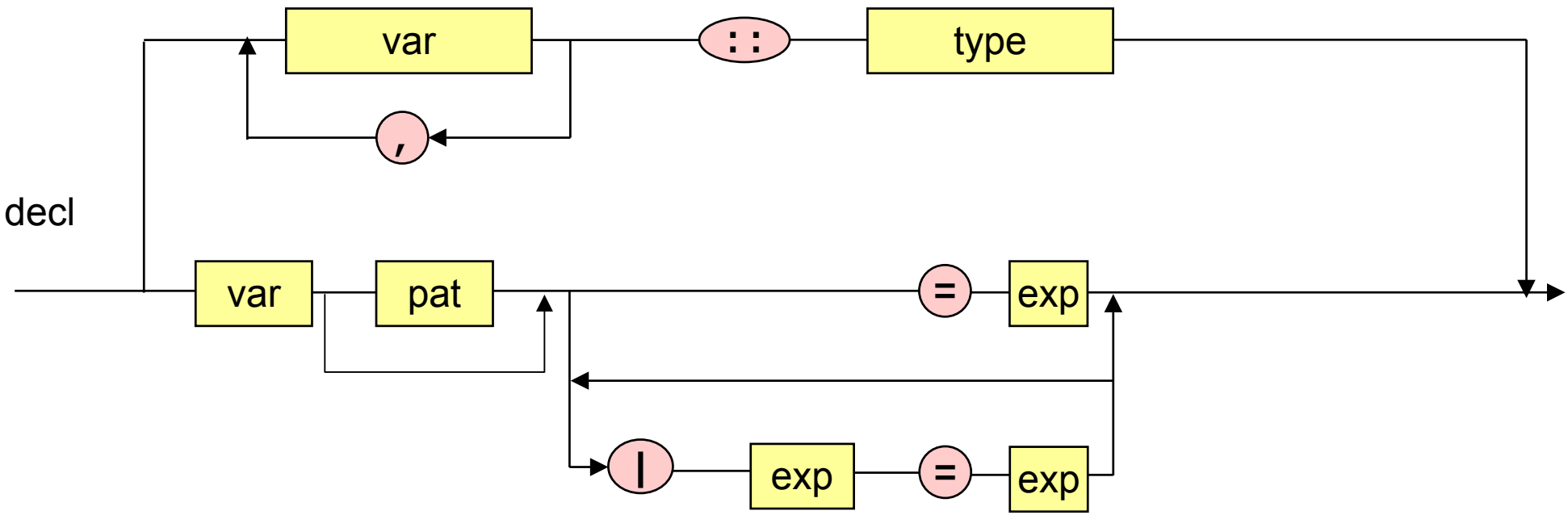


→ strikte Auswertung (call-by-value), innen links

→ nicht-strikte Auswertg. (call-by-name), außen links

Bedingte definierende Gleichungen

```
maxi :: (Int, Int) -> Int

maxi (x, y)  | x >= y      = x
             | otherwise   = y
```

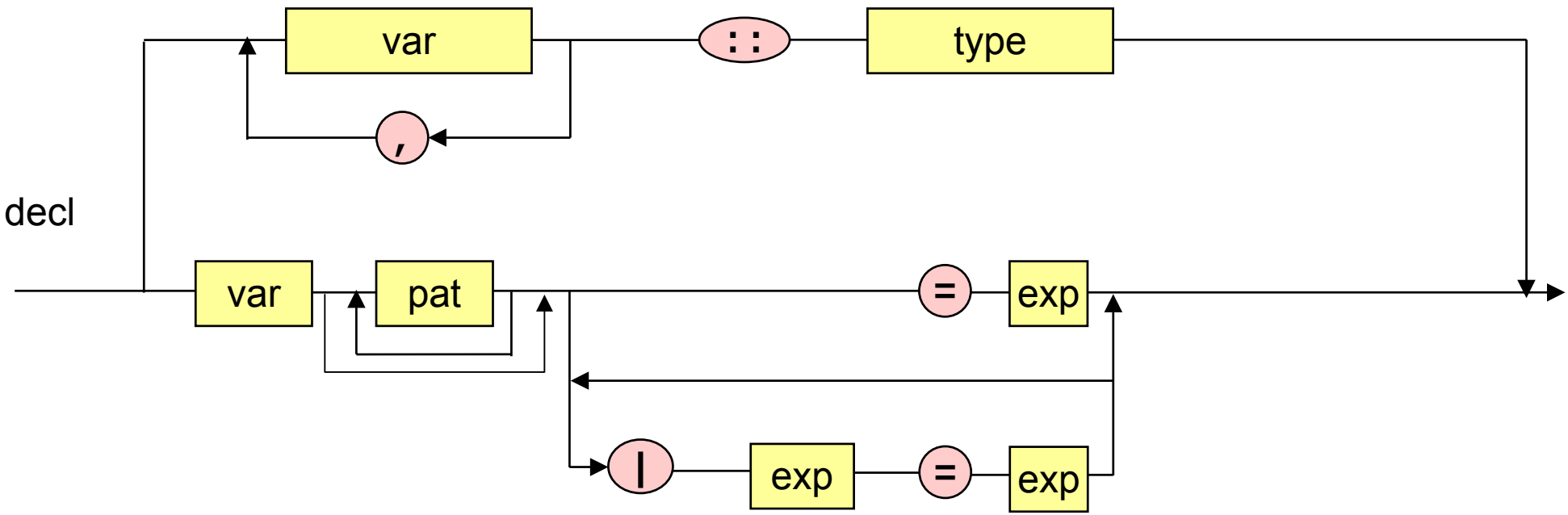


Currying

```
plus :: (Int, Int) -> Int
plus (x, y) = x + y
```

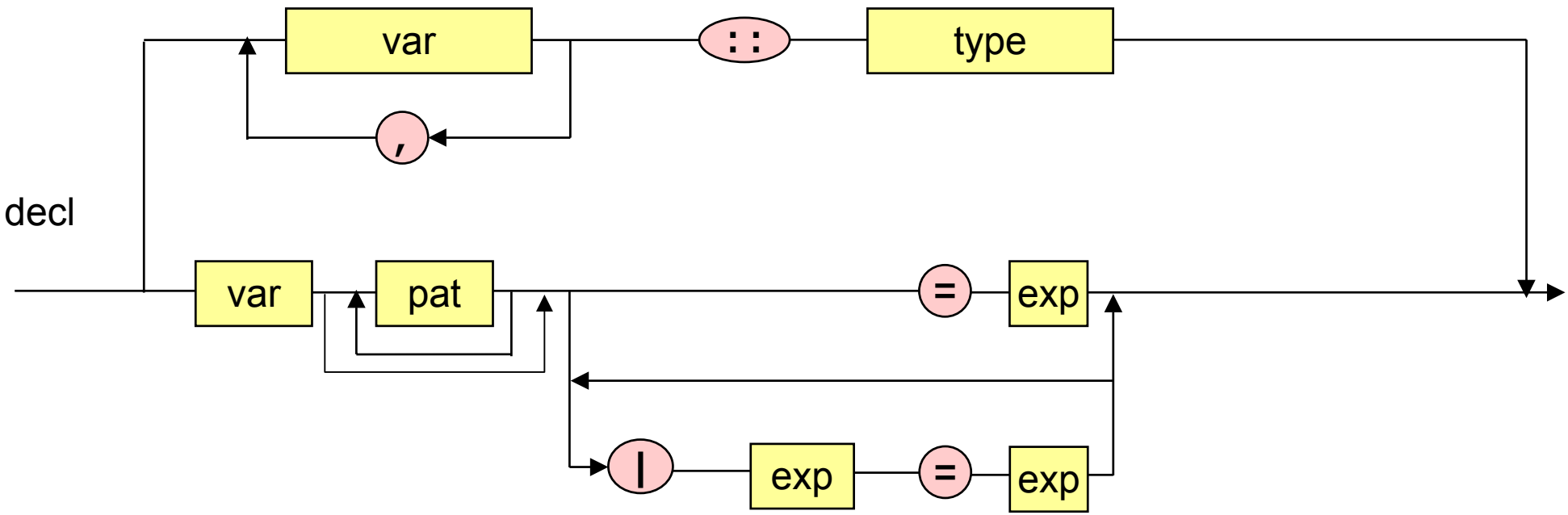
Currying

```
plus :: Int -> Int -> Int
plus x y = x + y
```



Pattern Matching

```
und :: Bool -> Bool -> Bool
und True  y  = y
und   x   y  = False
```



Pattern Matching

```
und :: Bool -> Bool -> Bool
und True  y  = y
und   x   y  = False
```

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

```
fac :: Int -> Int
fac 0        = 1
fac (x + 1) = (x+1) * fac x
```

```
half :: Int -> Int
half 0       = 0
half 1       = 0
half (x + 2) = 1 + half x
```

```
Bool = "True" | "False"
```

```
Liste = "[]" |
        Element ":" Liste
```

```
Int =  NInt | PInt

PInt = "0" | PInt "+ 1"
```

Lokale Deklarationen

```
roots :: Float -> Float -> Float -> (Float, Float)

roots a b c = ((-b - d)/e, (-b + d)/e)
                where    d = sqrt b*b - 4*a*c
                         e = 2*a
```