Mirko Westermeier

Lisp in Perl: Erwachsen werden durch Interpreterbau

"Jede Mitteilung geistiger Inhalte ist Sprache."
– Walter Benjamin

Zu meiner Zeit hat eine Laufbahn als Entwickler oder Informatiker oft damit begonnen, HTML zu "programmieren". Die Flamewars, die sich daraus entwickelten, ob HTML denn nun eine Programmiersprache sei oder nicht, waren abendfüllend. Nach einer gängigen Definition nennt man nämlich nur die formalen Sprachen auch Programmiersprachen, die *turing-vollständig* sind, also in der Lage sind, unter der Annahme eines unbegrenzten Speichers eine Turing-Maschine zu emulieren. Mit einer solchen Programmiersprache sind dann alle Berechnungen durchführbar, die allgemein als berechenbar angesehen werden. HTML ist aber nicht in der Lage, entsprechende Berechnungsvorschriften zu formulieren und daher keine Programmiersprache in diesem Sinne.

Jedoch ist HTML ein typisches Beispiel für eine strukturierte Art, Informationen auszutauschen, wie sie dem Programmierer im Alltag ständig begegnen. Überall werden CSV-Dateien gelesen, Websites maschinell auf Aktualisierungen überprüft, JSON-Objekte in den Speicher gelesen oder Shell-Eingaben interpretiert. All diesen Formaten ist gemein, dass sie sich auf klar definierte Art beschreiben lassen, wodurch ihre maschinelle Verarbeitung überhaupt erst möglich wird. Die Mechanismen, die dabei verwendet werden, ähneln teilweise denen aus dem Compilerbau, wenn es also darum geht, einen Programmtext einzulesen und in ausführbaren Maschinencode zu übersetzen oder direkt zu interpretieren.

Manche Menschen nennen eine Programmiersprache dann "erwachsen", wenn sich in ihr auf angenehme Weise ein Interpreter für dieselbe Sprache programmieren lässt, der sich selbst ausführen kann. Mit diesem Hintergedanken kann man es als Meilenstein für die Entwicklung eines Programmierers ansehen, wenn er einen Interpreter für eine Programmiersprache programmiert. Die Implementierungssprache und die interpretierte Sprache sind dabei nicht von zentraler Bedeutung. Viel interessanter sind die verwendeten Methoden und die getroffenen Design-Entscheidungen, denn die daraus gewonnenen Erkenntnisse lassen sich auf viele alltägliche Probleme anwenden. Mit diesem Artikel möchte ich alle Programmierer, die dies noch nicht getan haben, motivieren, einmal einen Interpreter für eine Programmiersprache zu schreiben. Es soll ihr Schaden nicht sein!

Überblick über einen Lisp-Interpreter

Der hier vorgestellte Interpreter erhebt keinen Anspruch auf überragende Effizienz oder Eleganz, kann aber meiner Hoffnung nach ein wenig Inspiration vermitteln, eine eigene Lösung für das gleiche Problem zu entwickeln: Programme formuliert in einer interessanten Programmiersprache auszuführen. Meine Wahl fiel dabei auf Perl als Implementierungssprache, weil sich die Ideen durch die vielfachen Ausdrucksmöglichkeiten der Sprache genau so schreiben lassen, wie sie in meinem Kopf formuliert waren. Außerdem ist Perls Mächtigkeit in String-Verarbeitung im Interpreterbau von großem Vorteil. Die zu implementierende Sprache ist Lisp, einerseits aus Gründen der Ehrfurcht vor diesem Urgestein schierer Eleganz, aber auch weil Lisp mit extrem wenig Syntax auskommt und so das Parsen des Programmtexts sehr einfach wird. Mit Lisp ist hier übrigens kein bestimmter standardisierter Dialekt der großen Lisp-Sprachfamilie gemeint. Ein Beispiel von der Eingabeaufforderung des Interpreters:

```
-> (define (square x) (* x x))
Function: (x) -> (* x x)
-> (square 42)
1764
```

Zentrale Datenstruktur der Programmierung in Lisp ist die Liste (rekursiv definiert über Kopf und Restliste). Wertet man eine Liste aus, so wird versucht, den Listenkopf als Funktion oder Operator aufzulösen und die restlichen Listeneinträge als Argumente zu übergeben. In obigem Code wird eine Quadrierungsfunktion an den Namen `square` gebunden. Der Aufruf in Form einer Liste führt dann zur Anwendung dieser Funktion auf das Argument (hier: die Zahl `42`).

Komplexe Programme lassen sich dann als Kombination von Operatoren und Funktionen zu weiteren Funktionen ausdrücken. Die Ausführung wird jeweils durch Auswertung eines Ausdrucks durchgeführt. Der Interpreter besteht daher im Wesentlichen aus einer Eingabeaufforderung, die bis zur Eingabe eines Beenden-Befehls die Eingaben im Kontext der vorherigen Geschehnisse interpretiert und das Ergebnis der Auswertung ausgibt, dem sogenannten Read-Eval-Print-Loop.

Herzstück des Read-Eval-Print-Loops ist der Aufruf einer `eval`-Methode des Interpreters, die zu einem gegeben Programmtext die generierten Rückgabewerte als Text zurückgibt (Der hier teilweise gekürzt gezeigte Quelltext lässt sich übrigens im *github-Repository|https://github.com/memowe/perlisp* nachlesen):

```
sub eval {
    my ($self, $string) = @_;

    # lex
    my $token_stream =
       $self->lexer->lex($string);

    # parse
    my @exprs =
      $self->parser->parse($token_stream);

    # eval
    my @values = map {
       $_->eval($self->context) } @exprs;

    # return
    return wantarray ? @values :
      shift @values;
}
```

Interpreter's Anatomy

Aus der Struktur der `eval`-Methode wird unmittelbar die Struktur des Auswertungsvorgangs ersichtlich:

• Der **Lexer** wandelt den als String übergebenen Programmtext in eine Folge von sogenannten Token um, also in Objekte, deren Typen durch die formale Beschreibung der Sprache festgelegt sind. Diese Objekte können mit einem Attribut versehen sein. Beispielsweise wird der String `"("` in das Token `LIST_START` übersetzt oder der String `42` in das Token `NUMBER` mit dem Attribut 42, also dem Zahlenwert selbst.

• Auf der Basis der Reihenfolge der Token findet im **Parser** die syntaktische Analyse des Programmtextes statt. Dabei werden zusammengehörige Token in geschachtelte Expression-Objekte zusammengefasst, den sogenannten Syntaxbaum.

• Jeder Knoten des Syntaxbaumes besitzt eine `eval`-Methode, die jeweils auf der Auswertung eventueller Teilausdrücke beruht. Nur bei atomaren Ausdrücken wie etwa Zahlen oder Strings ist der Rückgabewert dieser Methode die Zahl bzw. der String selbst.

Führt man die `eval`-Methode eines solchen Syntaxbaumes aus, wird also der gesamte zu interpretierende Ausdruck schrittweise ausgewertet. Der am Ende resultierende Rückgabewert der Auswertung wird an die Eingabeaufforderung zurückgeliefert.

Komplexe Ausdrücke auswerten

Am besten kann man das oben skizzierte Vorgehen an einem Beispiel nachvollziehen. Unter der Annahme, dass die oben definierte `square`-Funktion bereits bekannt ist, soll der folgende Ausdruck ausgewertet werden:

```
(- (square 42) 22)
```

Der Lexer erzeugt daraus die Tokenfolge:

```
LIST_START, SYMBOL(-),
  LIST_START, SYMBOL(square),
    NUMBER(42), LIST_END, NUMBER(22),
      LIST_END
```

Schließlich baut der Parser den folgenden Syntaxbaum bestehend aus Expression-Objekten auf:

```
          (-)
         /   \
    (square)  (22)
        |
      (42)
```

Aus den im Kontext bekannten Bindungen für – und `square` ist bekannt, was mit den Argumenten geschehen soll. Bei der Auswertung des ersten Operanden von – muss zunächst der Funktionsrumpf von `square` eingesetzt werden:

```
          (-)
         /   \
       (*)   (22)
      /   \
    (42)  (42)
```

Die Auswertung von unten nach oben ergibt schließlich die gewünschte Subtraktion `42 * 42 - 22`. Das Ergebnis `1742` wird zurückgeliefert.

Bei der Auswertung werden jeweils die `eval`-Methoden der Expression-Objekte des Syntaxbaumes aufgerufen, die bei atomaren Ausdrücken besonders einfach sind (nämlich das Expression-Objekt selbst zurückliefern), bei Funktionsaufrufen in Form von Listenauswertungen hingegen die Auswertung verschachtelter Teilausdrücke auslösen (vereinfacht):

```
sub eval {
    my ($self, $context) = @_;

    # get function expression and
    # arguments
    my $fn_expr = $self->car;
    my @args    = @{$self->cdr->exprs};

    # eval function expression
    my $function =
      $fn_expr->eval($context);
```

In diesem Fall muss der ausgewertete Kopf der Liste ein Operator oder eine Funktion sein, also eine `apply`-Methode bieten, um auf Argumente (die ausgewerteten Einträge der restlichen Liste) anwendbar zu sein:

```
 # check apply-ability (duck typing)
 die $fn_expr->to_string .
   " can't be applied.\n"
     unless $function->can('apply');

 # apply
 return $function->apply($context, \@args);
 }
```

Im obigen Beispiel wird diese Methode mehrfach ausgeführt, nämlich immer dann, wenn eine Liste ausgewertet und damit ein Operator oder eine Funktion auf Argumente angewendet werden soll. Also für die Subtraktion mit dem Operator – für den Aufruf der Funktion `square` sowie für die darin angegebene Multiplikation mit dem Operator `*`.

Interessante Fragen

• In PerLisp werden Argumente von Funktionsaufrufen standardmäßig ausgewertet bevor sie der Funktion übergeben werden, PerLisp ist also *strikt*. Nicht-striktes Vorgehen ist aber nicht nur nützlich (wie etwa bei der Kurzschlussauswertung von Perls `or`-Operator, sondern auch absolut nötig: Die bedingte Ausführung mittels eines *if-then-else*-Konstrukts etwa ist völlig sinnlos, wenn beide Zweige immer ausgewertet werden. Wie diese Unterscheidung konkret realisiert wird, ist eine interessante Frage beim Entwurf des Interpreters bzw. des verwendeten Sprachdialekts.

• Bei der Ausführung vom Rumpf einer Funktion ist relevant, aus welchem Kontext eventuell verwendete Bindungen stammen (statischer bzw. dynamischer Scope). Welche Sichtbarkeitsstrategie bei der Implementierung verfolgt wird, kann starken Einfluss auf die Ausführung bestehender Programme haben.

• In der Einleitung wurde erwähnt, dass es erstrebenswert sein kann, wenn ein implementierter Interpreter sich selbst ausführen kann. Bei unterschiedlichen Sprachen ist das natürlich nicht möglich. Die Frage nach der Selbstausführbarkeit kann dennoch beantwortet werden. PerLisp ist nämlich im Stande, einen einfachen Lisp-Interpreter (mit dynamischem Scope) auszuführen, der sich selbst ausführen kann. Der zugehörige Code ist in der Testsuite von PerLisp verfügbar.

Frisch ans Werk!

Die hier skizzierten Strategien zur Implementierung eines eigenen Interpreters sollen nur als Anhaltspunkt verstanden werden und individuelle Lösungen nicht zu sehr einschränken, daher sind die ausgewählten Code-Schnipsel auch möglichst allgemein. PerLisp zu schreiben war für mich eine sehr lehrreiche Erfahrung, die ich nur weiterempfehlen kann. Ich würde mich über eine kurze Rückmeldung freuen, wenn aus diesem Artikel neue Interpreter entstanden sind. Viel Spaß!