The Little Schemer:

Building an interpreter one feature at a time

Leon Roth

02 June 2022

Kurz: wer bin ich ueberhaupt

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

Und erzaehle heute von einem
Interpreter den ich geschrieben habe

Meine Motivation

Die Magie hinter Sprachen zu durchbrechen

Interresiert
Um tieferen einblick zu haben

- Eines Abends nach Zeugs gefragt um Javascript nachzubauen
- Soll ich mir nicht antuen => The little Schemer
- “Sollte gut daraus einen Interpreter bauen koennen”
- Also mal reingelesen

Gemerkt relativ einfacher Syntax

Erst immer in klammern die Funktion und dann die Parameter

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)
  }
}
```

Simpel in Kapitel eins anfaengt,
aber dann schon im zweiten der Zehn
  Funktionen an dich wirft

Code in Scheme Oben
Equivalaenter in Rust Unten

No answer.[1]

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

---

[1] L: `nil`

Schoen und Gut zum Scheme lernen
  aber
Schlampig
Viel offen laesst
nichts erklaert
Auch fuer Scheme eher lernen durch
  machen ansatz

Interpreter

War erstmal Plan: Meme
Ohne weitere Literatur

Erstmal angefangen den Code anhand der Klammern in Bloecke aufzuteilen

Parser
Baum aus dem Programm

Klammer fuer Klammer
Jede Box eine Liste
List von Listen bildet einen Baum

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

Little Schemer Schlampig mit Syntax

- Ganze Nacht am verzweifeln
- Problem: String Parsing beim Interpretieren –> zuviele listen probleme
- Danke Papa

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

- Dann weiter gemacht indem ich die einzelne woerter
- interne Repraesentierung
- Einteilen von einzelnen Woerter in Kategorien und Keywords
- Scheme zum glueck nur wenige Kategorien
- In Little Schemer:
  - String
  - Number
  - Symbol
  - Boolean
  - Keyword (define, lambda, let, cond)
- Einfache If else kette implementieren lassen

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

Klarer machen wie es die tiefsten zuerst durchgeht

- Arbeitet sich zur Laufzeit durch den Baum
- 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

In jeder sprache schwierig Scoping zu verstehen.

Faktorial durch gehen
Wie der Interpreter es intern auch macht

(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

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

Was hab ich gelernt?
* Tieferes Verständnis für Programmabläufe
* Scoping & Rekursion
* Was eine Sprache mit dem Code macht
* Erstes größeres Projekt abgeschlossen

Demo:

Der Beweis!

Was man noch machen koennte

Dann Zu sehr von compiler abgelengt

Github link zum Projekt

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

Vielleicht diese Seite Streichen

Kurze Übersicht über die Kapitel