# The Little Schemer:

Building an interpreter one feature at a time

Leon Roth

02 June 2022

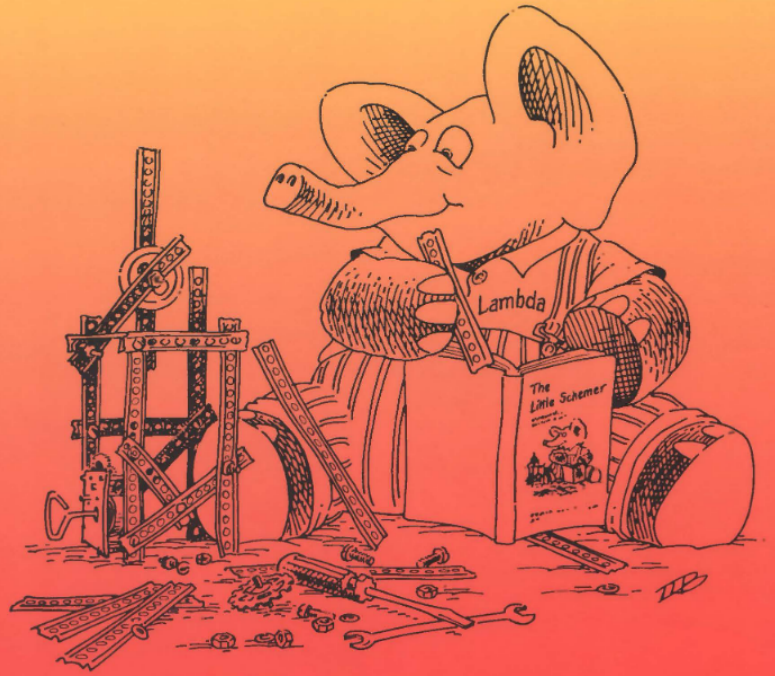
# Zu mir

# Meine Motivation

Die Magie hinter Sprachen zu durchbrechen

# The Little Schemer

Fourth Edition




Daniel P. Friedman and Matthias Felleisen

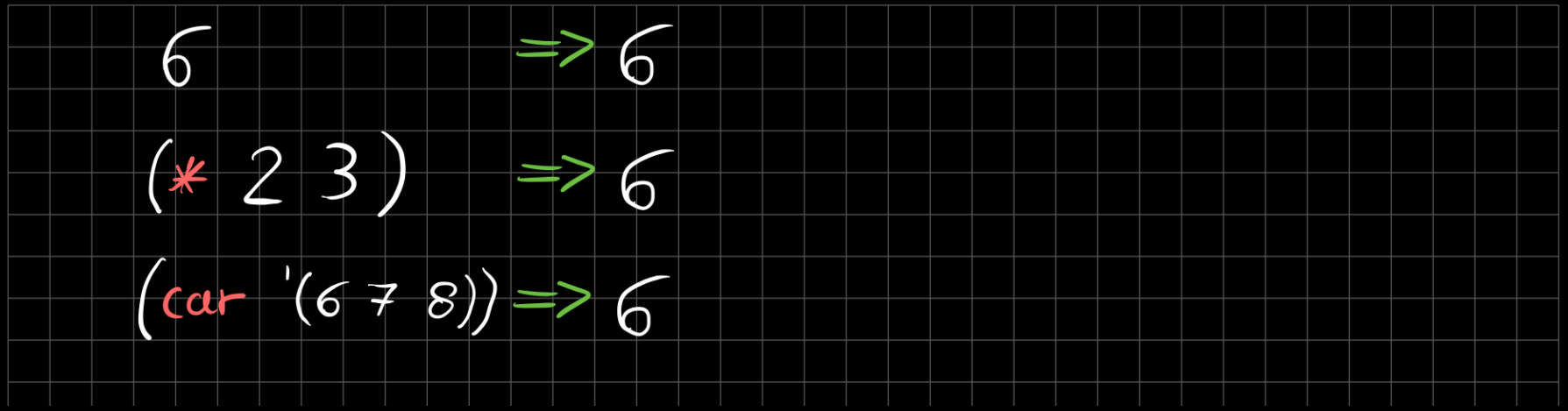
Foreword by Gerald J. Sussman

# Little Scheme

- Subset von Scheme
  - Funktionale Sprache
  - Dynamische Typen
  - Higher-order Fuctions
  - Anonymous Functions
  - Immutable

```
(define factorial
  (lambda (n)
    (cond
      ((eq? n 0) 1)
      (else (* n (factorial (- n 1))))
    )
  )
)
```

```
fn factorial(n: u32) -> u32 {
  if n == 0 {
    1
  } else {
    n * factorial(n - 1)
  }
}
```


6 ⇒ 6
(* 2 3) ⇒ 6
(car '(6 7 8)) ⇒ 6

No answer.[1]

You cannot ask for the *cdr* of the null list.

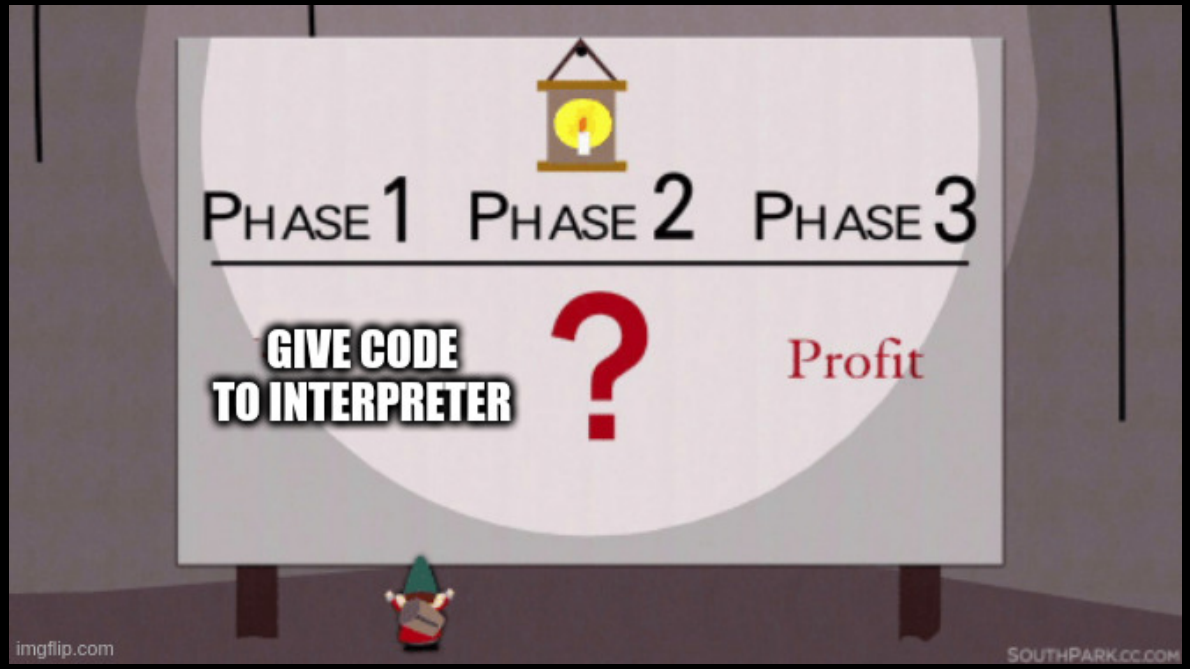
---

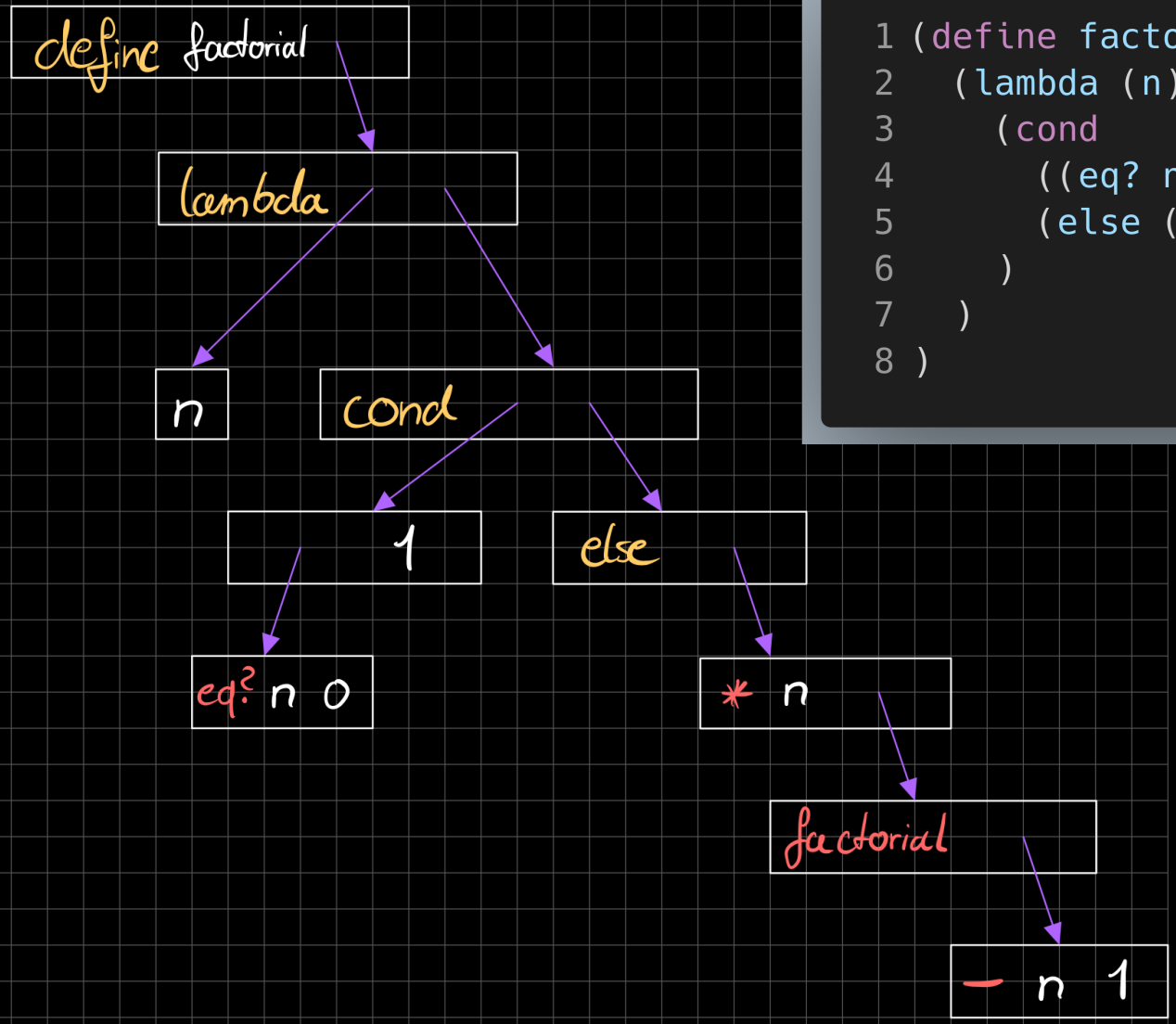
[1] L: `nil`

# Meine Arbeit mit dem Buch

- Ohne grosse Literatur daraus einen Interpreter bauen
  - Kapitel fuer Kapitel
  - Kaum Implementierungsdetails im Buch
  - Spaeter offiziele Scheme Doku dazugezogen um Genauigkeiten zu finden
- Erstes mal eine Programmiersprache “machen”

# Interpreter






```scheme
(define factorial
  (lambda (n)
    (cond
      ((eq? n 0) 1)
      (else (* n (factorial (- n 1)))))
    )
  )
)
```

# Parser

```
1 ⚠️ (car ('first 'second 'third)) => ('first)
2
3 ✅ (car ('first 'second 'third)) => 'first
```

# Tokenizer

```
1 "Hello Scheme!" => String
2
3 print_args => Symbol
4
5 42 => Integer
6
7 define => Keyword
```

# Quoting in Scheme

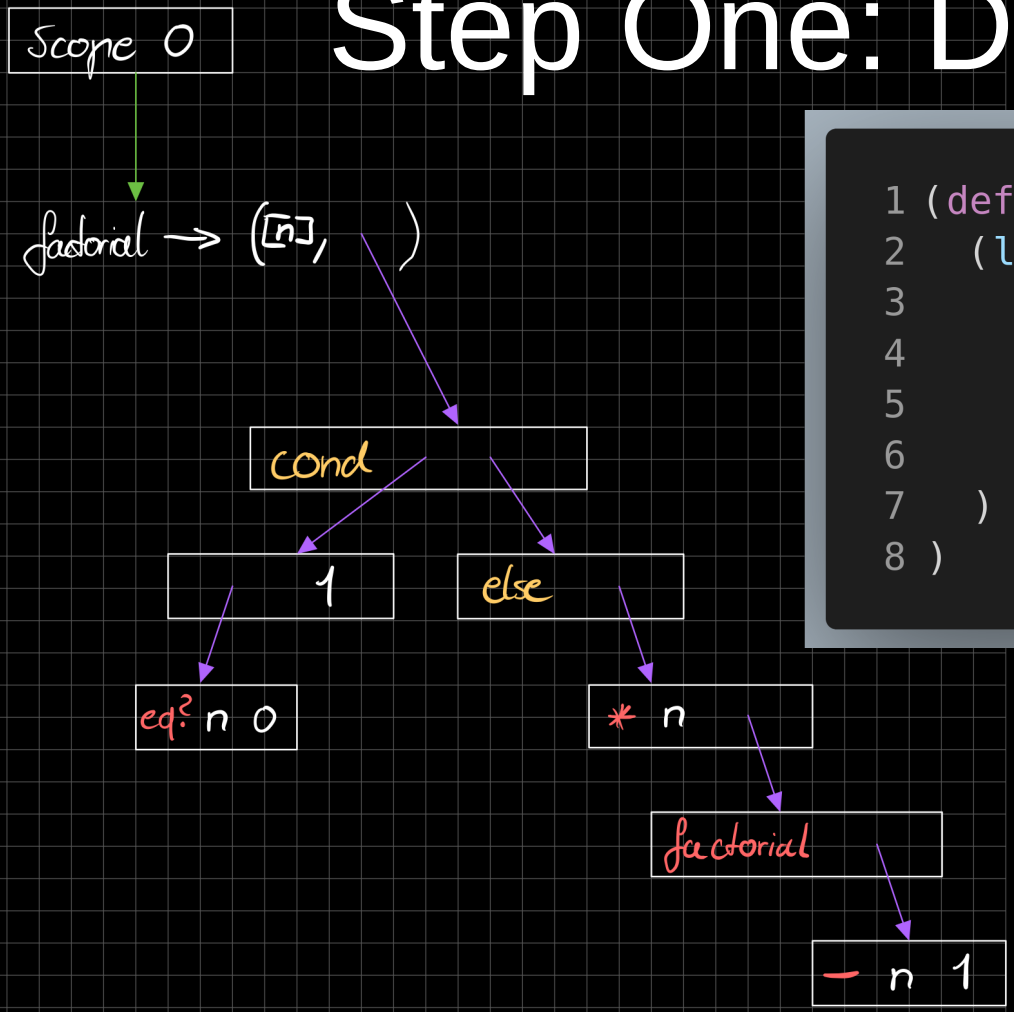
# Interpretieren

- Laeuft den geparseten Baum durch
  - Erst kinder aufrufen und deren Ergebnis nutzen → tiefster zuerst
- Ein Paar wenige Base-Types die alles repraesentieren
  - Integer
  - String
  - Symbol
  - Boolean
  - Keywords
  - List
  - Function

# Scope

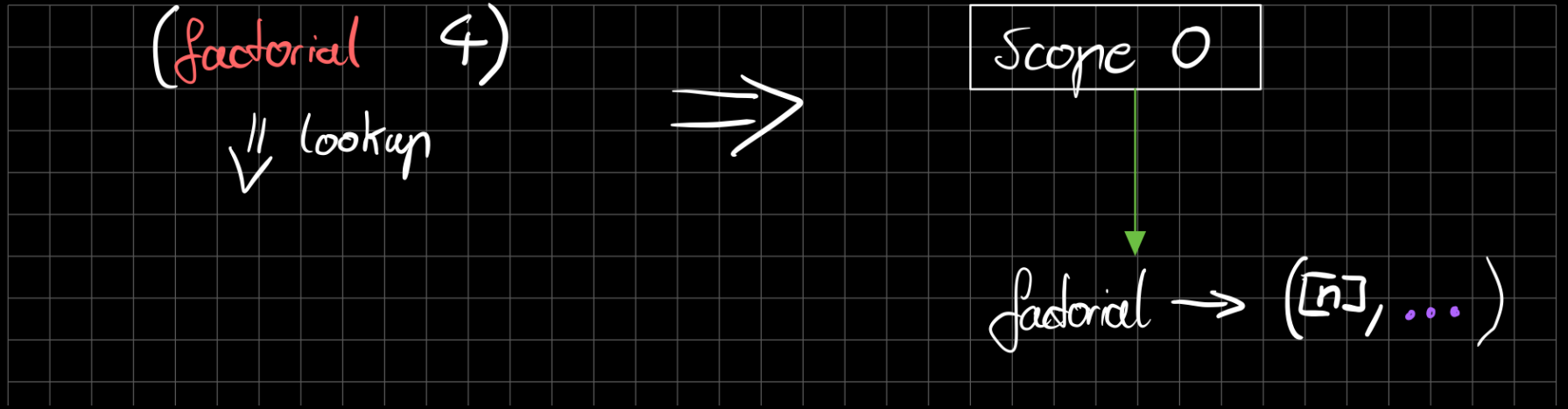
- Fuer Let und Lambda (eigene Funktionen)
- Linked List von Hashmaps
- Wie Genau Funktionieren Lambdas (Closures)

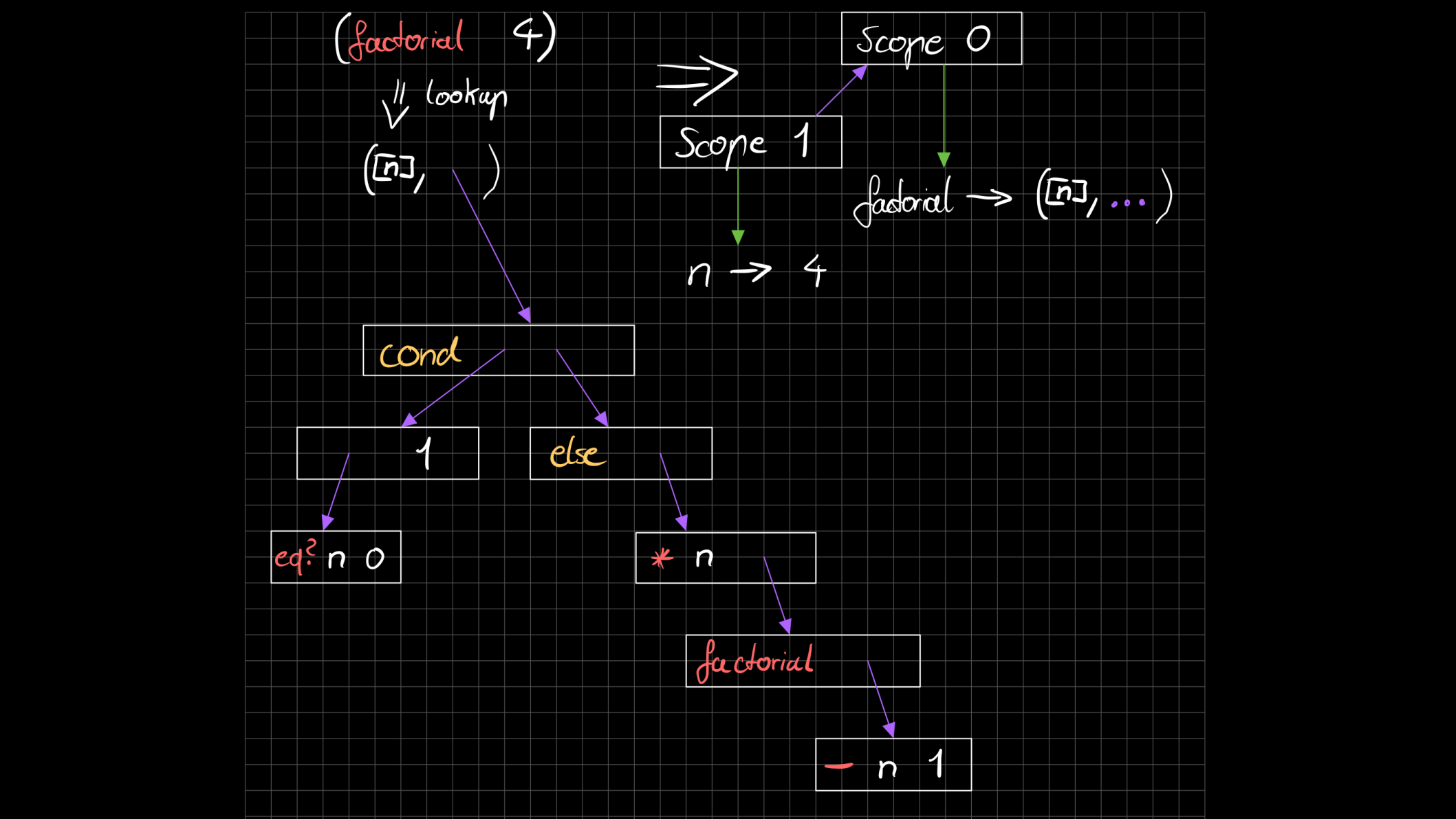
# Step One: Define

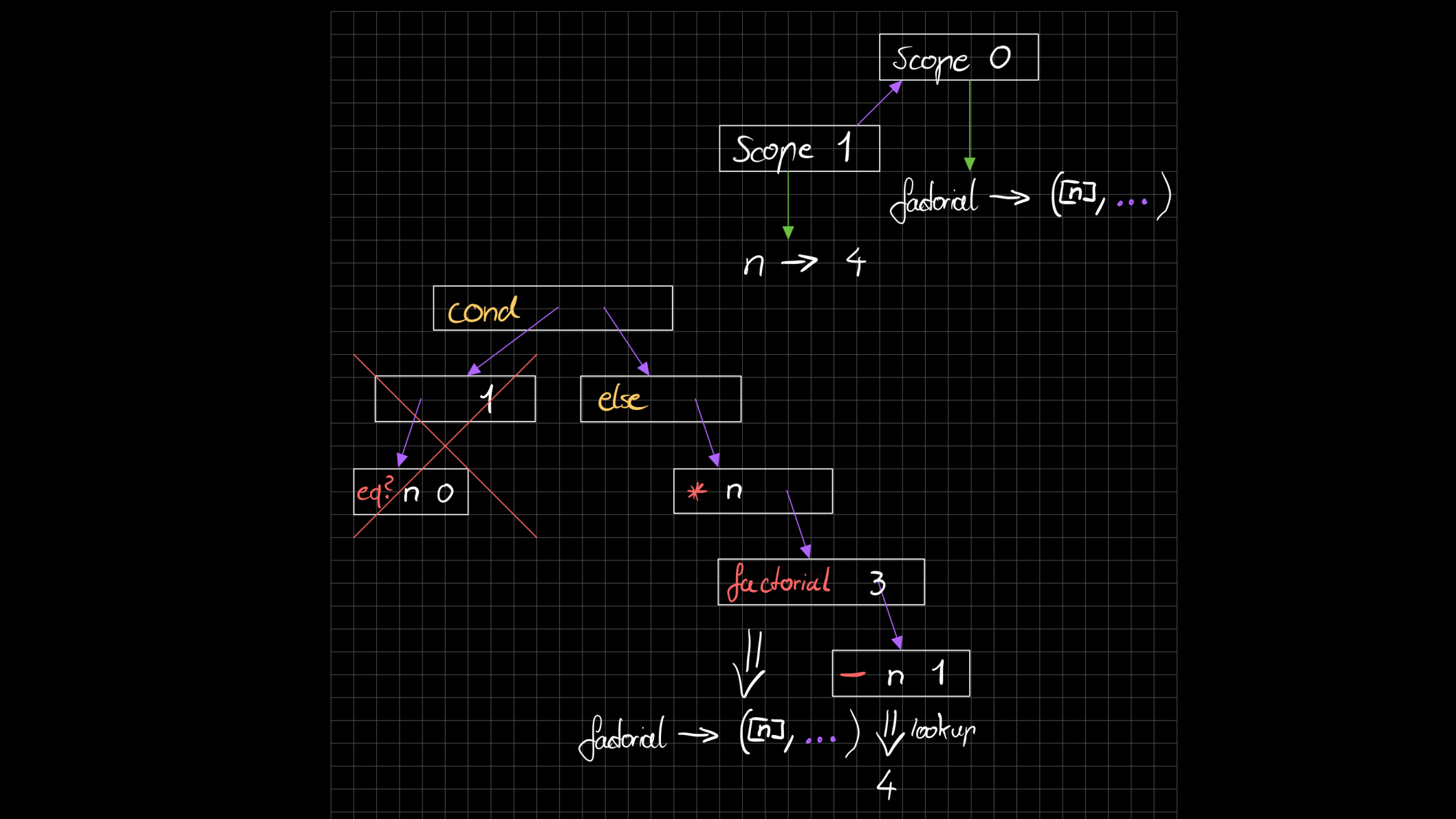


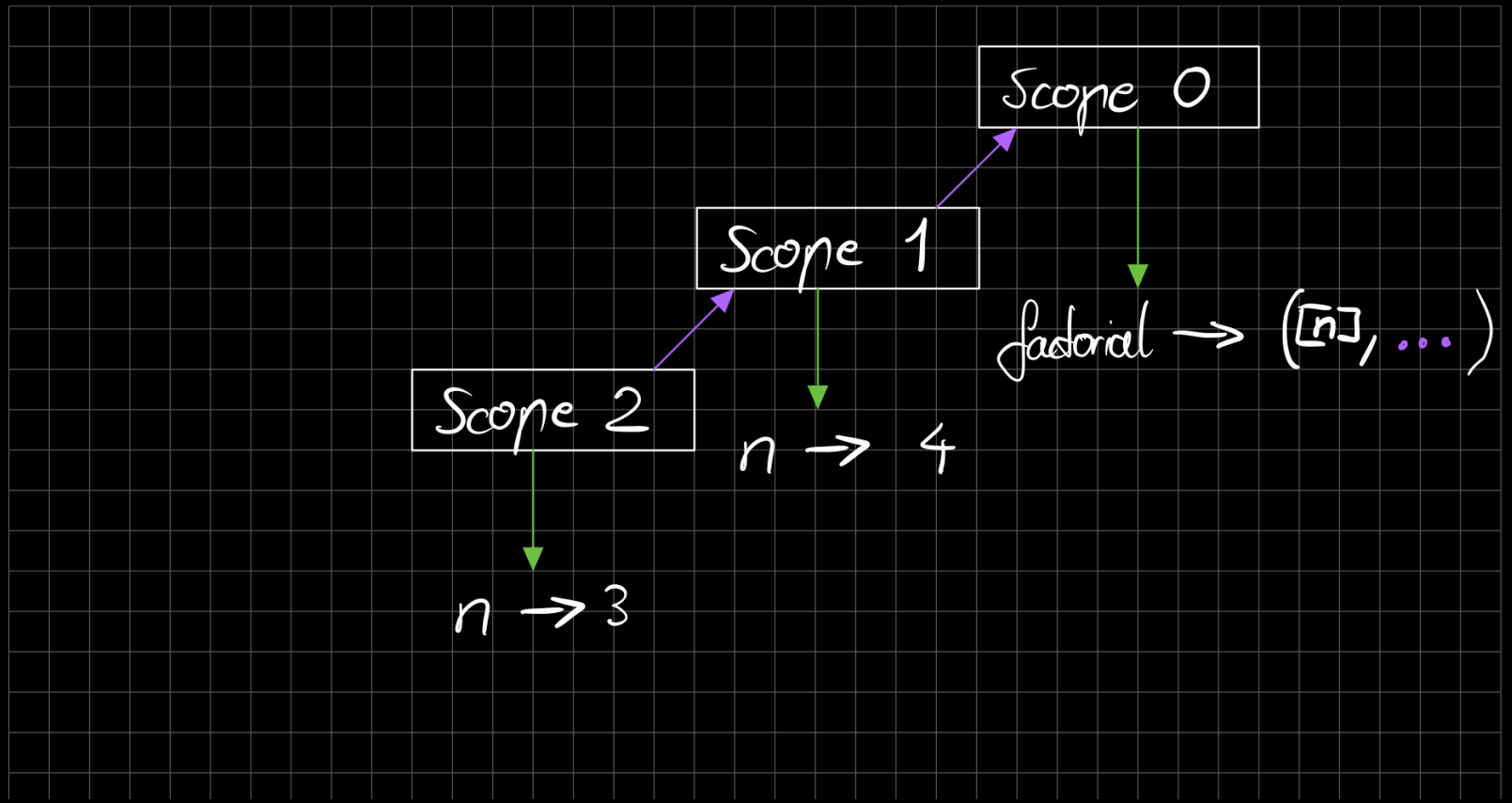

```scheme
(define factorial
  (lambda (n)
    (cond
      ((eq? n 0) 1)
      (else (* n (factorial (- n 1))))
    )
  )
)
```

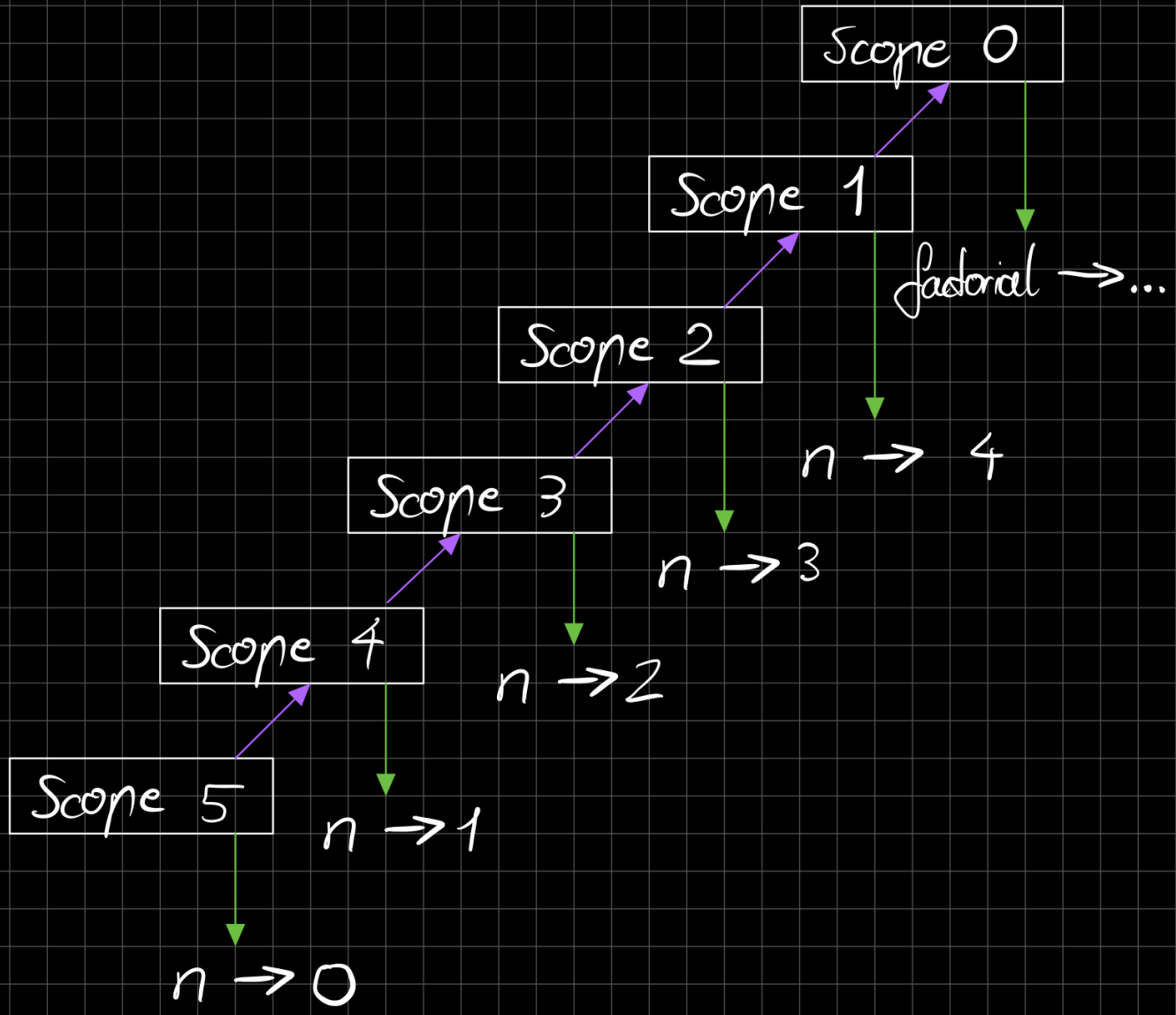
# Step Two: Call

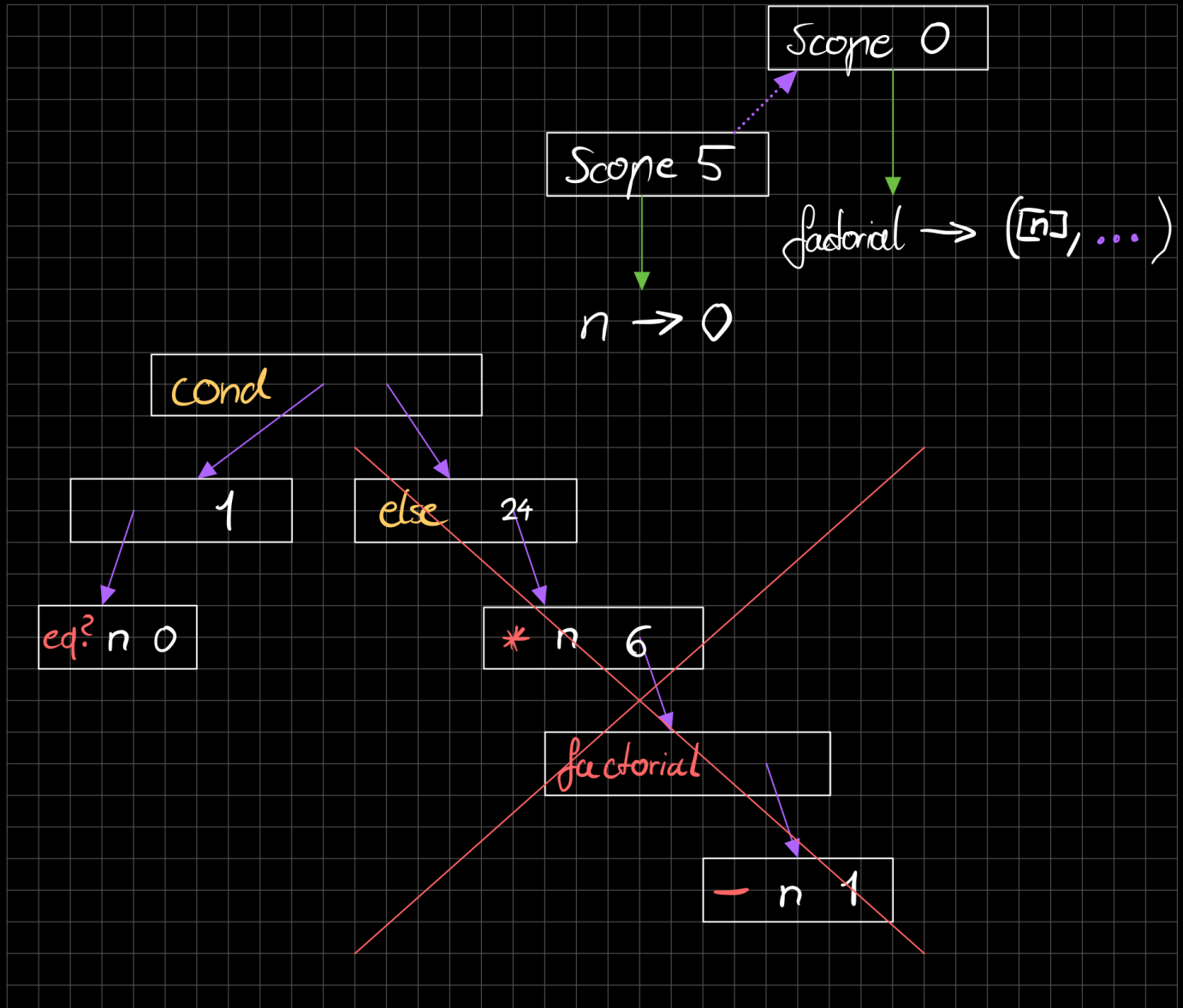


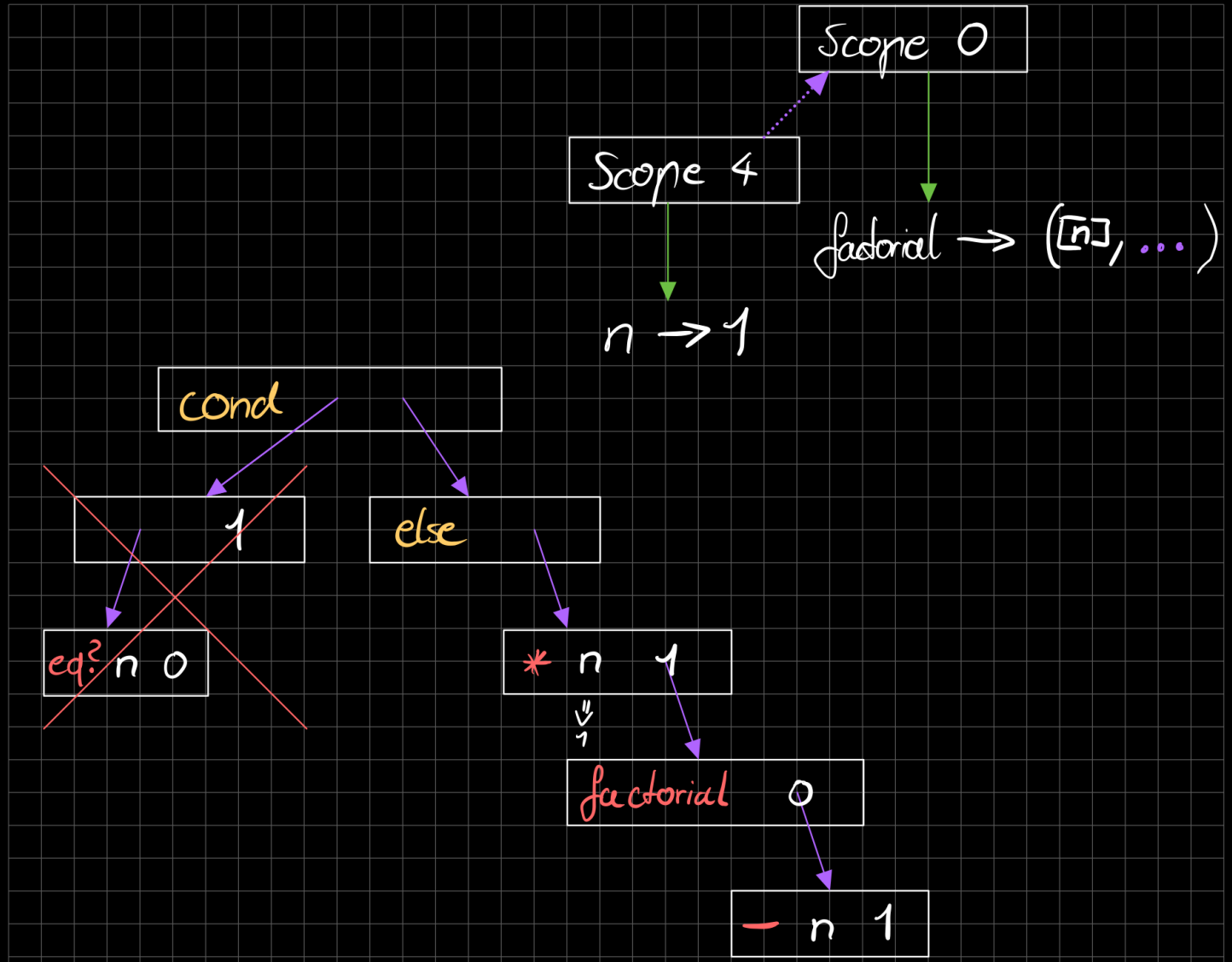

(factorial 4)
lookup
([n], )
Scope 0
Scope 1
factorial → ([n], ...)
n → 4
cond
1
else
eq? n 0
* n
factorial
- n 1

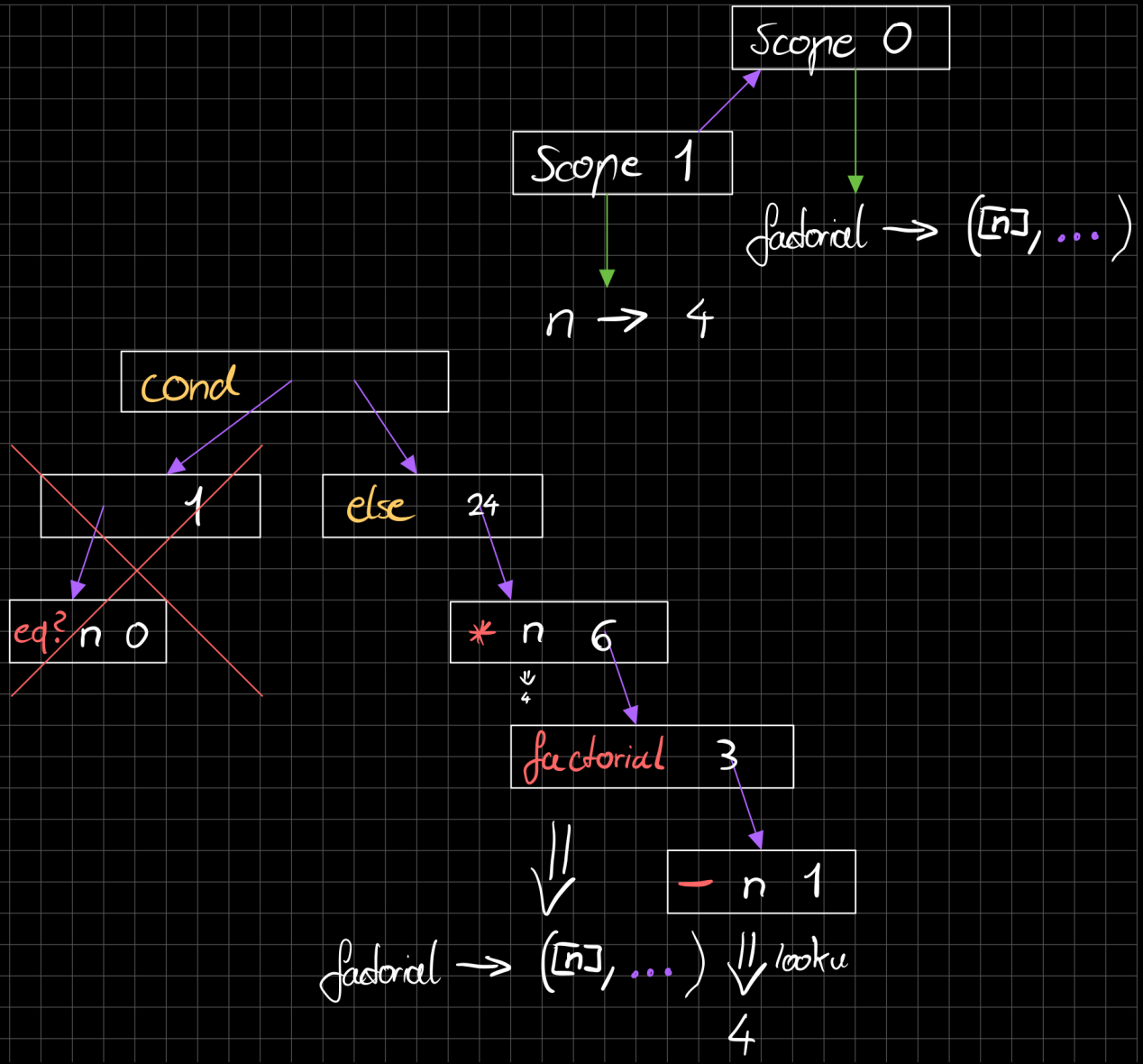

Scope 0
Scope 1
factorial → ([n], ...)
n → 4
cond
1
else
eq? n 0
* n
factorial 3
- n 1
factorial → ([n], ...)
lookup
4

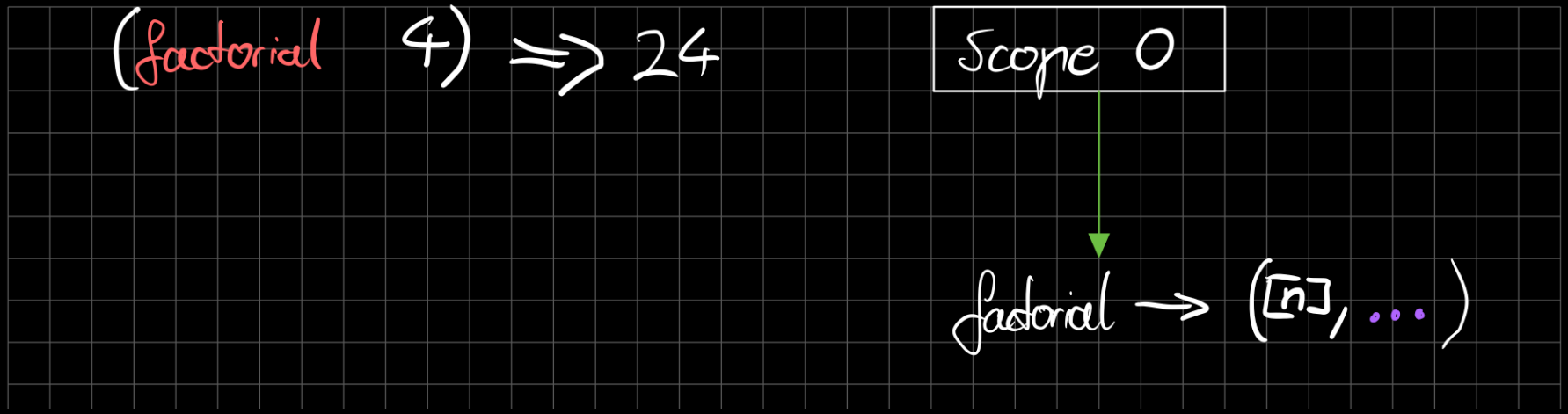

Scope 0
Scope 1
Scope 2
factorial → ([n], ...)
n → 4
n → 3


Scope 0
Scope 1
factorial → ...
Scope 2
n → 4
Scope 3
n → 3
Scope 4
n → 2
Scope 5
n → 1
n → 0


Scope 0
Scope 5
factorial → ([n], ...)
n → 0
cond
1
else
24
eq? n 0
* n 6
factorial
- n 1


Scope 0
Scope 4
factorial → ([n], ...)
n → 1
cond
1
else
eq? n 0
* n 1
1
factorial 0
- n 1


Scope 0
Scope 1
factorial → ([n], ...)
n → 4
cond
1
else 24
eq? n 0
* n 6
4
factorial 3
- n 1
factorial → ([n], ...)
lookup
4


(factorial 4) => 24
Scope 0
factorial → ([n], ...)

# Mein Endstand

- In Rust
- 1.1k LOC
- 1.5 Monate

# Demo:

# Was man noch machen koennte

- “grosses Scheme”
- Std Libary
- Compiler statt Interpreter bauen

# Kurze Uebersich ueber die Kapitel

(1) Expressions und Listen manipulation
(2) Simple Rekursive Funktionen
(3) Bauen eigener listen manipulierenden Funktionen
(4) Bauen eigener Mathematischen Funktionen (+,-,*,//,>,<,=)
(5) Listen von Listen durchgehende Funktionen (func*)
(6) Bau eines einfachen Taschenrechners fuer mathematische expr
(7) Sets und mit Sets arbeitenden Funktionen (set?,subset?,intersect,union)
(8) Funktionen mit Funktionen als Parameter
(9) Kleine Einfuehrung vom Y-Combinator
(10) Simpler Interpreter fuer Little Scheme in Little Scheme

# Github link zum Projekt

- https://github.com/lquiji/little-schemer-talk

# The Little Schemer:

## Building an interpreter one feature at a time

Leon Roth

02 June 2022

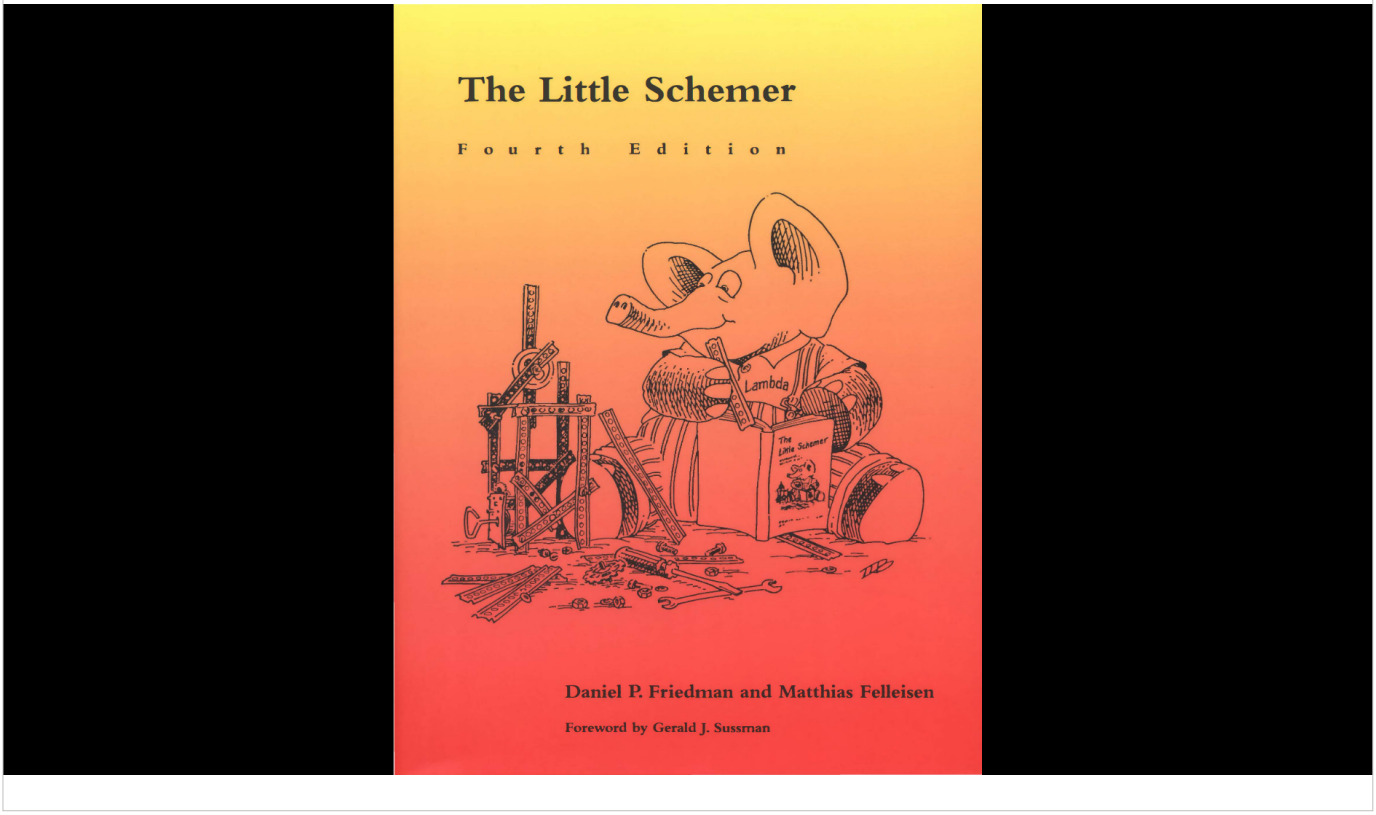
# Zu mir

Hobby: boulder und was auch immer ich lust hab zu programmieren
Unteranderem Compiler

Kurz Thema simpler eigener Interpreter erklaert

# Meine Motivation

Die Magie hinter Sprachen zu durchbrechen



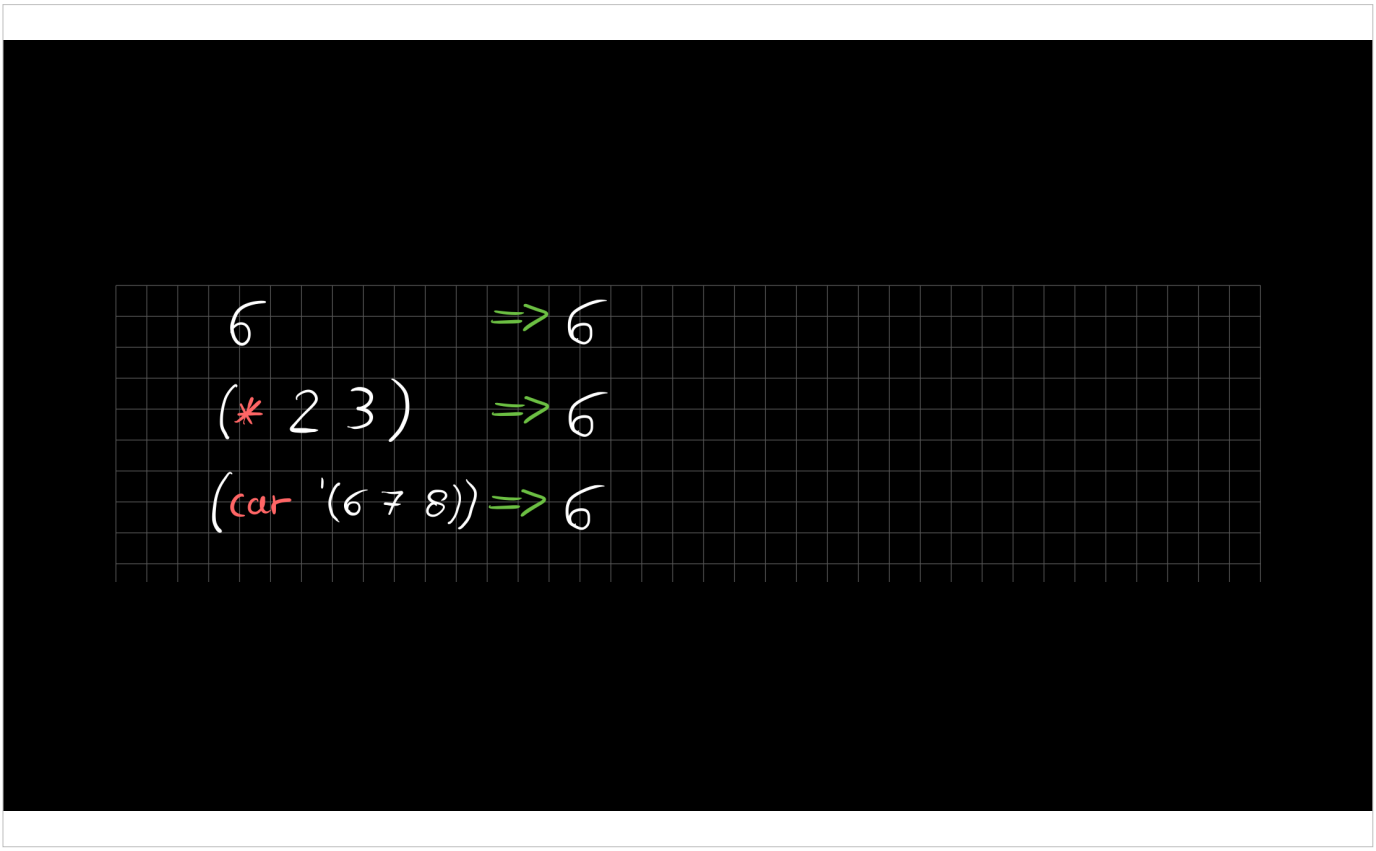

- Lehrt rekursives denken und dass verstehen von Programmen
- 1974
- Basiert auf einer MIT-Vorlesung
- Benutzt ein Subset von Scheme (Lisp Dialekt)
- 10 aufeinander aufbauende Kapitel
- Frage --> Antwort Stil
-
- Fuer das lernen von Scheme und funktionalen sprachen super
- Ohne andere Literatur daraus einen Interpreter bauen
  - Nicht die gedachte idee
-
- Faengt in Kapitel eins einfach an mit listen primitiven
- Darauf aufbauend

# Little Scheme

```scheme
(define factorial
  (lambda (n)
    (cond
      ((eq? n 0) 1)
      (else (* n (factorial (- n 1)))))
    )
  )
)
```

```rust
fn factorial(n: u32) -> u32 {
  if n == 0 {
    1
  } else {
    n * factorial(n - 1)
  }
}
```

Mehr die eigenschaften erklaeren
Rust und Scheme nennen
Einzelne punkte nicht erklaeren
Beispiel auf eigene slide




Polish notation
Example 1 und 3 weg und noch
(factorial 3) dazu

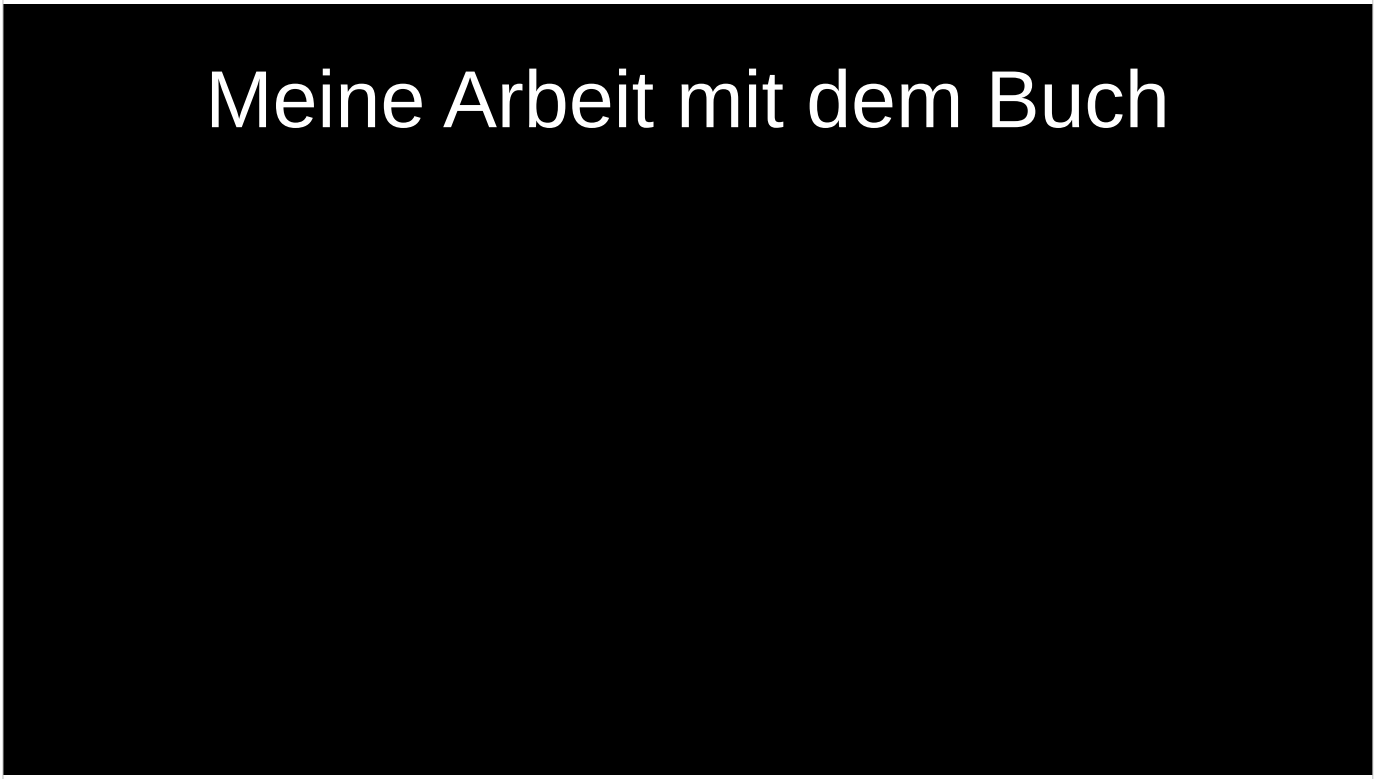
No answer.[1]
You cannot ask for the *cdr* of the null list.

---

[1] L: `nil`

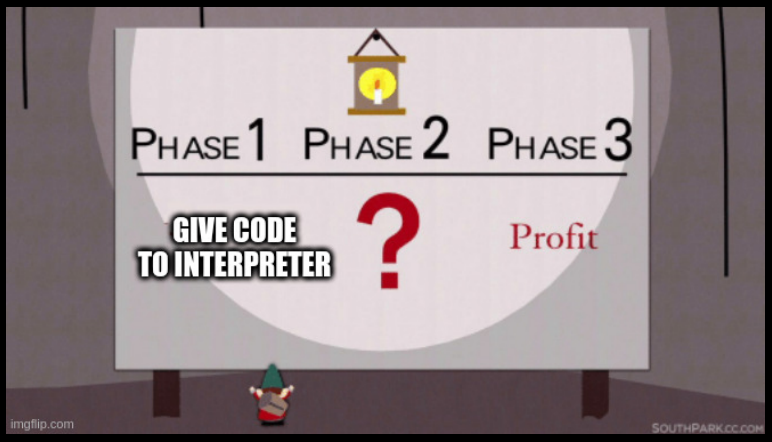
Schlampig offen laesst und nichts erklaert

# Meine Arbeit mit dem Buch

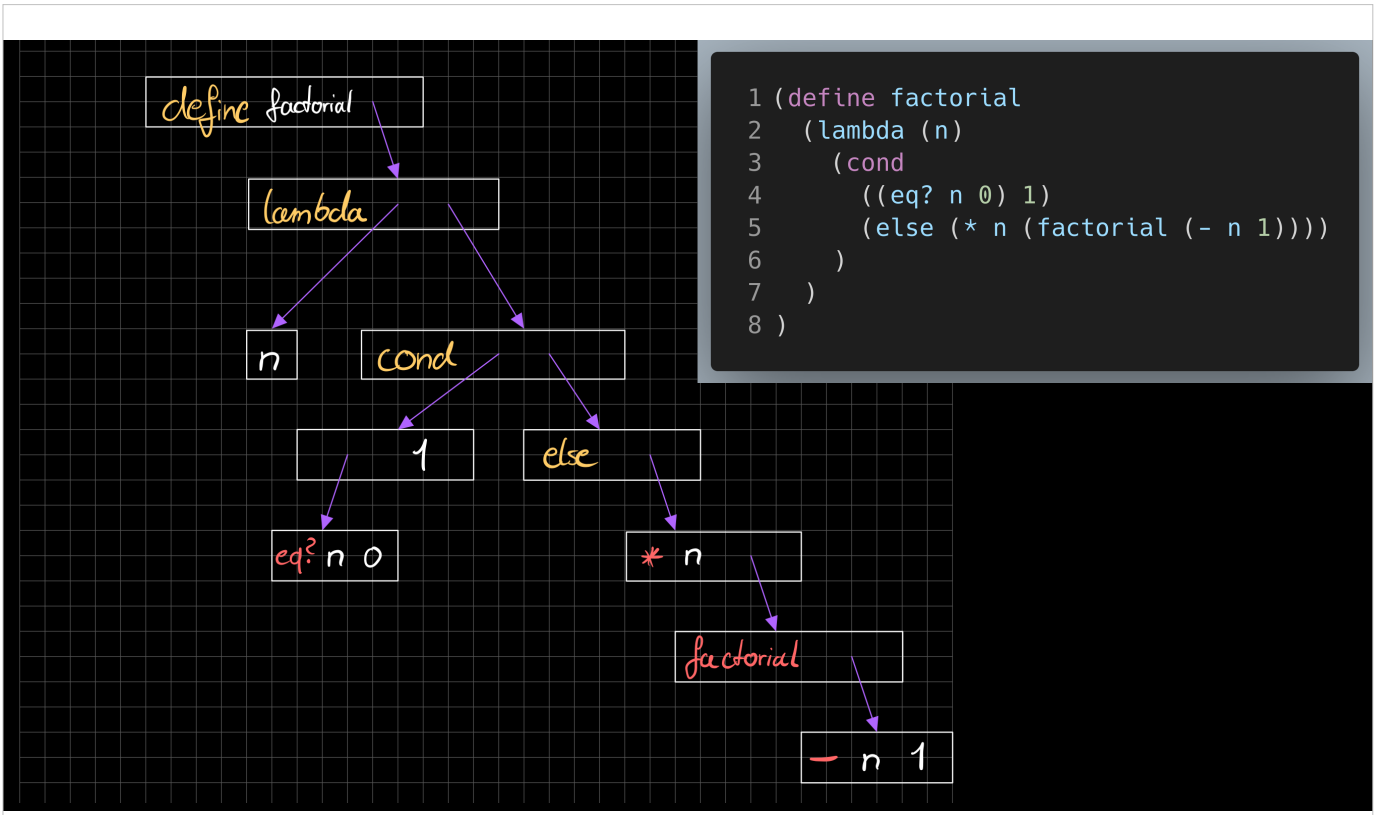


- Erstes Kapitel Gut
- Zweites zu viel auf einmal fuer einen Interpreter ohne Hilfe
- Muendlich zu Buch Slide dazu

# Interpreter




- Genereller Plan: Meme
- Nur meme und flowchart indikator
  - 2 Phasig: Verarbeitung zu einem Abstrakten Syntax Baum und dann die eigentliche Interpretation
  - Tokenizer:
    - Unterteilt in Woerter in interne Repraesentierung
  - Parser:
    - Baut einen Baum aus dem Programm
  - Interpreter:
    - Arbeitet sich zur Laufzeit durch den Baum




Parser

Klammer fuer Klammer

Jede Box eine Liste

List von Listen bildet einen Baum

# Parser

```
1 ⚠️ (car ('first 'second 'third)) => ('first)
2
3 ✅ (car ('first 'second 'third)) => 'first
```

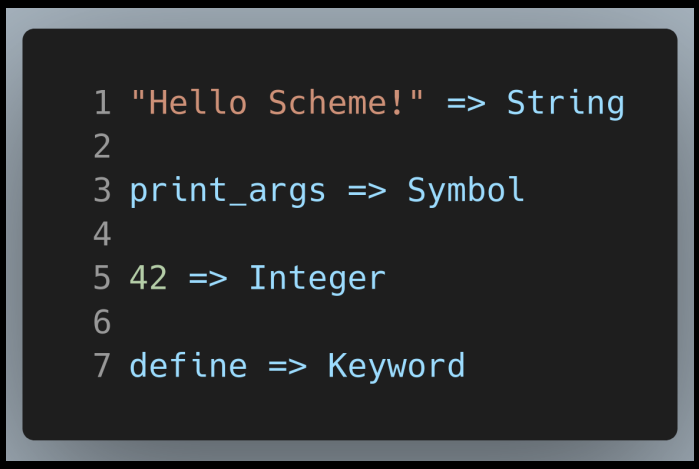
## Little Schemer Schlampig mit Syntax Weniger Text

- Problem: String Parsing beim Interpretieren –> zuviele listen probleme

## Ein Wenig geschichtlicher und papa annecken

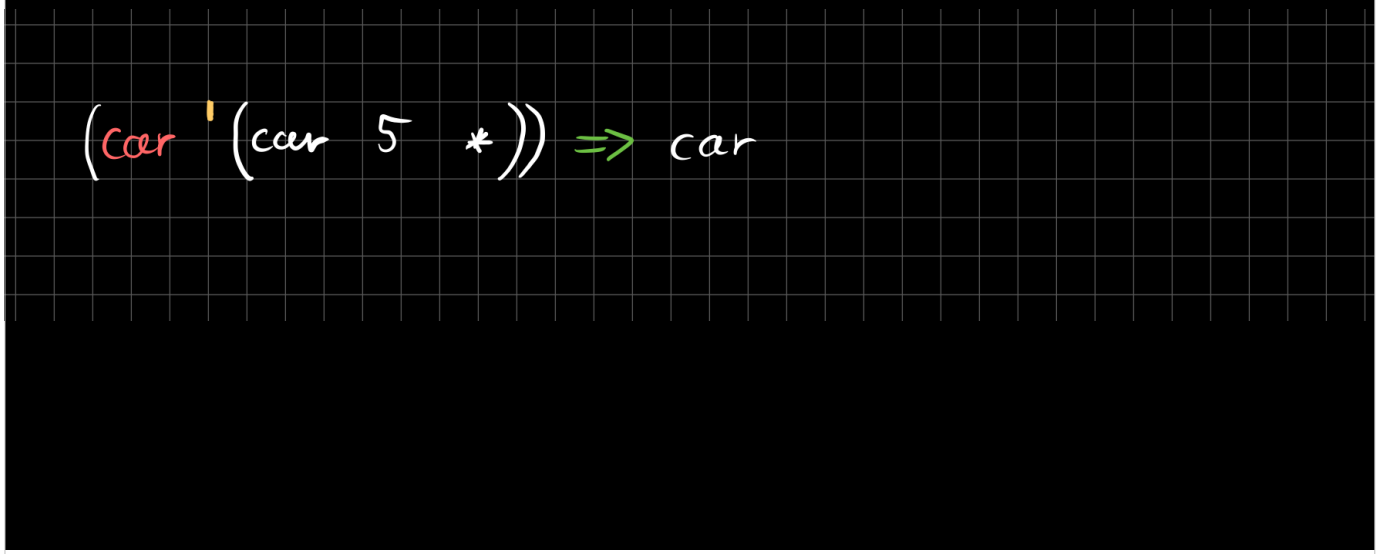
- Relativ einfach da man S-Expressions gut als Baume darstellen kann
  - Listen von Listen
- Klammern zeigen eine neue Tiefe an

# Tokenizer



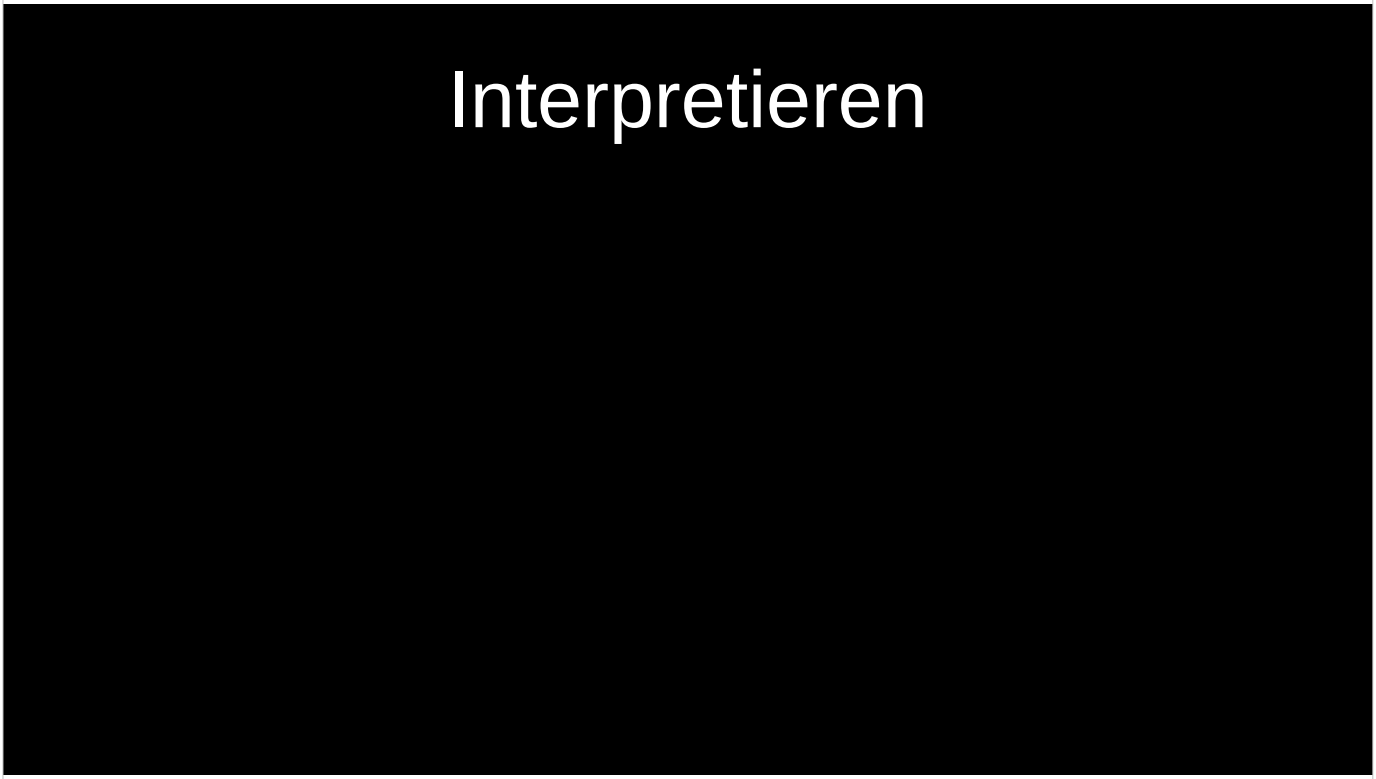

- Einteilen von einzelnen Woerter in Kategorien und Keywords
- In Little Schemer:
  - String
  - Number
  - Symbol
  - Boolean
  - Keyword (define, lambda, let, cond)
- Einfache If else kette implementieren lassen

# Quoting in Scheme




Daten statt ausfuehrbaren teil
Use case nicht ausfuehren sonder
liste machen for example
List statt quoting um liste zu machen

# Interpretieren



Klarer machen wie es die tiefsten zuerst durchgeht
Baum mit Pfeil unten nach oben
Rest oral

# Scope

In jeder sprache schwierig Scoping zu verstehen.

Kurze graphik

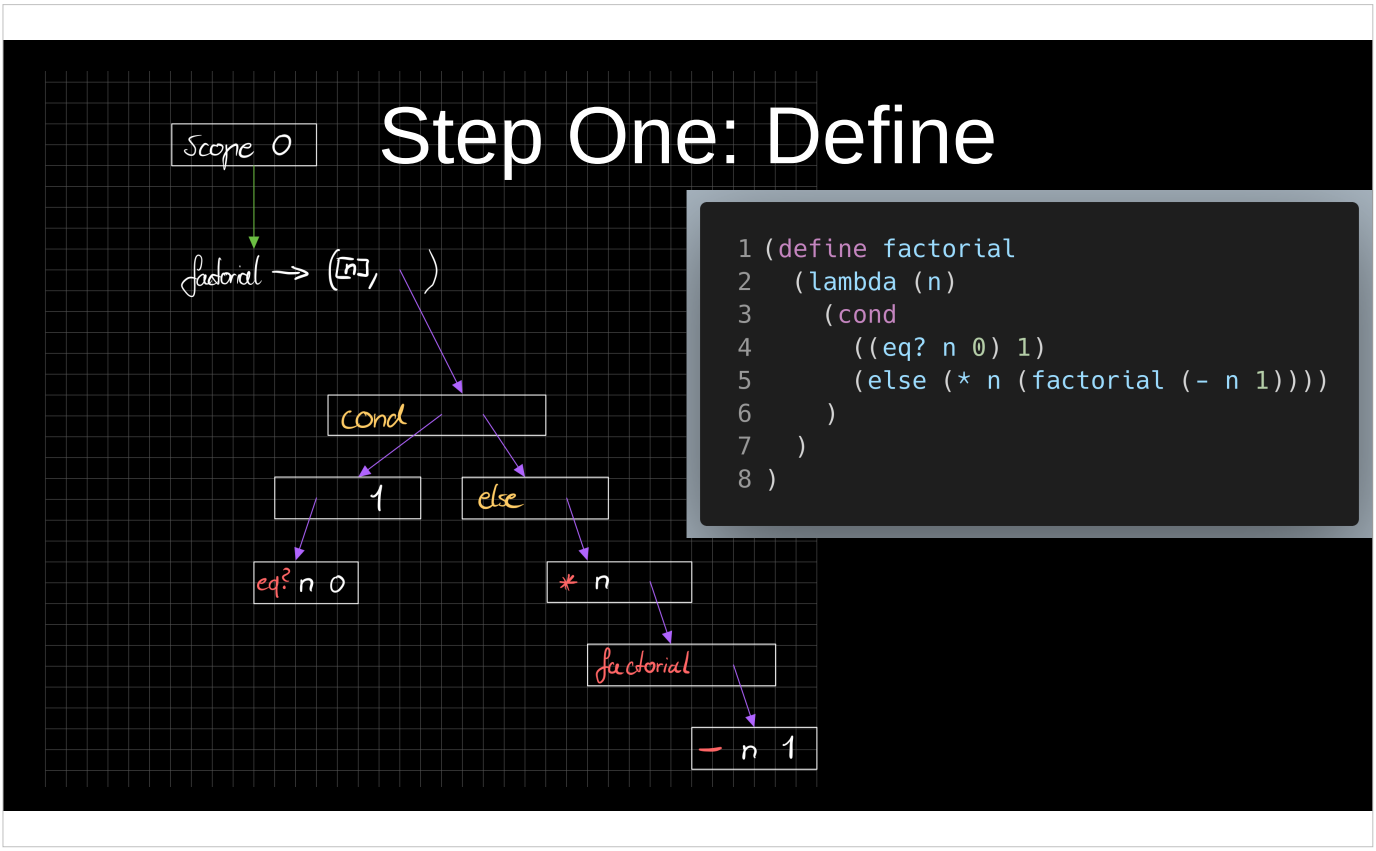
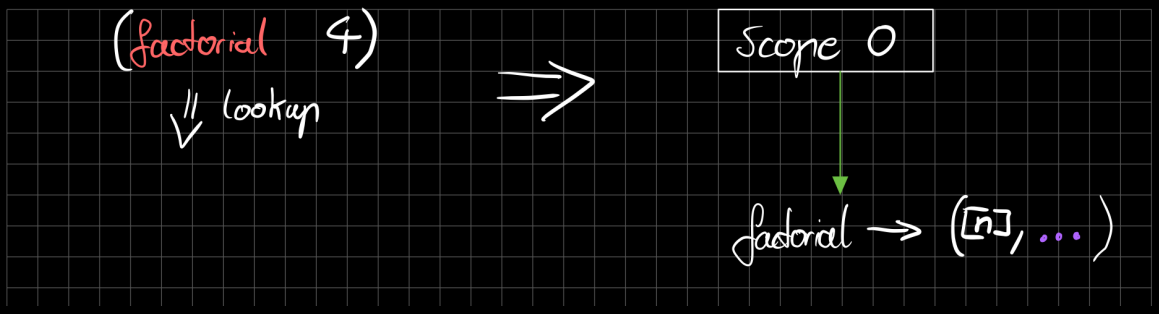
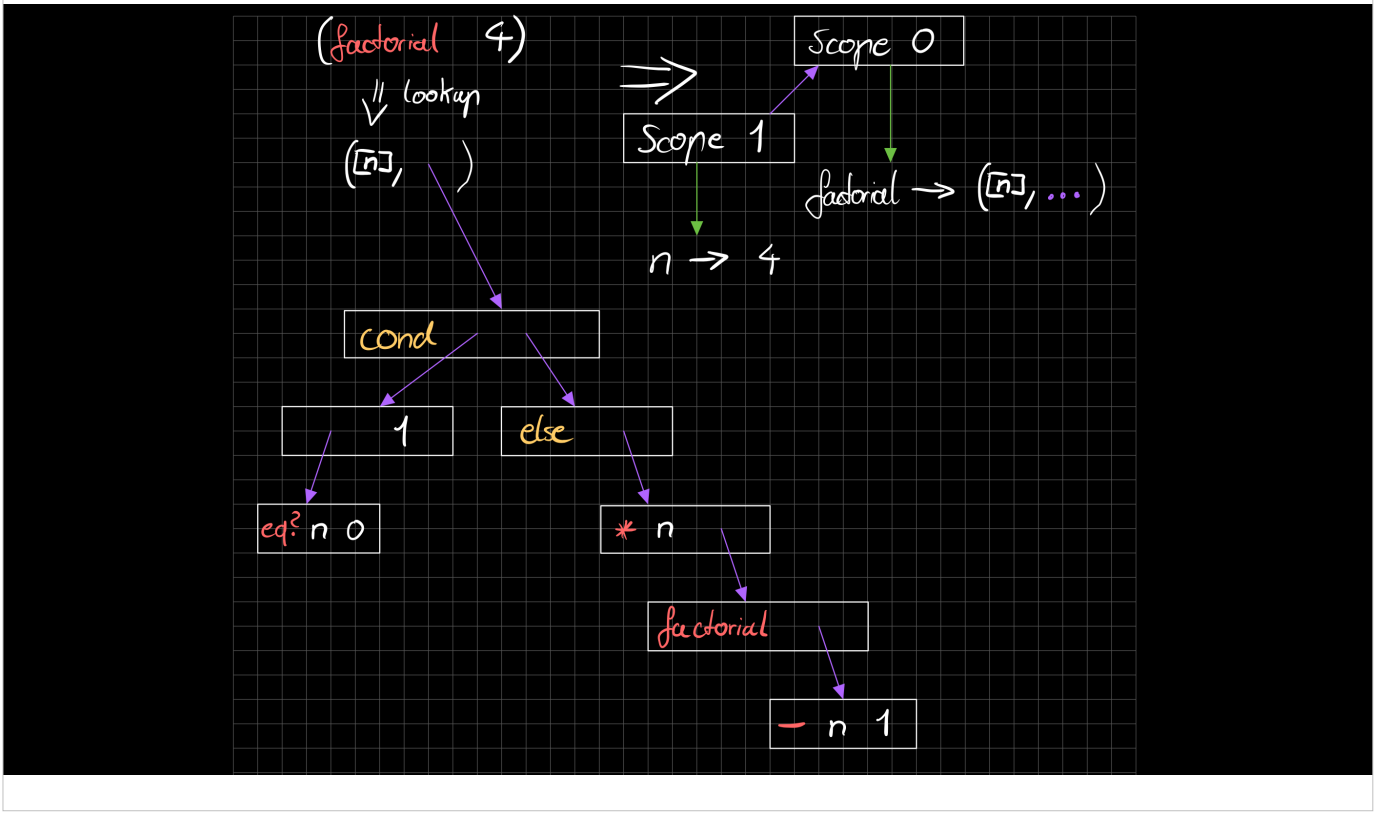


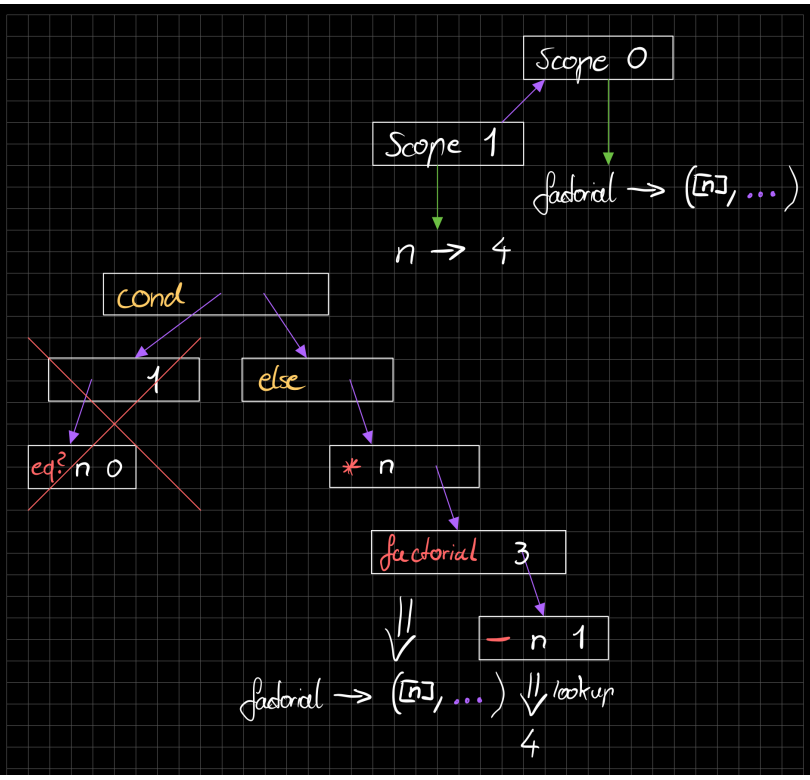

Bild vom code weg

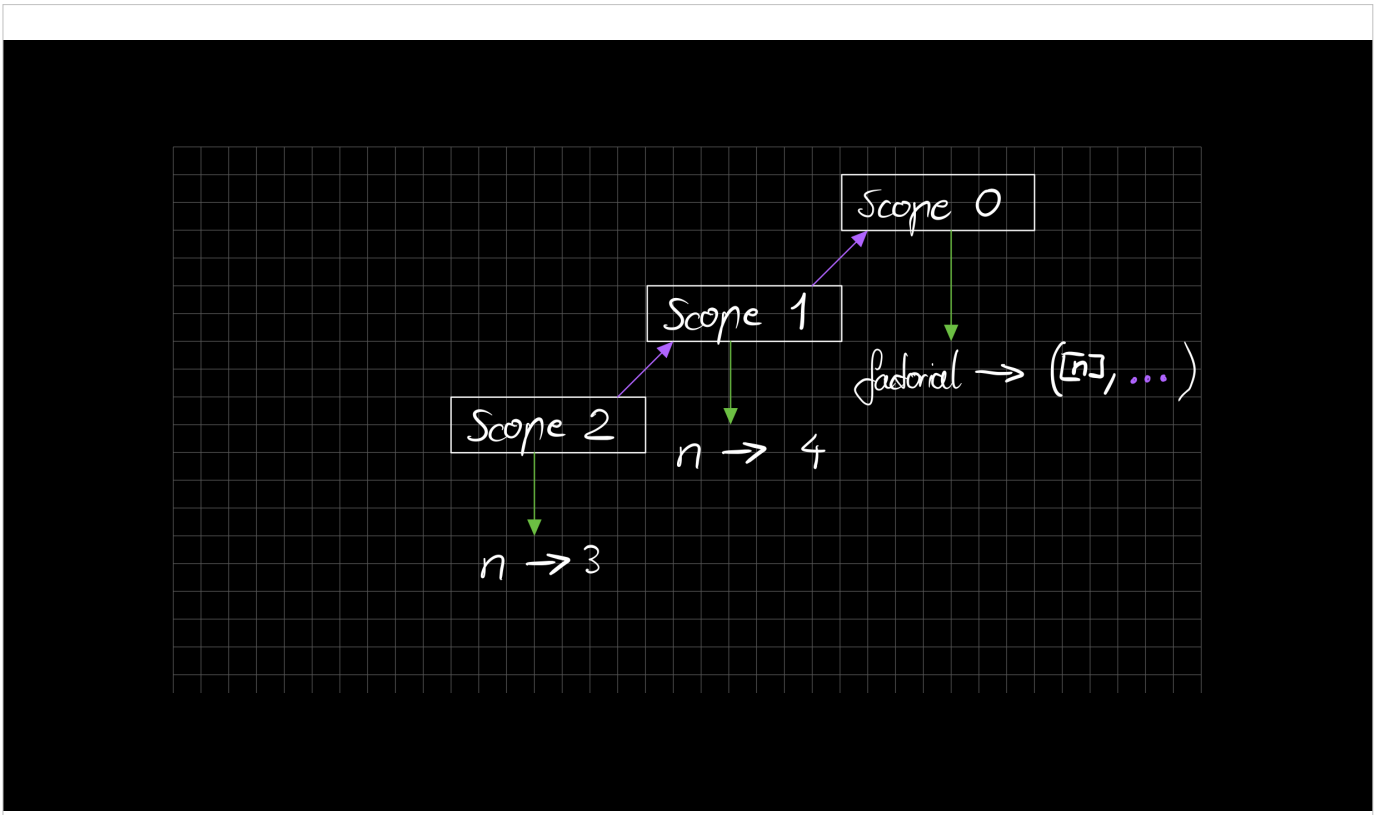
# Step Two: Call



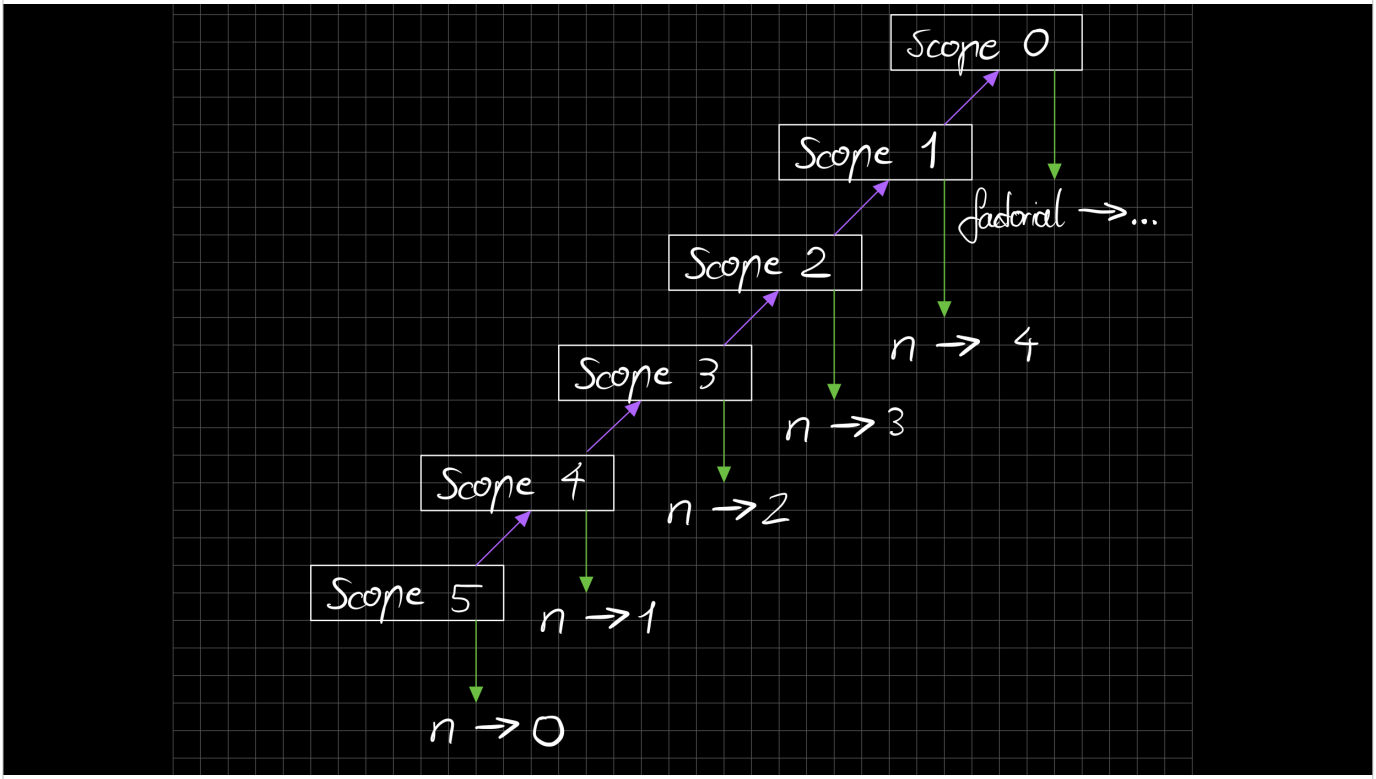


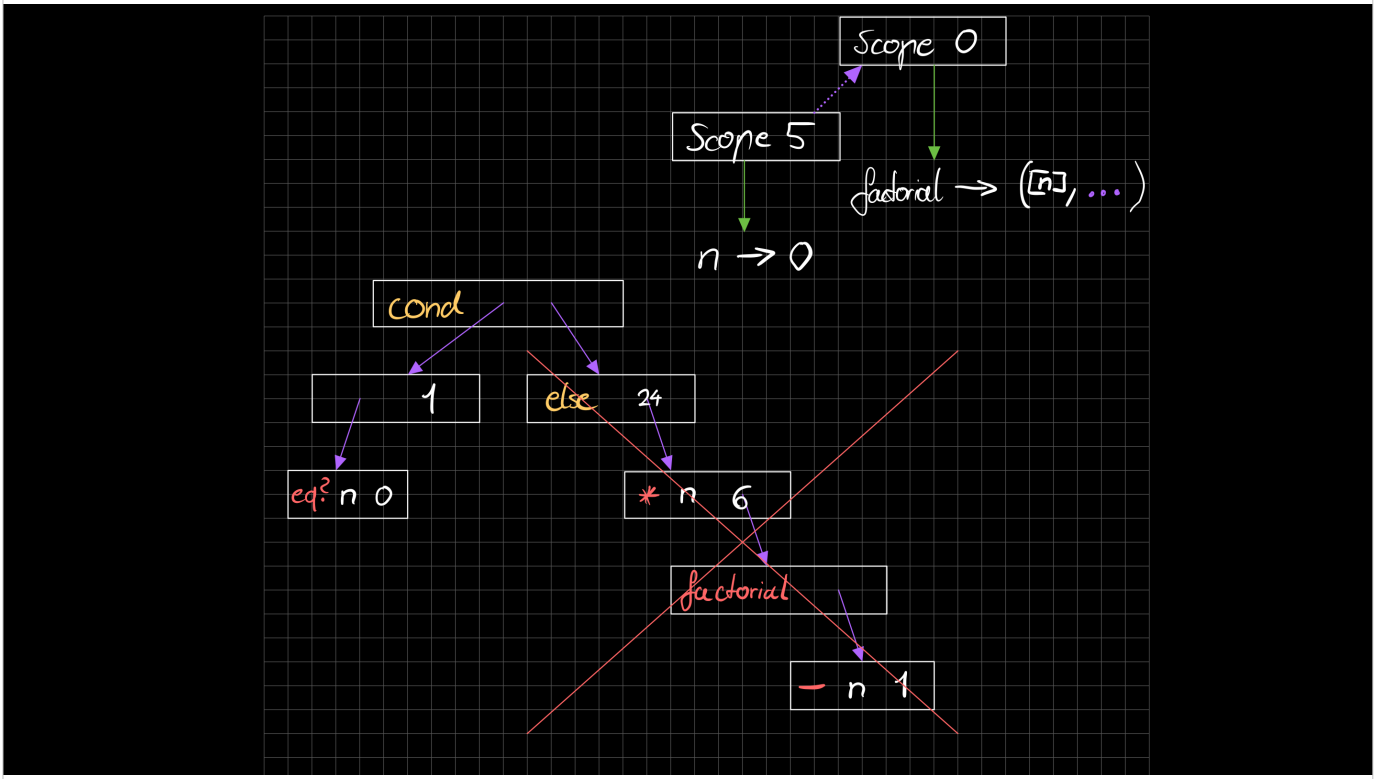

Pfeilen aufpassen und roter pfeil an der seite wo sind wir?


Scope 0
Scope 1
factorial → ([n], ...)
n → 4
cond
1
else
eq? n 0
* n
factorial 3
- n 1
factorial → ([n], ...)
lookup
4

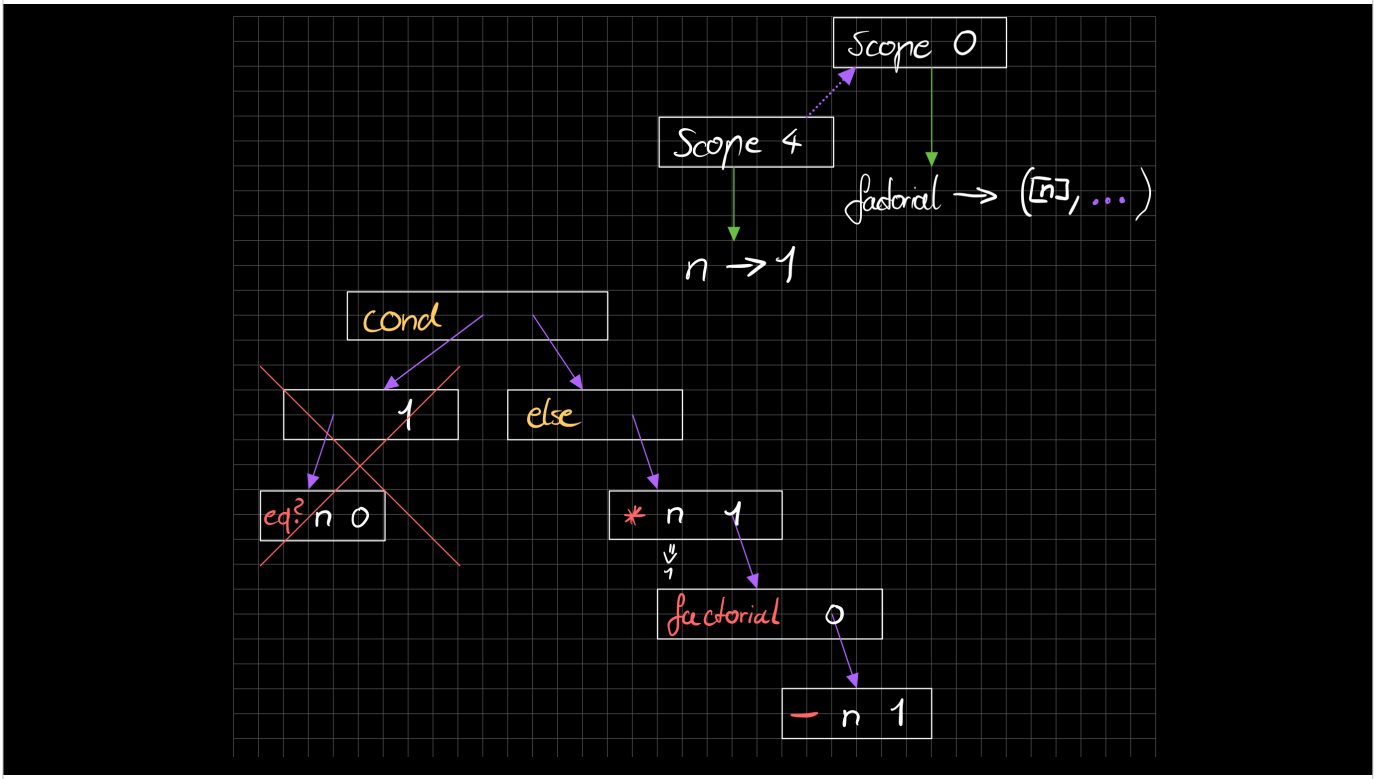


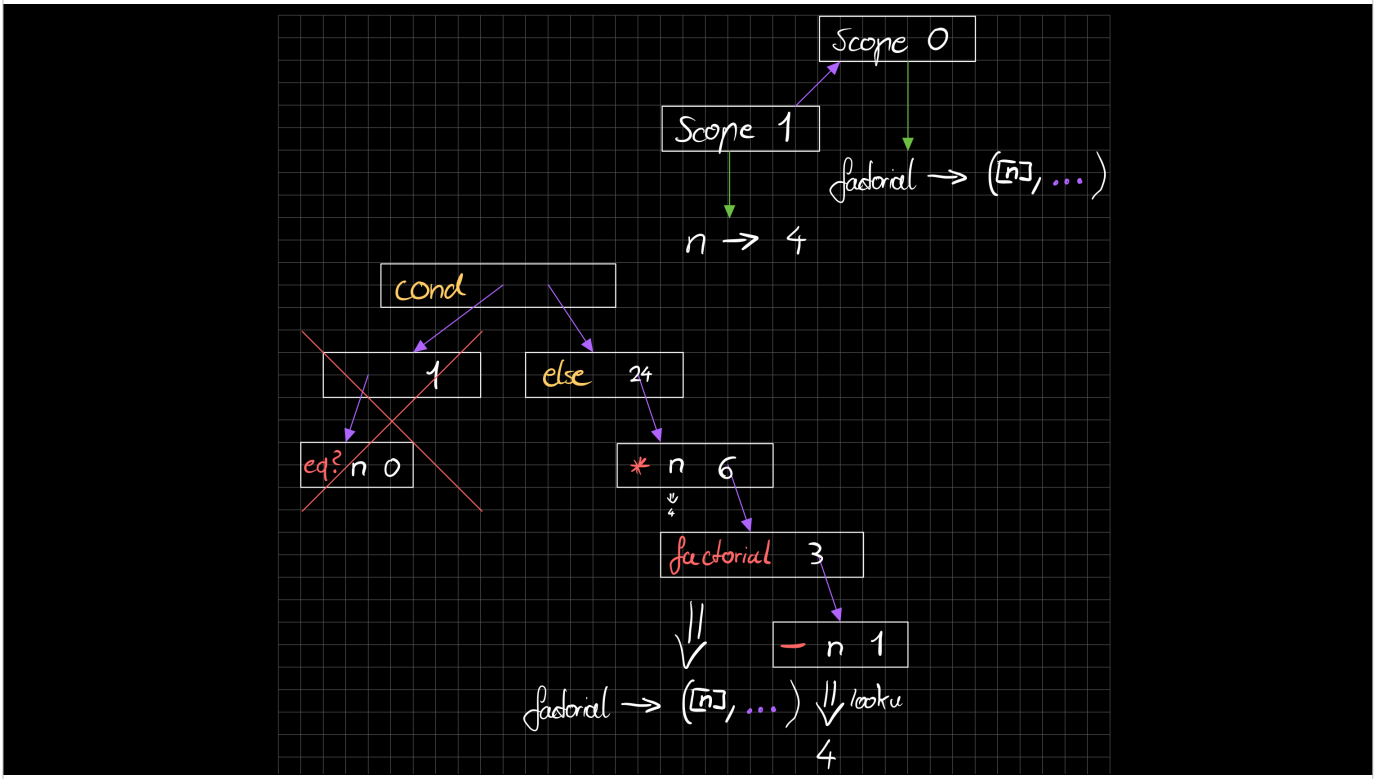

Das hier weg nur naechstes

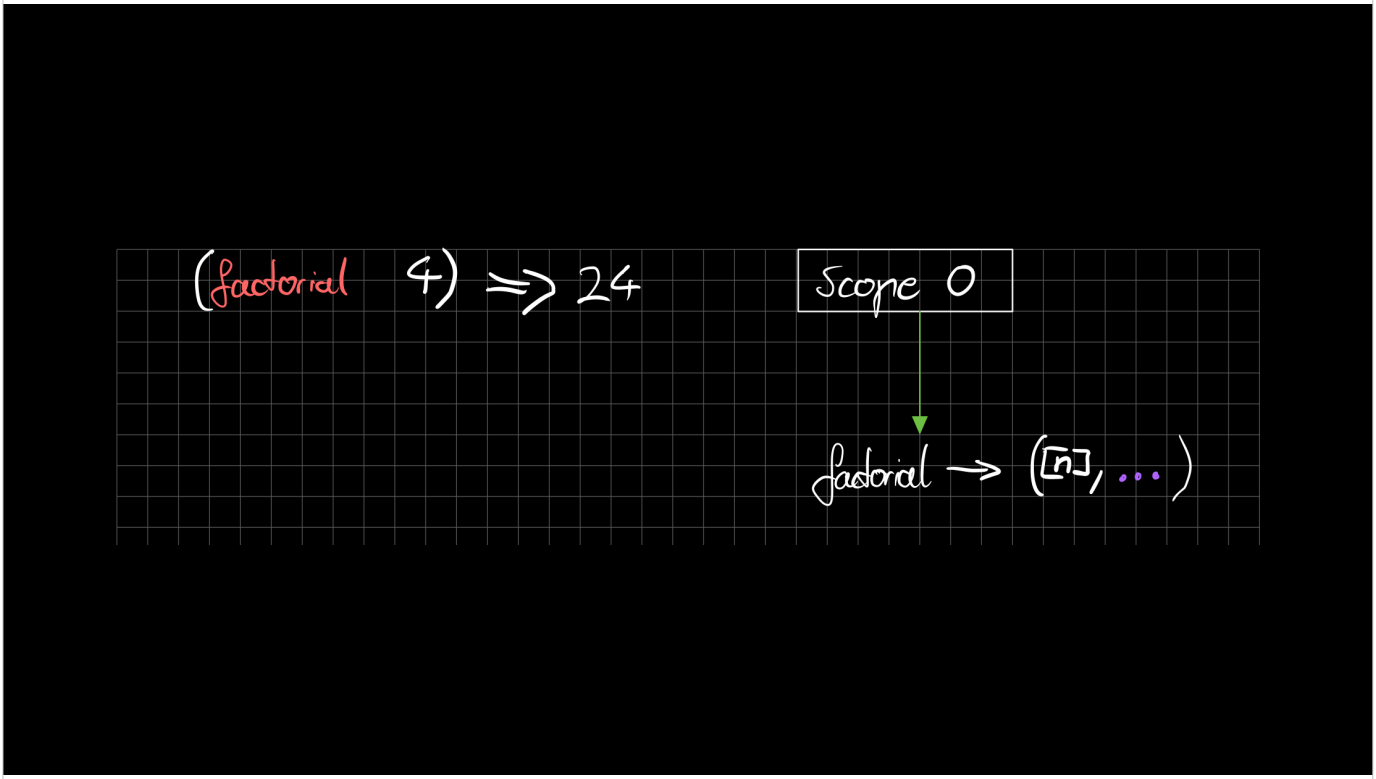

Scope 0
Scope 1
factorial →...
Scope 2
n → 4
Scope 3
n → 3
Scope 4
n → 2
Scope 5
n → 1
n → 0


Scope 0
Scope 5
factorial → ([n], ...)
n → 0
cond
1
else
24
eq? n 0
* n 6
factorial
- n 1

Fix this

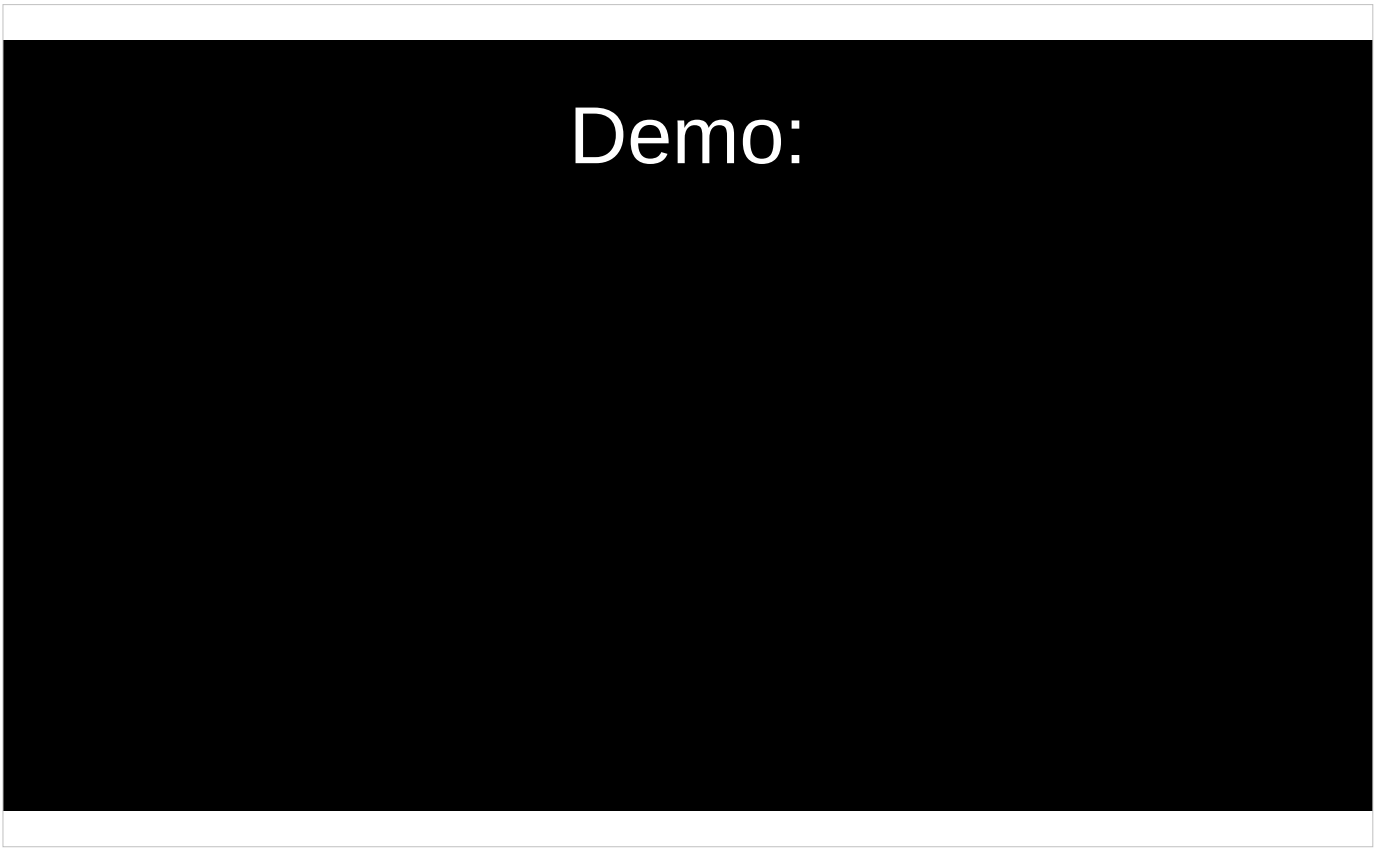

Scope 0
Scope 4
factorial → ([n], ...)
n → 1
cond
1
else
eq? n 0
* n 1
factorial 0
- n 1


Scope 0
Scope 1
factorial → ([n], ...)
n → 4
cond
1
else 24
eq? n 0
* n 6
4
factorial 3
- n 1
factorial → ([n], ...)
lookup
4


(factorial 4) => 24
Scope 0
factorial → ([n], ...)

# Mein Endstand

Alles unit tested bis Kapitel 4 und Y combinator
(Kapitel 2 schon basicly alles)




Der Beweis!

# Was man noch machen koennte

Dann Zu sehr von compiler abgelengt

# Kurze Uebersich ueber die Kapitel

# Github link zum Projekt

https://github.com/Iquiji/little-schemer-talk

Vielleicht diese Seite Streichen