

## Small step Semantik

Die simultan rekursiven Funktionen *even* und *odd* mit der fun-Schreibweise

```
let fun even (x:int) : bool = x = 0 orelse odd (x - 1)
    and odd (x:int) : bool = not (x = 0) andalso even (x - 1)
in even 1
end
```

steht für

```
let val (even, odd) = rec_tuple_2 : (int → bool) * (int → bool).
    (λ x:int. = (x, 0) orelse #2 _tuple_2 (- (x, 1)),
     λ x:int. not (= (x, 0)) andalso #1 _tuple_2 (- (x, 1)))
in even 1
end
```

→ let val (even, odd) = (λ x:int. = (x, 0) orelse #2 (rec\_tuple\_2 ... ) (- (x, 1)),  
λ x:int. not (= (x, 0)) andalso #1 (rec\_tuple\_2 ... ) (- (x, 1)))  
in even 1  
end

mit Regel (UNFOLD)

→ (λ x:int. = (x, 0) orelse #2 (rec\_tuple\_2 ... ) (- (x, 1))) 1  
mit der abgeleiteten Regel (LET-EXEC-n)

→ = (1, 0) orelse #2 (rec\_tuple\_2 ... ) (- (1, 1))  
mit Regel (BETA-V)

→ false orelse #2 (rec\_tuple\_2 ... ) (- (1, 1))  
mit Regel (OP)

→ #2 (rec\_tuple\_2 ... ) (- (1, 1))  
mit der abgeleiteten Regel (ORELSE-FALSE)

→ #2 (λ x:int. = (x, 0) orelse #2 (rec\_tuple\_2 ... ) (- (x, 1)),  
λ x:int. not (= (x, 0)) andalso #1 (rec\_tuple\_2 ... ) (- (x, 1)))  
(- (1, 1))  
mit Regel (UNFOLD)

→ (λ x:int. not (= (x, 0)) andalso #1 (rec\_tuple\_2 ... ) (- (x, 1))) (- (1, 1))  
mit Regel (PROJ)

→ (λ x:int. not (= (x, 0)) andalso #1 (rec\_tuple\_2 ... ) (- (x, 1))) 0  
mit Regel (OP)

→ not (= (0, 0)) andalso #1 (rec\_tuple\_2 ... ) (- (0, 1))  
mit Regel (BETA-V)

→ not true andalso #1 (rec\_tuple\_2 ... ) (- (0, 1))  
mit Regel (OP)

→ false andalso #1 (rec\_tuple\_2 ... ) (- (0, 1))  
mit Regel (OP)

→ false  
mit der abgeleiteten Regel (ANDALSO-FALSE)