

MAL - Making a Lisp

Vasilij Schneidermann

Januar 2017

- 1 Einführung
- 2 MAL
- 3 Implementierung
- 4 Weitere Schritte

Abschnitt 1

Einführung

- Vasilij Schneidermann, 24
- Wirtschaftsinformatikstudent
- Praktikant bei [bevuta IT GbmH](#)
- v.schneidermann@gmail.com
- <https://github.com/wasamasa>
- <http://emacs horrors.com/>
- <http://emacs ninja.com/>

- Lisp relativ einfach erlernbar
- Feinheiten weniger
- Quoting, Makros, Environments, eval, etc.
- Besseres Verständnis durch Implementieren von Lisp!
- MAL in vieler Hinsicht ideal
- Durch den Sprecher in Emacs Lisp, Chuck und Picolisp implementiert

Was ist MAL?

- Minimale, an [Clojure](#) angelehnte Programmiersprache
- Lisp-Implementierung in vielen Sprachen, [inklusive MAL](#)
- Showcase à la [Rosetta Code](#)
- Anleitung zum Interpreter-Bau
- Pädagogisches Werkzeug
- Einbettbar!

Warum nicht Scheme?

- [Scheme](#) ist deutlich größer
- Herausfordernder (call/cc, hygienische Makros, RNRS)
- Weniger Hilfestellung gegeben
- Etwas weniger praktikabel als Clojure (weniger eingebaute Datenstrukturen)
- Alternativ: [Forth](#), [Smalltalk](#)

Abschnitt 2

MAL

- Autor: Joel Martin ([@kanaka](#))
- Inspiration durch [gherkin](#) (Lisp in Bash 4)
- Schrieb ein Lisp in GNU Make (MAke Lisp)
- Abstrahierte die Gemeinsamkeiten weiterer Implementierungen
- Umbenennung zu Make A Lisp

Fun Facts

- Lizenz: MPL 2.0
- Contributors: 52
- Implementierungen: 63
- Schritte: 11
- Unit Tests: 636

- Abgespecktes Clojure
- Keine Namespaces, Concurrency, Lazy Sequences, Protokolle, Multimethods, First-Class Interop, ...
- `true`, `false`, `nil`, `Int`, `String`, `Symbol`, `Keyword`, `List`, `Vector`, `Hash Map`, `Atom`
- Unterstützt Variablen, Funktionen, Environments, Closures, TCO, IO, `eval`, Quoting, Makros, Exceptions, Metadaten
- 57 eingebaute, 10 besondere, 3 benutzerdefinierte Funktionen
- Hinreichend für Self-Hosting

Code-Beispiele

```
1 ;=> 1
```

```
"foo" ;=> "foo"
```

```
:bar ;=> :bar
```

```
'baz ;=> baz
```

```
true ;=> true
```

```
false ;=> false
```

```
nil ;=> nil
```

```
'(1 2 3) ;=> (1 2 3)
```

```
[4 5 6] ;=> [4 5 6]
```

```
{:foo 1 :bar 2 :baz 3} ;=> {:foo 1 :bar 2 :baz 3}
```

```
(+ 1 1) ;=> 2  
(first [1 2 3]) ;=> 1  
(rest [1 2 3]) ;=> (2 3)  
(get :answer {:answer 42}) ;=> 42  
(read-string "(+ 1 1)") ;=> (+ 1 1)
```

```
(if (= (+ 1 1) 2)
    "Yay"
    "...") ;=> "Yay"
```

```
(cond
  (= (+ 1 1) 2)
  "Yay"

  :else
  "...") ;=> "Yay"
```

Code-Beispiele

```
(def! x 1)  
x ;=> 1
```

```
(let* [x 2]  
  (+ x x)) ;=> 4
```

```
(let* [x 3]  
  (def! x (+ x 1))  
  x) ;=> 4
```

```
x ;=> 1
```


Aus dem Guide entnommen:

```
(def! not
  (fn* [x]
    (if x false true)))
```

```
(not false) ;=> true
```

```
(not true) ;=> false
```

Aus dem Guide entnommen:

```
(def! load-file
  (fn* [f]
    (eval
     (read-string
      (str \"(do \" (slurp f) \"\")\"))))
```

Triviales Makro:

```
(defmacro! comment  
  (fn* [& body]  
    nil))
```

```
(comment (/ 1 0)) ;=> nil
```

Exceptions:

```
(try* (/ 1 0) (catch* ex (prn "Prevented doom")))
;; "Prevented doom"
;; nil
```

Einfache Rekursion:

```
(def! fact
  (fn* [x]
    (if (<= x 1)
        x
        (* x (fact (- x 1))))))
```

```
(fact 4) ;=> 24
```

Komplexere Rekursion:

```
(def! reverse
  (let* [reverse*
        (fn* [arg acc]
          (let* [xs (seq arg)]
            (if xs
                (reverse* (rest xs) (cons (first xs) acc))
                acc))))]
    (fn* [xs] (reverse* xs ())))))
```

```
(reverse [1 2 3]) ;=> (3 2 1)
```

- Vorgefertige Make-Targets
- Viele Unit Tests
- CI
- Guide
- Cheatsheet
- Überschaubare Implementierungsschritte
- Balance zwischen Aha-Momenten und Schwierigkeit

- IRC-Channel auf Freenode, #ma1
- GitHub

Abschnitt 3

Implementierung

Auswahl einer geeigneten Implementierungssprache

- Theoretisch jede Turing-vollständige Sprache denkbar
- Abstrusere Beispiele:
 - AWK
 - Bash
 - Chuck
 - Emacs Lisp / VimScript
 - LOGO
 - GNU Make
 - MATLAB / R
 - PL/pgSQL / PL/SQL
 - PostScript
 - Visual Basic
 - VHDL

Auswahl einer geeigneten Implementierungssprache

- Schon verwendete Sprachen im Repository
- <https://lobste.rs/>
- <http://www.tiobe.com/tiobe-index/>
- <http://langpop.corger.nl/>
- https://en.wikipedia.org/wiki/List_of_programming_languages
- http://rosettacode.org/wiki/Rosetta_Code

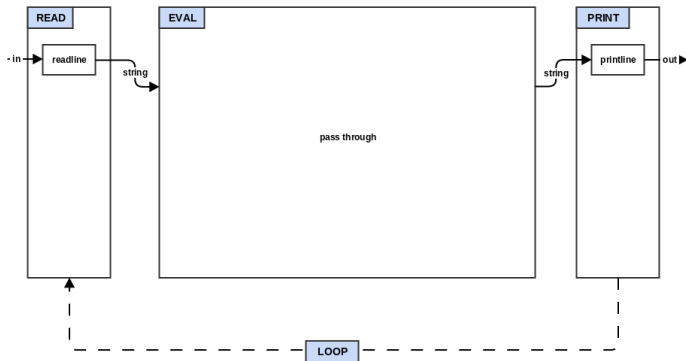
Wünschenswerte Features

- Sequentielle Datenstruktur (Array, Liste, Vektor)
- Assoziative Datenstruktur (Dictionary, Hash Table/Map, Assoziatives Array)
- Funktionsreferenzen (Funktionspointer, First-Class Functions, Delegates)
- Exception Handling (try, catch, throw)
- Varargs (apply, Splat-Operator, ...)
- Lexikalische Closures
- Reguläre Ausdrücke (nötig für READ)

Erforderliche Features

- Zahlen(!)
- (Erweiterbare) Objekte, Structs, Records
- Brauchbares statisches Typsystem / dynamische Typisierung
- IO für Konsolen-Input, -Output und Auslesen von Dateien
- Präzise Zeitmessung (alternativ "Shelling out")
- Laden von Code aus anderen Dateien (Module, `require`, `import`)
- Linux-Support (alternativ OS X, Windows), Aufruf aus Konsole

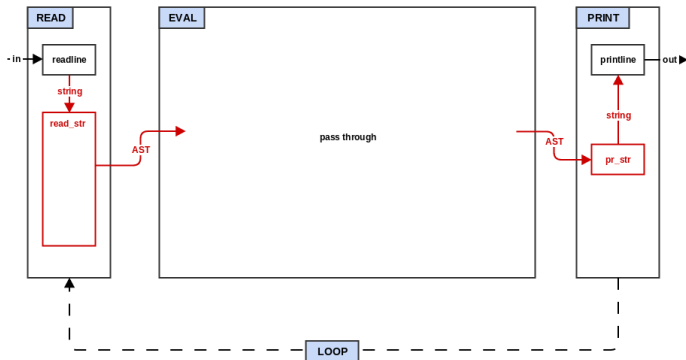
Schritt 0



Schritt 0

- REPL
- Grundlegende Konsolen-IO erforderlich
- Einlesen von Benutzereingabe sollte abbrechbar sein
- Falls nötig, kann zu Hacks gegriffen werden. . .
- Sanity Check des Testgerüsts
- Optional: Readline

Schritt 1



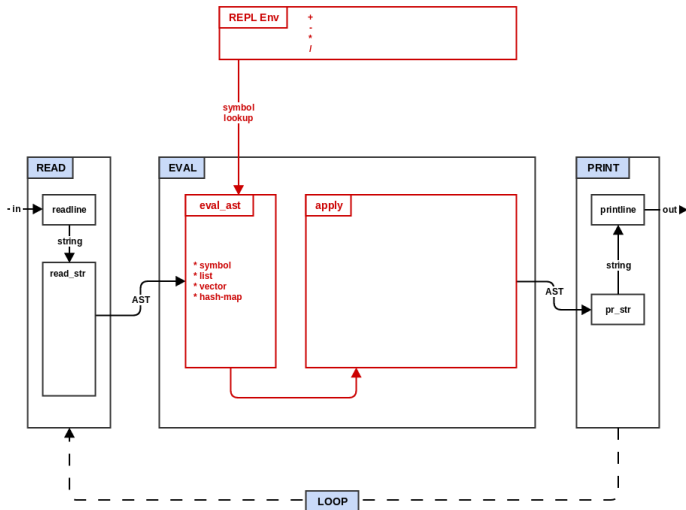
Schritt 1

- READ
- Aufteilung der Eingabe in Tokens
- "(+ 1 1)" -> "(", "+", "1", "1", ")"
- Bevorzugt mit RE, ansonsten von Hand
- `[\s,]*(~@|[\[\]\{\}\(\)\'\`~^@]|"(?:\\\.|[\^\\""])*"|
;.*|[\^\s\[\]\{\}\(\)\'\`~,;])*`
- Reader liest eine *Form* aus Tokens
- Entweder ein Skalar oder eine Liste/Vektor/Map aus weiteren Formen
- Minimale Fehlerbehandlung
- Reader-Makros werden zu Objekten konvertiert

Schritt 1

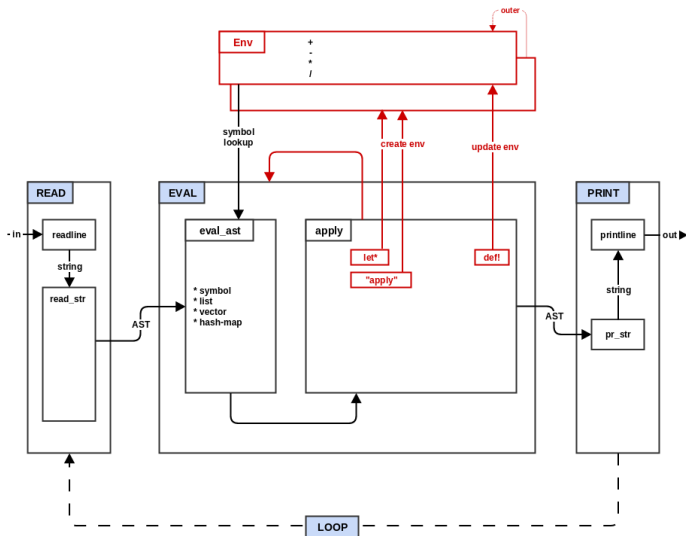
- PRINT
- Wesentlich einfacher
- Skalare werden direkt zu Strings konvertiert, komplexere Formen rekursiv
- Printer muss in der Lage sein besondere Zeichen "lesbar" zu drucken (`print` vs. `prn`)
- Alle Typen müssen repräsentierbar sein
- Korrekte Behandlung von Kommentaren und Newlines
- Schwierigster Schritt, da Parsen, Drucken und Umsetzung des Typsystems von MAL notwendig sind

Schritt 2



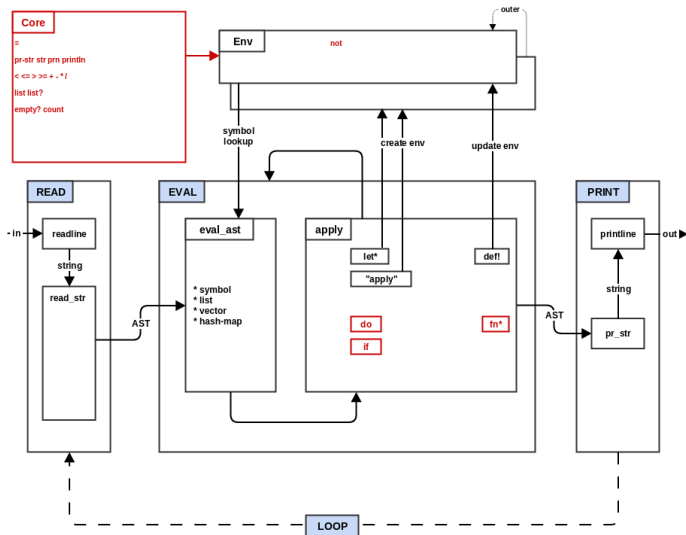
- EVAL (Tree Walker)
- Definition von REPL-Environment mit $+$, $-$, $*$, $/$
- Symbole im Environment nachschlagen
- Skalar evaluiert zu seinem Wert
- Liste wird als Funktionsaufruf (Apply-Phase) interpretiert:
- Jedes Argument evaluieren, nachgeschlagene Funktion mit Argumenten aufrufen
- Sonderfall: Leere Liste
- Resultat: Taschenrechner

Schritt 3



- Environment: Outer Environment + Bindings
- Binding: Name \rightarrow Wert
- `let*`, `def!`: Spezielle Formen, besonders behandelt in EVAL
- Normale Formen werden wie Funktionsaufrufe behandelt
- `let*` erzeugt ein neues Environment mit gegebenen Bindings
- `def!` mutiert Bindings in aktuellem Environment
- Resultat: Taschenrechner mit Speicherfunktion

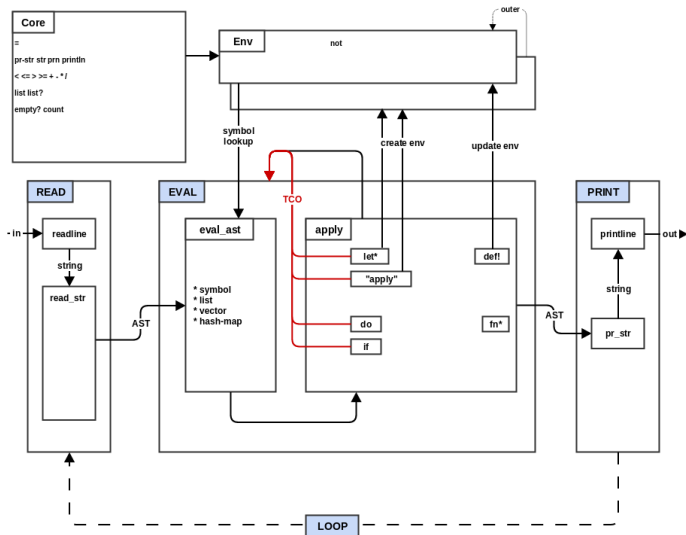
Schritt 4



Schritt 4

- `fn*` erzeugt benutzerdefinierte Funktion mit aktuellem Environment, Argument-Liste und Body
- Am einfachsten als Closure implementierbar die ein neues Environment mit Argumenten erzeugt und Body damit evaluiert
- Apply-Phase muss diesen Fall berücksichtigen
- Alternativ EVAL-Sonderfall einführen
- `if` und `do` müssen spezielle Formen sein, da besonderes Verhalten
- Einführung einer Core-Datei für eingebaute Funktionen
- Resultat: Einfache Kontrollstrukturen

Schritt 5



- TCO: Tail Call Optimization
- Endrekursion verhält sich wie eine Schleife
- Vermeiden neuer Stackframes, Rekursionslimit
- Vielfältig implementierbar (GOTO, Trampolin, Cheney on the M.T.A.)
- MAL verwendet eine Schleife in `EVAL` und `continue` für TCO-Fälle, sonst `return`
- Wichtig: Testen dass Code sich mit TCO identisch verhält
- Resultat: Iteration durch Rekursion

TCO-Beispiel

Keine TCO möglich

```
(def! fact
  (fn* [x]
    (if (<= x 1)
        x
        (* x (fact (- x 1))))))
```

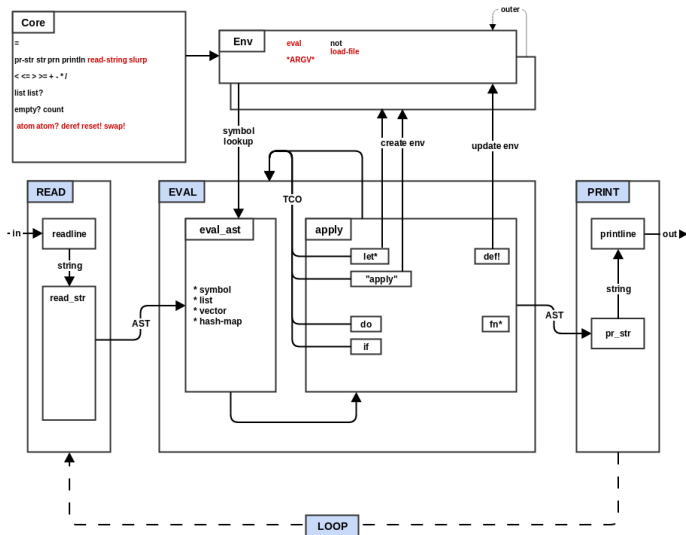
TCO-Beispiel

TCO möglich!

```
(def! fact*  
  (fn* [x acc]  
    (if (<= x 1)  
        acc  
        (fact* (- x 1) (* acc x)))))
```

```
(def! fact  
  (fn* [x]  
    (fact* x 1)))
```

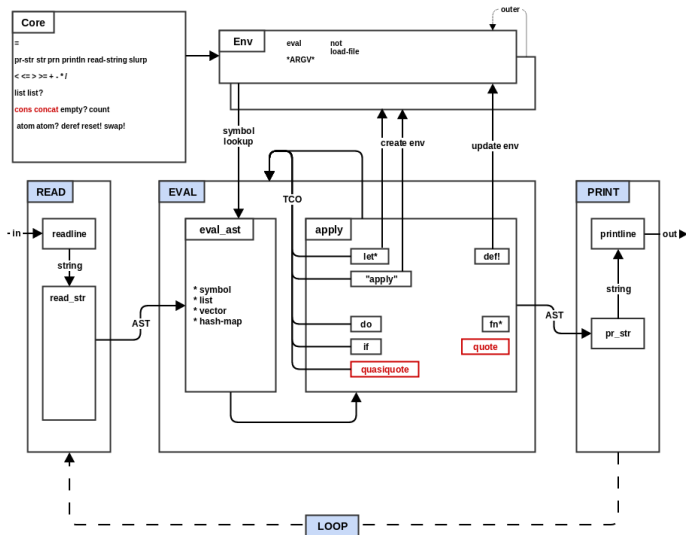
Schritt 6



Schritt 6

- `eval` ist eine Closure die EVAL mit REPL-Environment ausführt
- `read-string`
- `slurp` für das Einlesen einer Datei
- Implementierung von Atoms (mutierbarer State)
- Wichtig: `apply` für `swap!` nötig
- `load-file: (eval (read-string (str "(do" (slurp f) "))))`
- `*ARGV*` und Skriptmodus
- Resultat: Interpreter kann Skripte ausführen

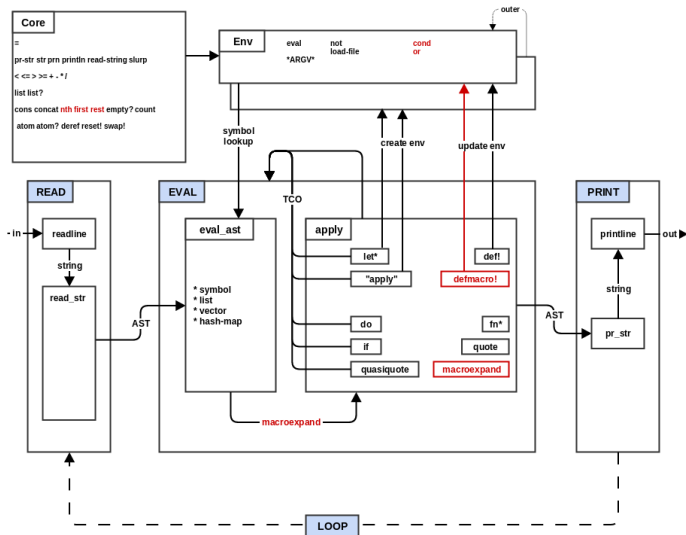
Schritt 7



Schritt 7

- Bisher nicht möglich etwas zum Symbol zu evaluieren
- `quote` gibt das Argument unverändert zurück
- `(quote foo) -> foo`, `(quote (1 2 3)) -> (1 2 3)`
- `quasiquote` erlaubt teilweises Evaluieren der Liste mit `unquote` und `splice-unquote`
- `(let* [x '(2 3)] '(1 ~x)) -> (1 (2 3))`
- `(let* [x '(2 3)] '(1 ~@x)) -> (1 2 3)`
- Listenmanipulation in EVAL
- Resultat: Vorbereitung auf Makros

Schritt 8



Schritt 8

- Benutzerdefinierte spezielle Formen
- Argumente werden nicht evaluiert
- Makro-Aufruf wird durch Expansion ersetzt
- `defmacro!` markiert ein Symbol als Makro
- Expansion geschieht in EVAL als erster Schritt
- `(defmacro! comment (fn* [& body] nil))`
- `(comment (/ 1 0)) -> nil`
- `macroexpand` für Debugging
- Resultat: Makros

Makro-Beispiele

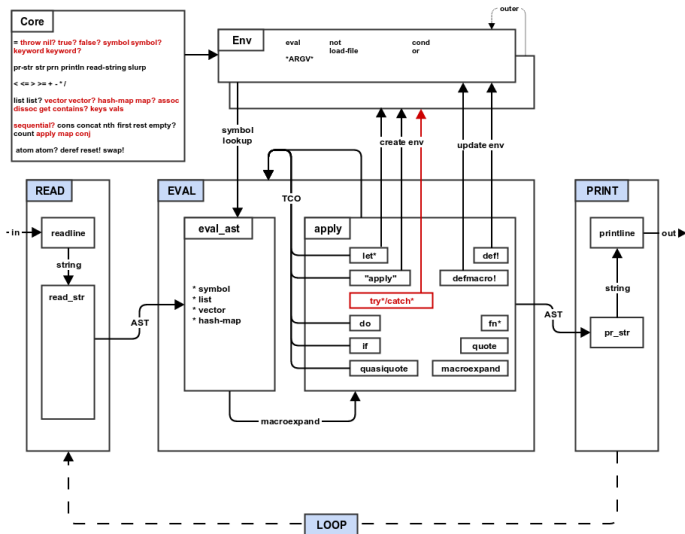
```
(defmacro! when
  (fn* [test & body]
    `(if ~test
        (do ~@body))))
```

Makro-Beispiele

Aus dem Guide entnommen:

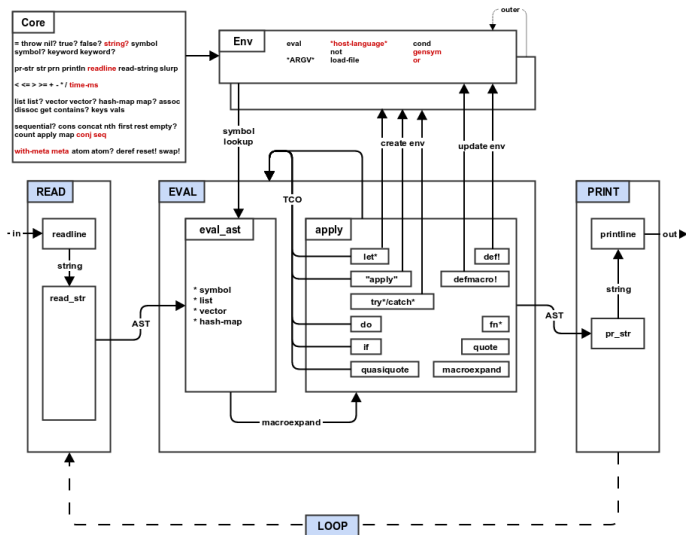
```
(defmacro! or
  (fn* [& xs]
    (if (empty? xs)
        nil
        (if (= 1 (count xs))
            (first xs)
            (let* (condvar (gensym))
                `(let* (~condvar ~(first xs))
                    (if ~condvar
                        ~condvar
                        (or ~@(rest xs))))))))))
```

Schritt 9



- Exception Handling
- `try*` fängt Exceptions, führt im Fehlerfall `catch*` aus
- `throw*` wirft benutzerdefinierte Exception
- Fortführung der Fehlerbehandlung aus `READ`
- Falls Sprache Exceptions unterstützt, trivial, andernfalls mühselig
- Taktiken: Globaler Fehlerstatus (`errno`), Fehlerobjekte
- Implementierung der meisten fehlenden eingebauten Funktionen (inklusive `apply` und `map`)
- Resultat: Brauchbare minimale Programmiersprache

Schritt A



Schritt A

- Restliche Schritte für Self-Hosting
- Metadaten-Support (`.clone`), `readline`, etc.
- Ausführen der Implementierung in MAL im Skriptmodus
- Debugging ist tricky
- Finaler Schritt
- Optional:
 - Performance-Messung (`time-ms`)
 - Interop (`., <lang>-eval`)

Abschnitt 4

Weitere Schritte

- Einreichen einer neuen Implementierung
- Diskussion und Verbesserung von MAL
- Entwickeln einer eigenen Programmiersprache
- PLT
- Byte-Code Interpreter
- Compiler (*ML)
- Scheme, Forth, Smalltalk

- Scheme Implementation Techniques
- Clojure-Sourcen
- SICP
- LiSP

Fragen?