Rückblick

Dominik Keller, G4a

1 Produkt

Das Produkt ist auf Github gehostet und unter dieser URL erreichbar: https://www.github.com/domse007/dlisp. Das finale Produkt befindet sich auf dem `master` Branch.

2 Anforderungen

Das Ziel ist es eine möglichst vollständigen Lisp Core implementierung zu schreiben. Dieser Umfasst das Generieren eines AST und die anschliessende Ausführung des Trees. Core soll bedeuten, dass die wichtigsten Lisp Funktionen verfügbar sind, welche alle in Rust geschrieben sind. Das bedeutet, es soll keine Standard Library entstehen.

Der Evaluator soll Variablen unterstützen, weshalb dieser auch Special Forms unterstützen musss. Special Forms ist ein Sammelbegriff für interne Funktionen, die aussehen wie Lisp Funktionen, aber keine sind.

Die Implementierung soll ohne externe Crates kompilierbar sein. Zudem soll das ganze Crate ohne unsicherer (Note: `unsafe` Keyword) Code geschrieben werden.

3 Implementierung

Die Lisp Implementierung funktioniert in den meisten Fällen. Es gibt allerdings noch einige Probleme. Das grösste ist sicherlich, dass es keinen Macro Support gibt.

3.1 Eval

Eval ist eine Rust Funktion, welche sich rekursiv aufruft und nach und nach alles auflöst, bis sie **einen** Rückgabewert produziert hat. Das erste, was eine Funktion überprüft ist, ob die Eingabe gequoted ist. Gequoted heisst, dass der Rückgabewert dem Eingabewert ist und somit die Daten nicht verändert werden. Danach wird der Typ des `LispObjects` überprüft. Wenn es ein Symbol ist, wird überprüft ob der Manager eine Variable besitzt. Falls ja, wird das `Symbol` mit dem Wert ersetzt. Interessant wird es bei einer liste. Listen sind, wenn sie nicht gequoted sind, Funktionsaufrufe, wobei das erste Element der Funktionsname und der Rest die Argumente für die Funktion sind. Für die Ausführung muss allerdings überprüft werden, auf welchem Level die Funktion defininiert wurde:

1. `Rust Core`: Funktionen wie Additionen und Subtraktionen sind in Rust implementiert und verhalten sich wie Funktionen die in Lisp geschrieben sind.

2. `Special Forms`: Gewisse Funktionen können nicht in der standard Lisp Funktionsweise definiert werden. Diese fallen unter die Special Forms. Auch diese müssen in Rust geschrieben werden. Die Folgenden Special Forms wurden implementiert:

   (a) `defun`: Defun definiert eine Funktion. Diese wird im Manager als Key-Value Pair gespeichert, wobei der Key der Funktionsname ist und der Body die Value.

   (b) `set`: Set nutzt den manager und setzt ein Key-Value Pair. Für diese Operation wird der Manager genutzt, welcher für normale Funktionen nicht zur Verfügung steht.

   (c) `quote`: Diese Special Form setzt das `Quoteflag`.

3. `Lisp Funktionen`: Diese Funktionen werden durch `defun` gesetzt. Und genau da ist der Fehler, denn defun müsste keine special Forms sein, sondern im besten Fall ein Makro. In der aktuellen Implementierung funktioniert es mehrheitlich. Der Rückgabewert einer Funktion ist dabei das Resultat des letzten `eval` Aufrufs. Probleme gibt es zur Zeit noch bei den Parmatern. Diese sind aus zeitlichen Gründen noch nicht implementiert.

4 Rückblick

Es war vom Zeitaufwand, vor allem in dieser Zeit, ein doch viel stressigeres Projekt, als ich anfangs angenommen habe. Schlussendlich habe ich die letzten Bugs in den letzten Stunden vor der Abgabe beheben können, weshalb ich nicht ausführöich testen konnte, ob es Bedingungen gibt (und wenn ja wieviele?), auf die der Interpreter aufgibt.

Zudem hatte ich nicht die Zeit um die Errors umzustellen. Begonnen habe ich mit `&'static str` als Error Typ, denn diese sind schnell geschrieben und trotzdem hilfreich beim Debuggen. In einem weiteren Schritt würde es darum gehen auf `LispError` umzusteigen, denn der sollte für den Nutzer deutlich aufschlussreicher sein, denn er erlaubt es mehr Informationen zu speichern und ausgeben.

Das Projekt nutzt zudem keine `unsafe` Codeblöcke. Das wenn im nächsten Schritt das Error Handling finalisiert wird, kann das Programm nicht mehr abstürzen.

Schlussendlich bin ich aber ziemlich zufrieden mit der Implementierung, denn ich habe alles ohne externe Tutorials oder anderen Resourcen entwickelt. Alles was ich brauchte war `Emacs` mit `lsp-mode` als Interface für `rust-analyzer` und den Compiler für Tipps und Fehler.

Abschliessend heisst das, dass alle am Anfang definierten Punkte erfüllt wurden.