MAL - Making a Lisp

Vasilij Schneidermann

August 2016

Outline

Section 1

Einführung

Sprecher

- Vasilij Schneidermann, 24
- Wirtschaftsinformatikstudent
- Praktikant bei bevuta IT GbmH
- v.schneidermann@gmail.com
- https://github.com/wasamasa
- http://emacshorrors.com/
- http://emacsninja.com/

Motivation

- 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 und ChucK 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

Section 2

MAL

Historie

- 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: 49
- Implementierungen: 57
- Schritte: 11
- Unit Tests: 636

Sprache

- 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 Subrs, 10 Special Forms, 5 in der Sprache definierte Formen
- Hinreichend für Self-Hosting

Code-Beispiele

Aus dem Guide entnommen:

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

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

Code-Beispiele

Triviales Makro:

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

Code-Beispiele

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 ()))))
```

Hilfestellung

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

Community

- IRC-Channel auf Freenode, `#mal`
- GitHub

Section 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

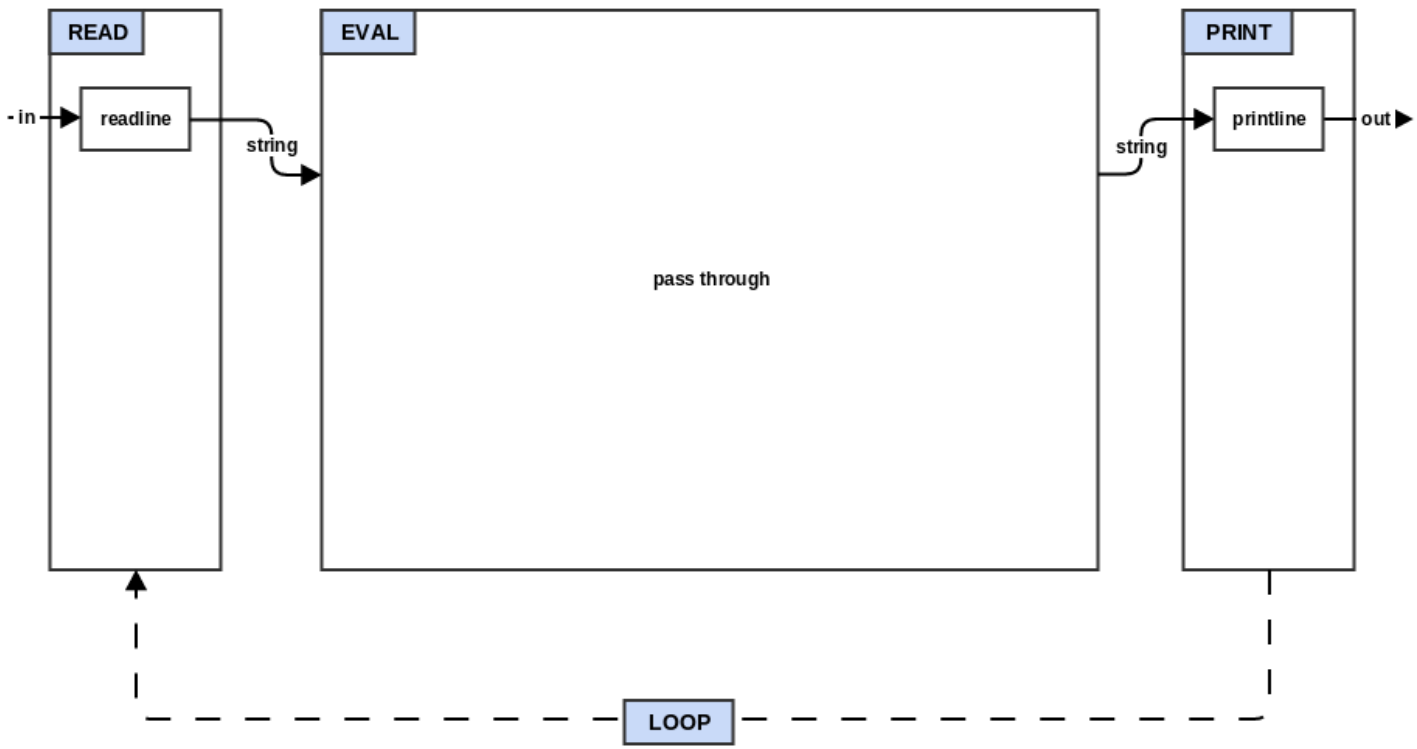
Wünschenswerte Features

- 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

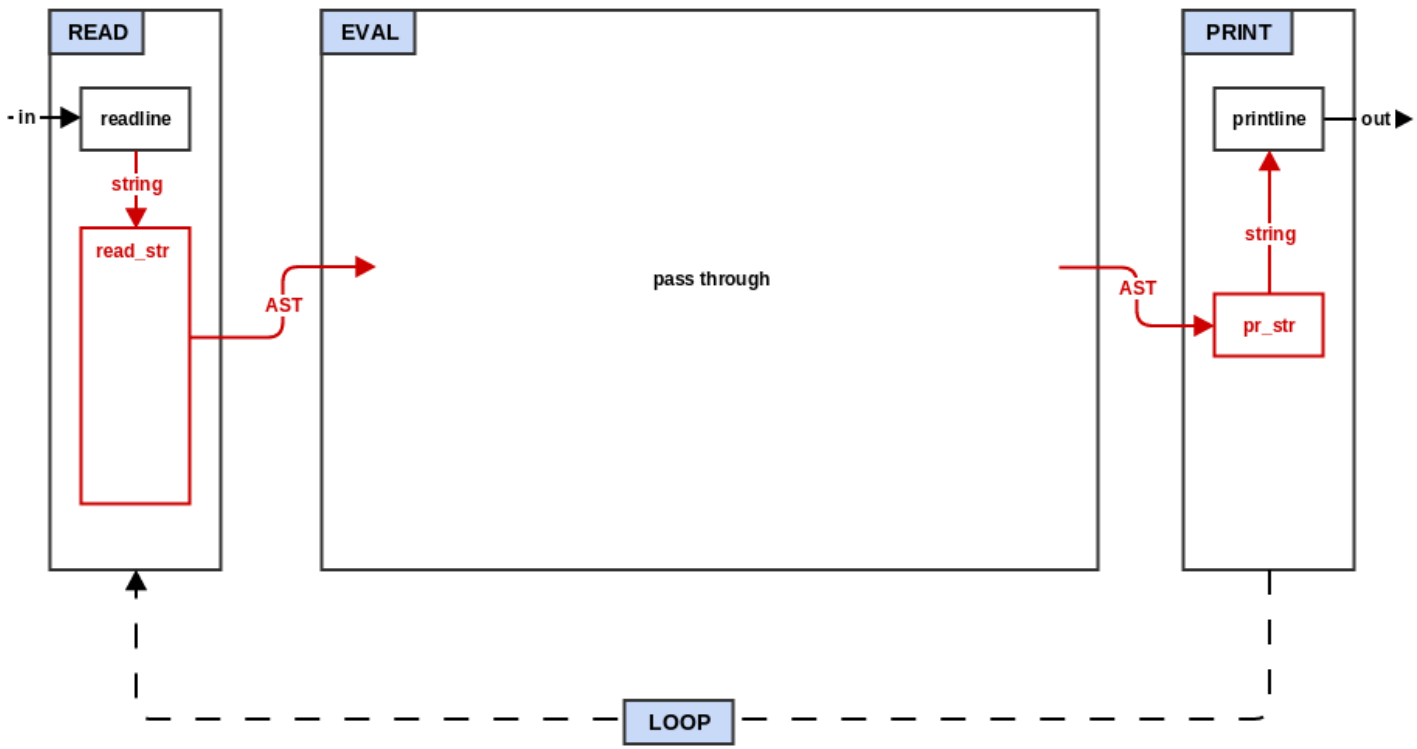
Schritt 0


READ
EVAL
PRINT
- in
readline
string
pass through
string
printline
out
LOOP

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


READ
EVAL
PRINT
- in
readline
string
read_str
AST
pass through
AST
printline
out
string
pr_str
LOOP

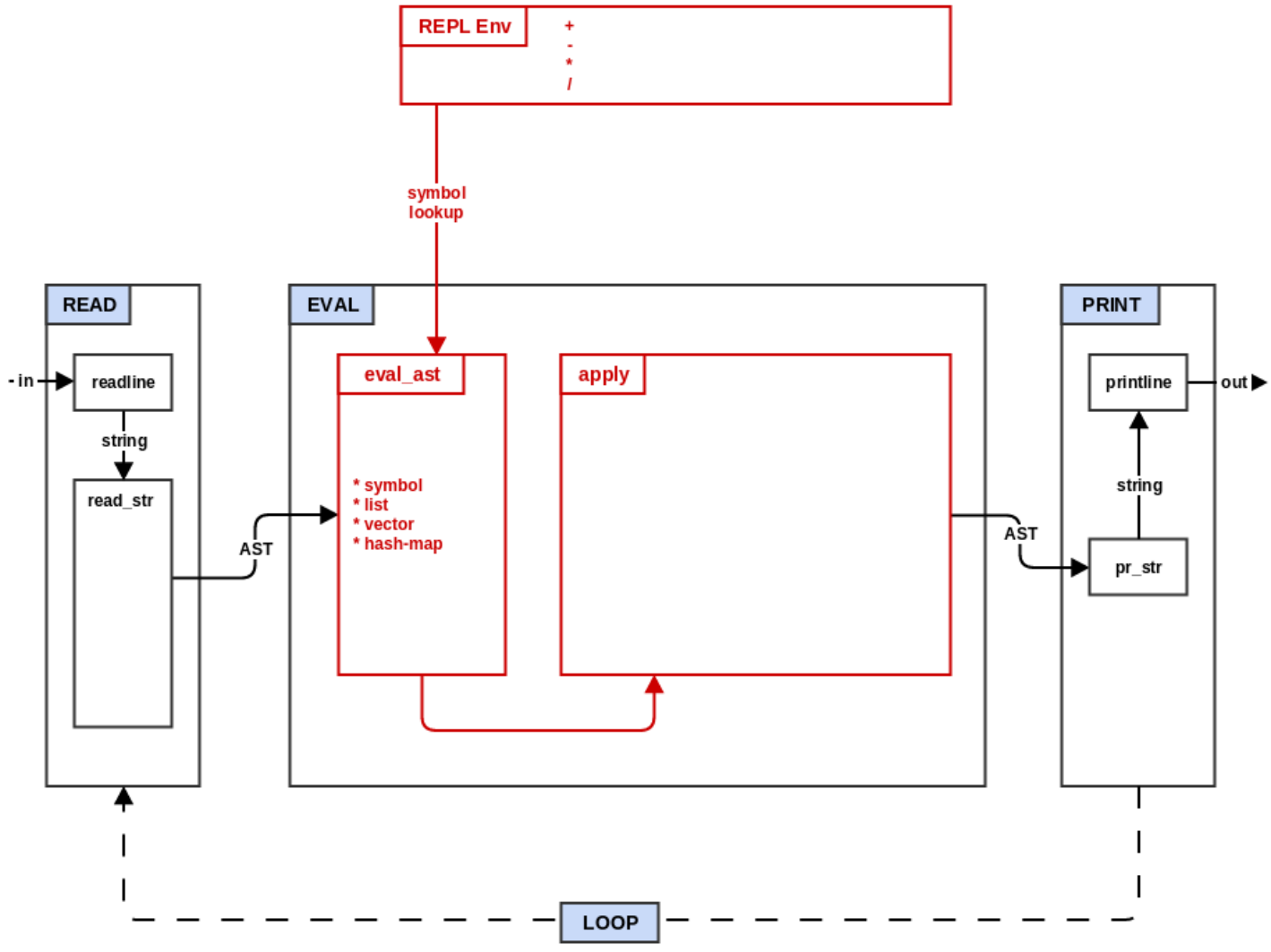
Schritt 1

- `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

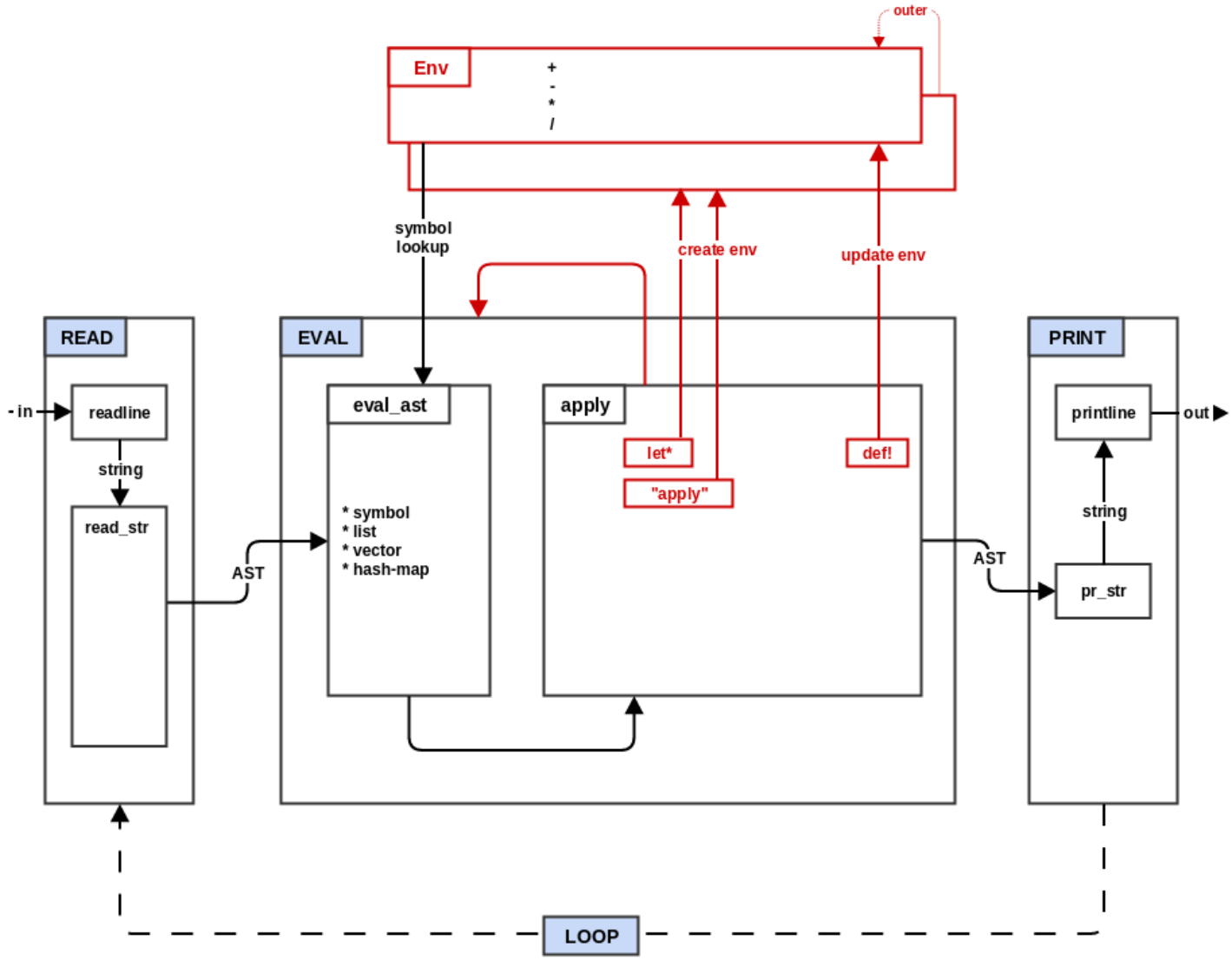
Schritt 2


REPL Env
+
-
*
/
symbol
lookup
READ
EVAL
PRINT
- in
readline
string
read_str
AST
eval_ast
* symbol
* list
* vector
* hash-map
apply
AST
printline
out
string
pr_str
LOOP

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

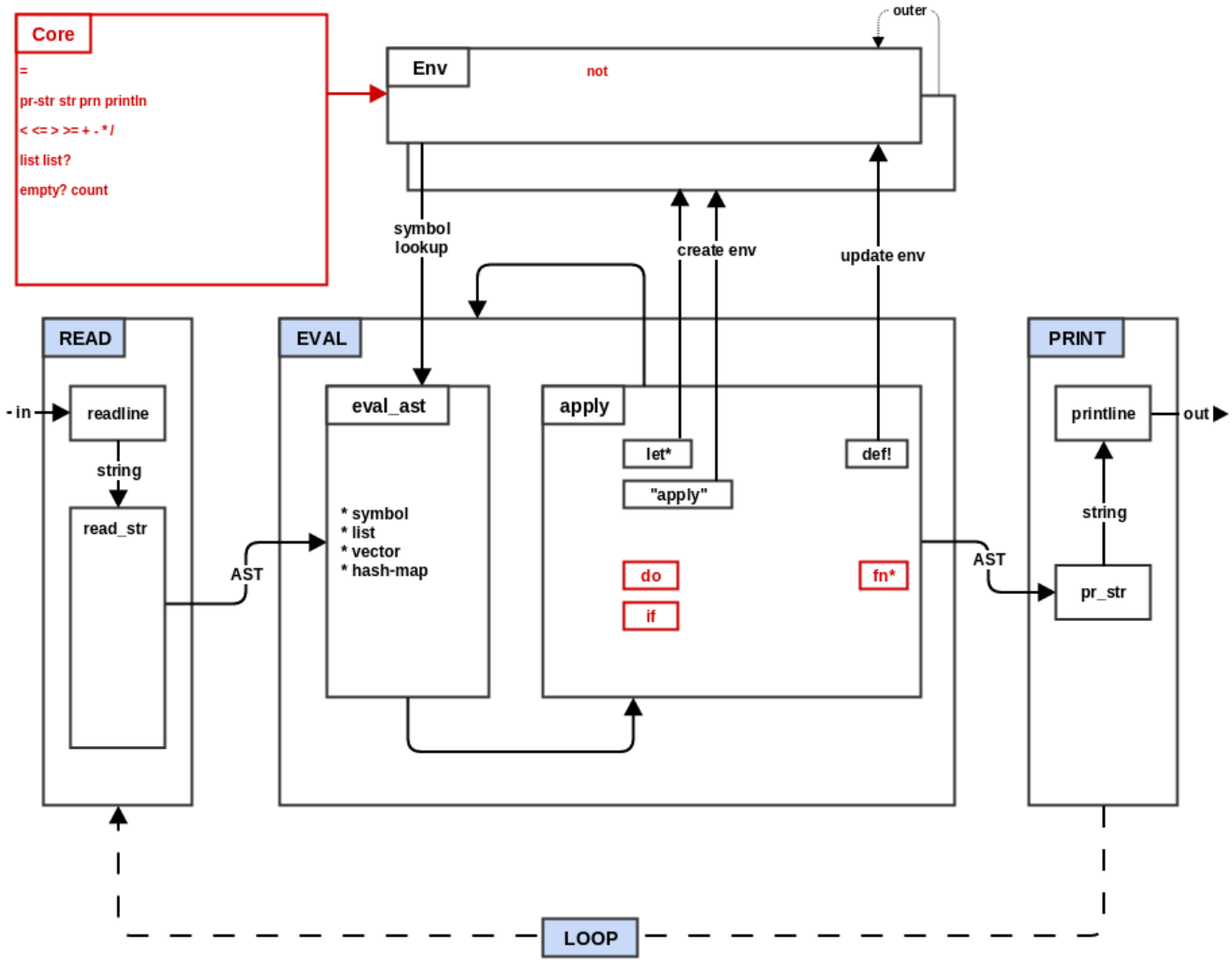
Schritt 3


outer
Env
+
-
*
/
symbol
lookup
create env
update env
READ
EVAL
PRINT
- in
readline
string
read_str
AST
eval_ast
* symbol
* list
* vector
* hash-map
apply
let*
"apply"
def!
AST
printline
out
string
pr_str
LOOP

Schritt 3

- Environment: Outer Environment + Bindings
- Binding: Name -> 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

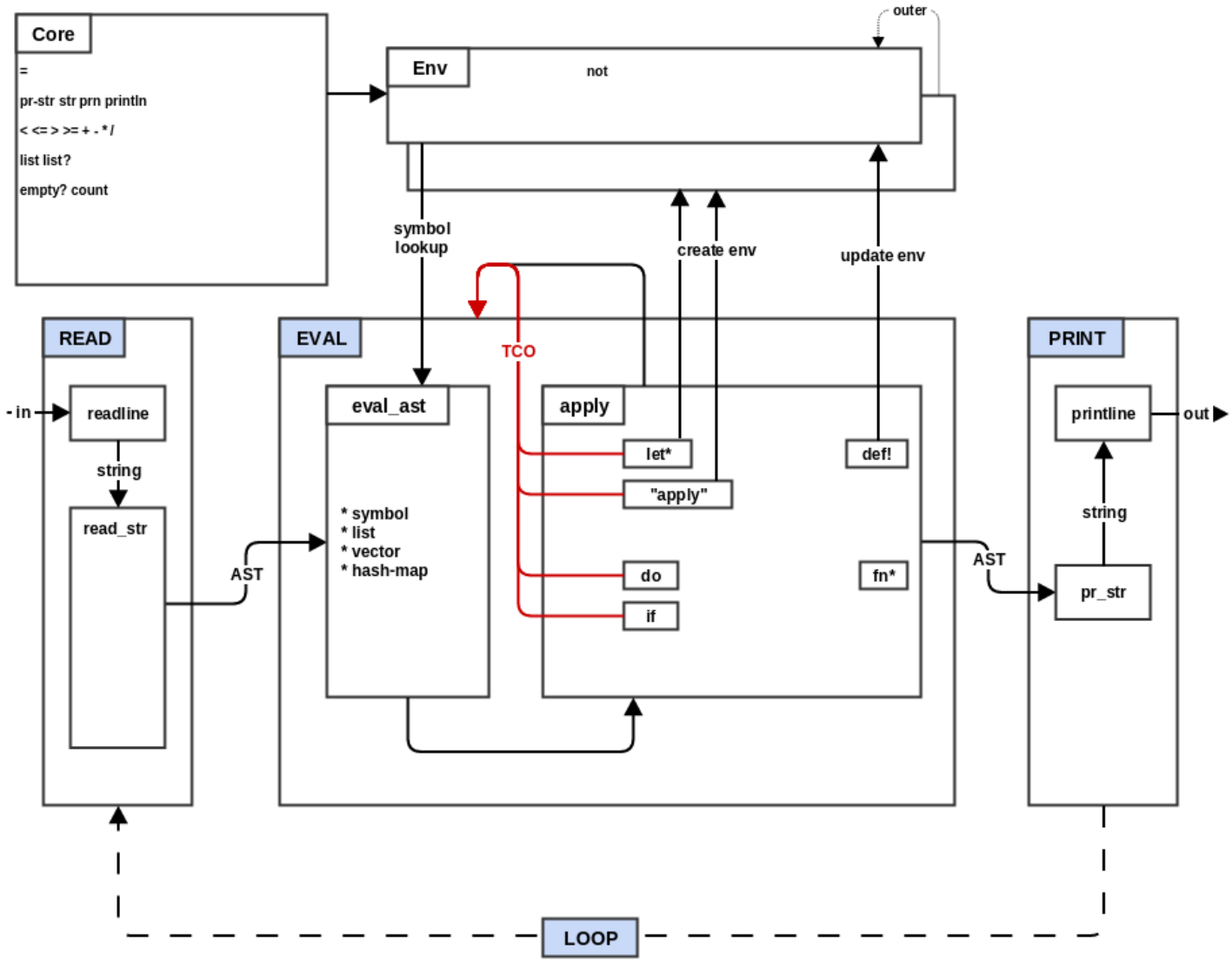
Schritt 4


Core
=
pr-str str prn println
< <= > >= + - * /
list list?
empty? count
Env
not
outer
symbol lookup
create env
update env
READ
- in
readline
string
read_str
AST
EVAL
eval_ast
* symbol
* list
* vector
* hash-map
apply
let*
"apply"
def!
do
if
fn*
AST
PRINT
printline
out
string
pr_str
LOOP

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 mit weiteren Subrs
- Resultat: Einfache Kontrollstrukturen

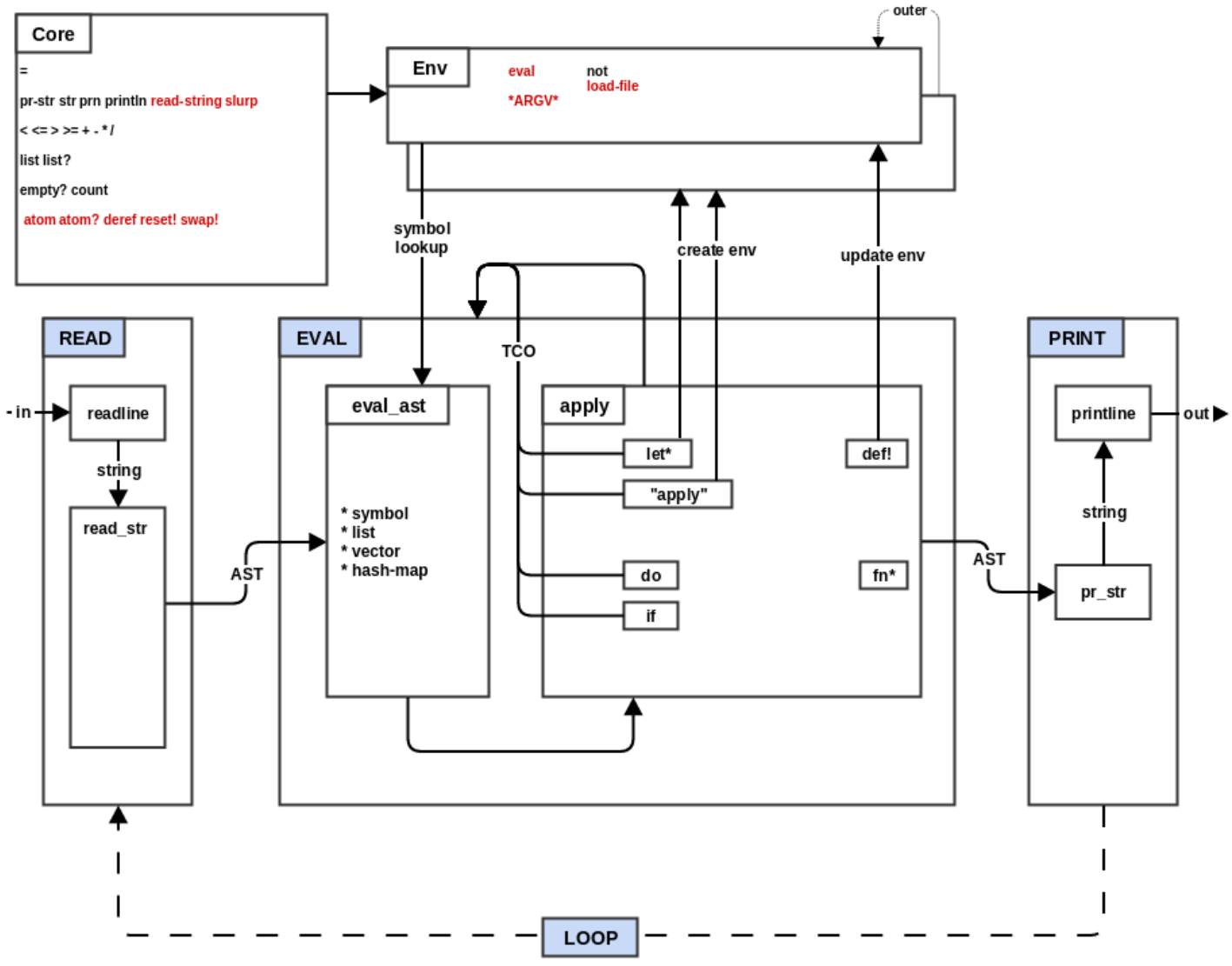
Schritt 5


Core
=
pr-str str prn println
< <= > >= + - * /
list list?
empty? count
Env
not
outer
symbol lookup
create env
update env
TCO
READ
- in
readline
string
read_str
AST
EVAL
eval_ast
* symbol
* list
* vector
* hash-map
apply
let*
"apply"
do
if
def!
fn*
AST
PRINT
printline
out
string
pr_str
LOOP

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

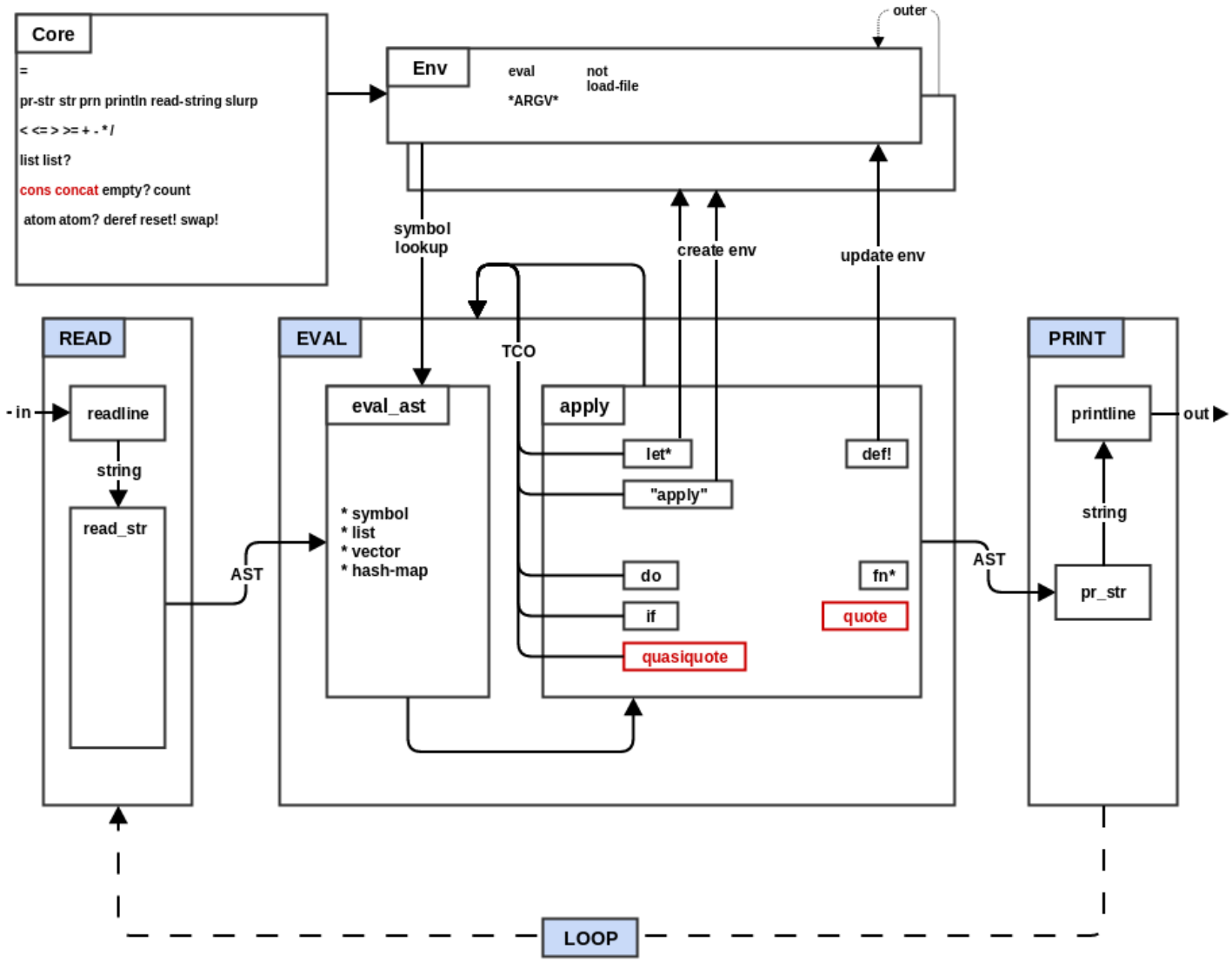
Schritt 6


Core
=
pr-str str prn println read-string slurp
< <= > >= + - * /
list list?
empty? count
atom atom? deref reset! swap!
Env
eval
not
load-file
ARGV
outer
symbol
lookup
create env
update env
READ
- in
readline
string
read_str
AST
EVAL
TCO
eval_ast
* symbol
* list
* vector
* hash-map
apply
let*
"apply"
do
if
def!
fn*
AST
PRINT
printline
out
string
pr_str
LOOP

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

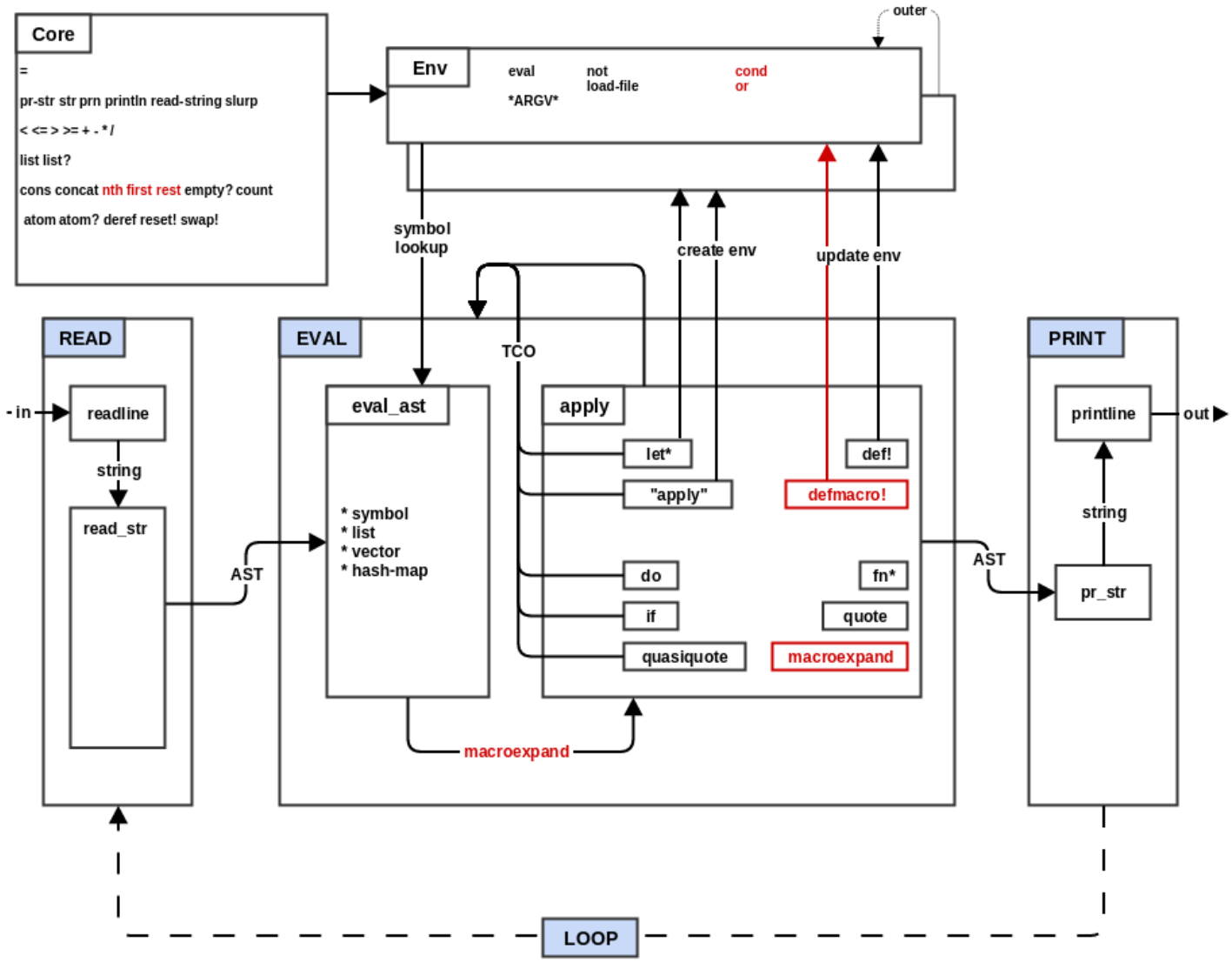
Schritt 7


Core
=
pr-str str prn println read-string slurp
< <= > >= + - * /
list list?
cons concat empty? count
atom atom? deref reset! swap!
Env
eval
not
load-file
ARGV
outer
symbol
lookup
create env
update env
TCO
READ
- in
readline
string
read_str
AST
EVAL
eval_ast
* symbol
* list
* vector
* hash-map
apply
let*
"apply"
def!
do
fn*
if
quote
quasiquote
AST
PRINT
printline
out
string
pr_str
LOOP

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

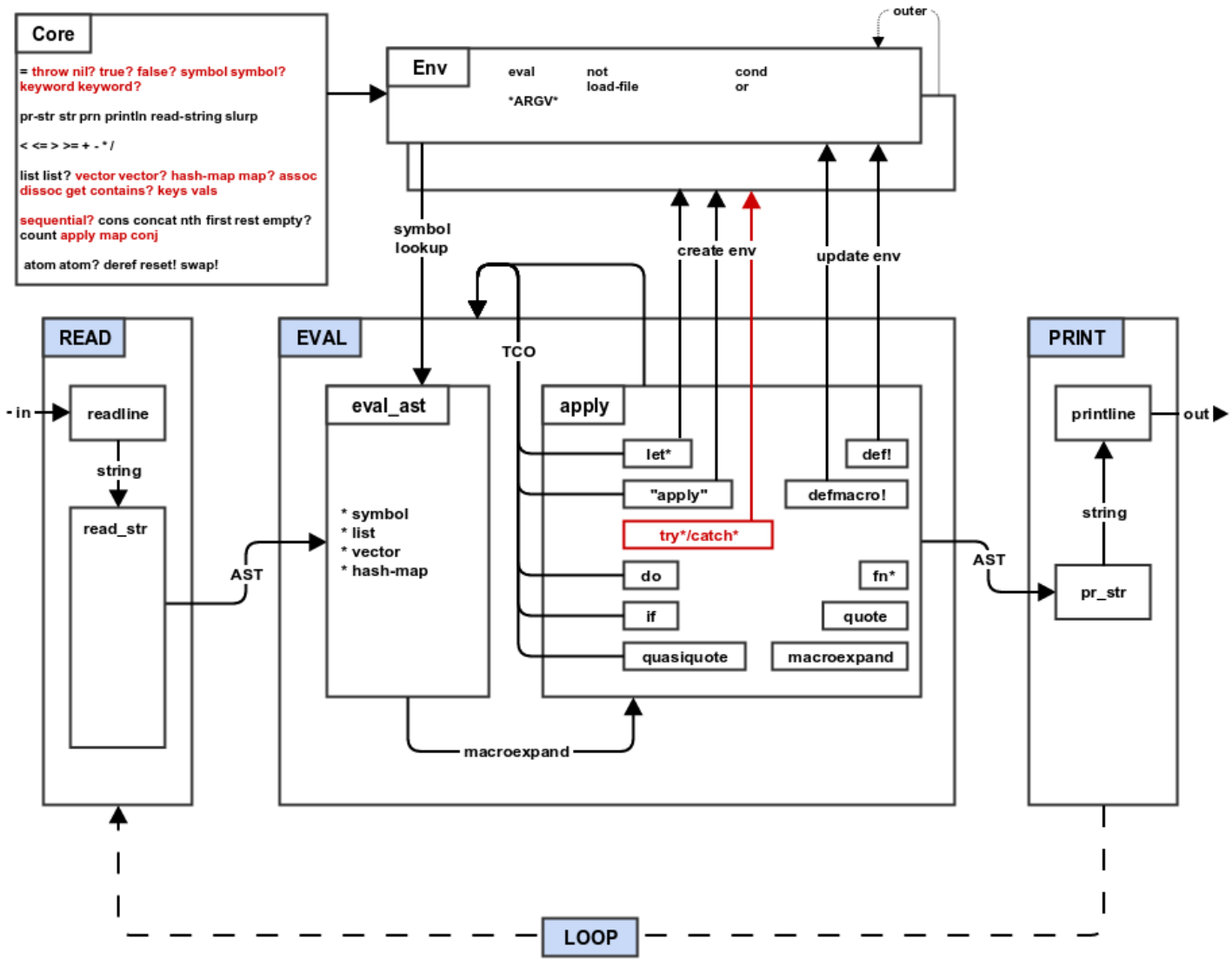
Schritt 8


Core
=
pr-str str prn println read-string slurp
< <= > >= + - * /
list list?
cons concat nth first rest empty? count
atom atom? deref reset! swap!
Env
outer
eval
not
load-file
cond
or
ARGV
symbol lookup
create env
update env
READ
EVAL
PRINT
TCO
- in
readline
string
read_str
AST
eval_ast
* symbol
* list
* vector
* hash-map
apply
let*
"apply"
def!
defmacro!
do
if
quasiquote
fn*
quote
macroexpand
macroexpand
AST
printline
out
string
pr_str
LOOP

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

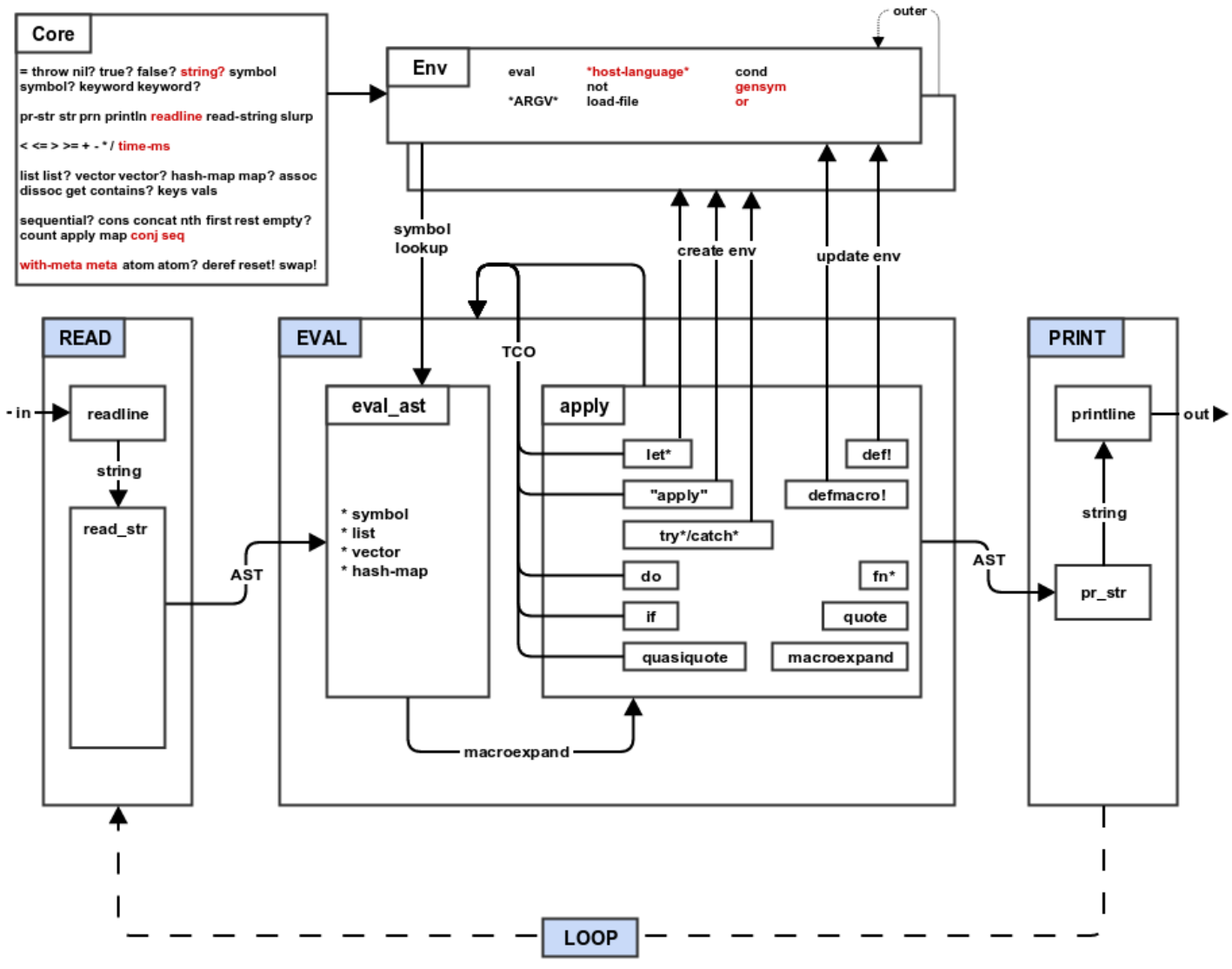
Schritt 9


Core
= throw nil? true? false? symbol symbol? keyword keyword?
pr-str str prn println read-string slurp
< <= > >= + - * /
list list? vector vector? hash-map map? assoc dissoc get contains? keys vals
sequential? cons concat nth first rest empty? count apply map conj
atom atom? deref reset! swap!
Env
outer
eval
not
cond
load-file
or
ARGV
symbol lookup
create env
update env
READ
EVAL
PRINT
TCO
- in
readline
string
read_str
AST
eval_ast
* symbol
* list
* vector
* hash-map
apply
let*
"apply"
try*/catch*
do
if
quasiquote
def!
defmacro!
fn*
quote
macroexpand
macroexpand
AST
printline
out
string
pr_str
LOOP

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 Subrs (inklusive `apply` und `map`)
- Resultat: Brauchbare minimale Programmiersprache

Schritt A


Core
= throw nil? true? false? string? symbol
symbol? keyword keyword?
pr-str str prn println readline read-string slurp
< <= > >= + - * / time-ms
list list? vector vector? hash-map map? assoc
dissoc get contains? keys vals
sequential? cons concat nth first rest empty?
count apply map conj seq
with-meta meta atom atom? deref reset! swap!
Env
outer
eval
ARGV
host-language
not
load-file
cond
gensym
or
symbol
lookup
create env
update env
READ
EVAL
PRINT
TCO
- in
readline
string
read_str
AST
eval_ast
* symbol
* list
* vector
* hash-map
apply
let*
"apply"
try*/catch*
do
if
quasiquote
def!
defmacro!
fn*
quote
macroexpand
macroexpand
AST
printline
out
string
pr_str
LOOP

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`)

Section 4

Weitere Schritte

Weitere Schritte

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

Lesematerial

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

Fragen?