The Little Schemer:

Building an interpreter one feature at a time

Leon Roth

02 June 2022

Meine Motivation

Die Magie hinter Sprachen zu durchbrechen

The Little Schemer

Fourth Edition

Daniel P. Friedman and Matthias Felleisen

Foreword by Gerald J. Sussman

(* 2 3) ⇒ 6

(factorial 3) ⇒ 6

Little Scheme

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

```
1 (define factorial
2   (lambda (n)
3     (cond
4       ((eq? n 0) 1)
5       (else (* n (factorial (- n 1))))
6     )
7   )
8 )
```

```
1 fn factorial(n: u32) -> u32 {
2   if n == 0 {
3     1
4   } else {
5     n * factorial(n - 1)
6   }
7 }
```

No answer.[1]

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

---

[1] L: `nil`

Interpreter

```
1 (define factorial
2   (lambda (n)
3     (cond
4       ((eq? n 0) 1)
5       (else (* n (factorial (- n 1))))
6     )
7   )
8 )
```

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

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

Quoting in Scheme

'(* 2 3) => (* 2 3)

(* 2 3) => 6

'* => Symbol

* => Funktion

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

Scope

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

```
1 (define factorial
2   (lambda (n)
3     (cond
4       ((eq? n 0) 1)
5       (else (* n (factorial (- n 1))))
6     )
7   )
8 )
```

(factorial 4)

Scope 0

factorial → ([n], ...)

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

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
false
* n
factorial
- n 1

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

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

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
eq? n 0
* n
factorial
- n 1

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

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

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

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

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

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

(factorial 4) => 24

Scope 0

factorial -> ([n], ...)

Mein Endstand

- In Rust
- 1.1k LOC
- 1.5 Monate
- Bis Kapitel 4 alles unit getestet

Demo:

Was man noch machen koennte

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

Github link zum Projekt

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

Kurze Übersicht über 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