Praktische Algorithmen der Bioinformatik und Computerlinguistik mit Lisp

Teil 2: Einführung

Tilman Becker

13.5.2021

Übersicht

- Was ist Lisp?
- Geschichtliches
- Lisp im Vergleich zu anderen Sprachen
- Auswertung von Lisp-Ausdrücken
- Listen
- Interne Darstellung von Listen
- Repräsentation anderer Datenstrukturen durch Listen

Außerdem: Einführung in den Editor EMACS (Übungen)

Was ist Lisp?

LISP - LISt Processor

- Lisp ist (ursprünglich und primär) eine “listen-orientierte” Sprache
- Listen dienen zur Darstellung symbolischer Information, so dass Lisp (primär) eine Sprache für die Verarbeitung symbolischer Information ist.
- Lisp ist eine der wichtigsten Programmiersprachen für den Bereich der Künstlichen Intelligenz.
- Lisp ist ursprünglich und primär eine funktionale Sprache, abgeleitet vom Lambda-Kalkül - einem der Modelle für (partiell) berechenbare Funktionen

Was ist Lisp?

LISP - LISt Processor

- Lisp unterstützt jedoch auch prozedurale und objekt-orientierte Programmierstile
- Anfänglich nur KI-Sprache, ist Lisp heute eine Programmiersprache für viele Zwecke (“general purpose”)
- Lisp erlaubt prägnante Formulierung von Programmen und dennoch Ausführungszeiten, die sich mit denen prozeduraler Sprachen vergleichen lassen.

Etwas Geschichte

- von John McCarthy, einem der Väter der Künstlichen Intelligenz, um 1958 entwickelt (Lisp ist fast so alt wie Fortran), speziell für “symbolisches Rechnen” im Bereich der KI
- erste Implementierung eines Interpreters um 1960 am Artificial Intelligence Laboratory des MIT (IBM 704)
- Entwicklung umfangreicher und komfortabler Programmierumgebungen
- Entwicklung spezieller Rechnerarchitekturen (Hardware und Software) zur Unterstützung von Lisp: Lisp-Maschinen seit 1980 kommerziell verfügbar
- Entwicklung objekt-orientierter Erweiterungen, insbesondere im Umfeld der Lisp-Maschinen

Etwas Geschichte

- Zusammenführung verschiedener Dialekte in Common Lisp (1984), Standardisierung als ANSI Common Lisp
- Einsatz als Grundlage von Systemen in Nicht-KI-Bereichen, z.B.
  - Editoren: Multics Emacs, Gnu Emacs
  - Statistik-Paket: Lisp-Stat
  - CAD: AutoCAD
  - Publishing-Programme: Interleaf
  - Web-Server
  - Datenbank-Managementsystem

  In allen Fällen ist (eine Version von) Lisp die Sprache, in der Benutzer-Erweiterungen programmiert werden.
- Moderne Lisp-Systeme sind so effizient wie C/C++.
- Es gibt heute praktisch keinen Bereich, einschließlich industrieller Anwendungen, in dem Lisp nicht erfolgreich eingesetzt wird.

Programmier-Stile

Es lassen sich unter anderem folgende Programmier-Stile (Programmier-Paradigmen) unterscheiden. Sie treten meistens nicht in Reinform auf:

- **prozedural/imperativ:** z.B.Java/Pascal typische Strukturen:
  - Programm-Zustand
  - Zuweisung
  - Kontrollstrukturen: Schleifen, bedingte Zuweisungen
  - Prozeduren
- **funktional:** z.B. ML, Miranda, Haskell typische Strukturen:
  - Werte und Ausdrücke
  - Funktionen und Funktionsaufrufe
  - Komposition: Verschachtelung von Funktionsaufrufen
  - Funktionen höherer Ordnung (Funktionen als Argumente)

Programmier-Stile

- **objekt-orientiert** (z.B. Smalltalk, Java, C++, ...)
  typische Strukturen:
  - Klassen und Instanzen
  - Vererbung von Eigenschaften
  - “Nachrichten” bzw. “Methoden” arbeiten auf Klassen und ihren Unterklassen
- **relational** (z.B.Prolog)
  häufig (fälschlich) als “logisches Programmieren” bezeichnet

Wie unterscheidet sich Lisp von anderen Programmiersprachen?

- Unterstützung verschiedener Programmier-Stile/-Paradigmen:
  - Lisp kombiniert funktionale und prozedurale Elemente.
    Sofern möglich, wird ein funktionaler Stil bevorzugt.
  - Spracherweiterung für objekt-orientiertes Programmieren:
    CLOS ist Teil des Common Lisp-Standards.
  - der relationale Programmierstil wird nicht direkt unterstützt, lässt sich aber gut simulieren.
- Extrem einfache Syntax: Hauptsächlich Klammer-Notation (im Gegensatz zu vielen imperativen Sprachen)

Wie unterscheidet sich Lisp von anderen Programmiersprachen?

- Typisierung:
  - Typsystem vorhanden, aber Typisierung wird nicht gefordert und von vielen Compilern nicht vollständig unterstützt
  - Dadurch: Größere Flexibilität (siehe Sprachen wie C)
  - Jedoch: Wenig statische Fehlerkontrolle möglich (im Gegensatz zu stark getypten Sprachen wie Pascal, ML, Eiffel)
- auf interaktive Benutzung ausgerichtet: ein Lisp-System ist primär (auf der obersten Ebene) ein Interpreter, mit dem der Benutzer kommuniziert (gemeinsam mit anderen funktionalen Sprachen, im Gegensatz zu vielen imperativen Sprachen)
- interpretative Semantik: eine Eingabe kann sofort durch einen Interpreter ausgeführt werden

Bemerkungen zur Entwicklungsumgebung: Lisp-Dialekte

In der fast 60-jährigen Lisp-Geschichte sind viele Dialekte entstanden:

- MacLisp, Zetalisp, Interlisp...

Teilweise haben sich daraus eigene Sprachen entwickelt:

- Scheme, Loops,...

In den 80er Jahren haben Standardisierungsbemühungen zur Herausbildung von *Common Lisp* geführt, das jetzt ein ANSI-Standard und die Grundlage praktisch aller Implementierungen ist.

Interpretation

Lisp ist zunächst eine *interpretative* Sprache, d.h. jeder Ausdruck (jede Form) erhält durch Interpretation einen Wert.
Die oberste Ebene des Lisp-Systems ist ein *Interpreter* für Eingaben.

Der Interpreter ist realisiert als eine "`read-eval-print`-Schleife":

Ein Ausdruck wird

- mit `read` eingelesen. `read` verwandelt die externe Repräsentation in eine interne.
- mit `eval` ausgewertet.
- mit `print` wird die interne Repräsentation des Ergebnisses ausgegeben.

Wir beschäftigen uns im folgenden damit, welche Ausdrücke man in das Lisp-System eingeben kann, und wie diese ausgewertet werden.

Bemerkungen zur Entwicklungsumgebung: Common Lisp

Implementierungen von Common Lisp gibt es für alle wichtigen Betriebssysteme, sowohl kommerzielle wie auch freie.

- Steel Bank Common Lisp (SBCL)
- CLISP
- Allegro Common Lisp (entwickelt von FranzInc.)
- LispWorks (entwickelt von LispWorks Ltd.)
- viele mehr (sowohl frei als auch kommerziell)

Allegro Common Lisp und LispWorks sind jeweils mit eingeschränkter Funktionalität frei erhältlich.

Die Common Lisp-Dialekte unterscheiden sich hauptsächlich in Bezug auf ihre Entwicklungsumgebung.

Bemerkungen zur Entwicklungsumgebung: Emacs/Slime - Das Wichtigste auf einen Blick

Der Editor Emacs ist selbst in Lisp implementiert und besonders zur Entwicklung mit Lisp geeignet.

Für die Lisp-Entwicklung kann der Emacs-Mode SLIME verwendet werden.

- Start: `Alt-x slime`
- SLIME unterstützt
  - das Auswerten (in einer REPL) und Kompilieren von Code
  - das Nachschlagen von Dokumentation, z.B.
  - Funktionsdefinitionen
  - Makroexpansion
  - Debugging (mit SLDB)

Weiterführende Mechanismen werden später behandelt.

Syntax und operationelle Semantik

Es gibt – im wesentlichen – nur *Atome* und *Listen*.

Atome sind (u.a.)

Zahlen	(ganze Zahlen beliebiger Größe, rationale Zahlen, reelle Zahlen, komplexe Zahlen)
Zeichen	(engl. Character)
Zeichenreihen	(engl. Strings)
Symbole	dies sind Variablen, denen ein Wert zugewiesen werden kann. Alles, was nicht anderes interpretiert werden kann ist eine Symbol. Groß- und Kleinschreibung spielt keine Rolle. Wertzuweisung geschieht etwa mit dem setf Makro (s.h.)

```
(setf x 4)
```

z.B. definiert die Variable x und weist ihr den Wert 4 zu.

Syntax und operationelle Semantik

Operationelle Semantik: Was ist der Wert eines Atoms?

- Atome, die keine Symbole sind, evaluieren zu sich selbst.
- Symbole evaluieren zu ihrem Wert, falls vorher einer zugewiesen wurde.

Syntax und operationelle Semantik: Bespiele

Beispiel: Auswertung von Konstanten

```
1234567890          ==>   1234567890
6/8                 ==>   3/4
12.23456678         ==>   12.23456678
"hello, world"      ==>   "hello, world"
:hallo              ==>   :hallo
```

Beispiel: Symbole

```
pi                  ==>   3.141592653589793
x                   ==>   >>Error: The symbol X has no
                          global value
```

Syntax und operationelle Semantik: Bespiele

Um eine globale Variable anzulegen, wird `defvar` (oder `defparameter`) verwendet:

```
(defvar x 1) ==> X
x            ==> 1
```

Um ein Symbol auszuwerten, muß zuerst ein Wert zugewiesen werden:

```
(setf x 4)   ==> 4
x            ==> 4
```

Nach der setf Anweisung ist das Symbol x an den Wert 4 gebunden; x ist eine *globale Variable*.

Syntax und operationelle Semantik: Listen

Eine **Liste** hat entweder die Form `nil` oder `(a b c ...)`. Für `nil` kann man alternativ auch `()` verwenden. Programmteile werden durch Listen (“Formen”) repräsentiert: einheitliche Syntax in Präfix-Notation und Funktionskomposition durch Verschachtelung von Listen; **z.B:**

```
(+ 3 4)
(< 3 4)
(+ 2 (* 3 4))
```

Syntax und operationelle Semantik: Listen

Operationelle Semantik: Was ist der Wert einer Liste?
Das erste Element einer Liste wird i.A. als Funktionsname interpretiert. Gibt es keine Funktion dieses Namens, so signalisiert der Interpreter einen Fehler. Die restlichen Elemente der Liste werden von links nach rechts evaluiert, um als Argumente der Funktion übergeben zu werden.

Bemerkung: In Common Lisp gibt es neben normalen Funktionsanwendungen auch sogenannte *Spezialformen* und *Makros*, bei denen die Argumente nicht unbedingt von links nach rechts ausgewertet werden – mehr dazu später.

Quotierung

Manchmal ist man daran interessiert, Ausdrücke vor der Auswertung zu schützen; dies geschieht mit der Spezialform quote.

```
y               ==> >>Error: The symbol Y has no
                    global value

(quote y)       ==> Y
```

quote benötigt genau ein Argument und gibt dieses Argument unmodifiziert und nicht evaluiert zurück.

Häufiger wird die Abkürzung

```
'y              ==> Y
```

benutzt.

Quotierung

In ähnlicher Weise lassen sich andere Lisp-Ausdrücke wie z.B. Listen quotieren:

```
(setf l '(a b c))
```

Beispiel: `nil`, `()`, `'()` und `'nil` evaluieren alle zu `nil`.

Listen

Konstruktion von Listen:

`nil`	Erzeugt leere Liste.
`(cons a l)`	Fügt Element `a` an den Anfang der Liste `l`
`(list a1 a2 …)`	Konstruiert eine Liste der Argumente
`(append l1 l2)`	hängt Liste `l1`, `l2` zusammen

Beispiel:

```
(cons 'a '(b c d))                  ==> (A B C D)
(list 1 (+ 1 1) (+ 1 (+ 1 1)))      ==> (1 2 3)
(append '(1 2 3) '(4 5))            ==> (1 2 3 4 5)
```

Listen

Selektoren auf Listen; z.B:

```
(first nil)          ==> NIL
(first '(a b c))     ==> A
(rest nil)           ==> NIL
(rest '(a b c))      ==> (B C)
(third '(a b c))     ==> C
(nth 2 '(a b c))     ==> C
(last '(a b c))      ==> (C)
```

`car` ist ein älterer Name für `first`, `cdr` ein älterer Name für `rest`.

`(caddr mylist)` steht für `(car (cdr (cdr mylist)))`

Listen und Funktionsaufrufe

Die Unterscheidung zwischen Listenkonstruktion und Funktionsaufruf bereitet anfänglich immer Schwierigkeiten.

- Durch `(f a1 a2)` wird die Funktion `f` auf die Argumente `a1`, `a2` angewendet. Es wird dadurch nicht die Liste mit Elementen `f`, `a1`, `a2` konstruiert!
- Durch `(list expr1 expr2)` wird eine Liste konstruiert, deren Elemente die Werte von `expr1, expr2` sind.
- Durch `'(e1 e2 e3)` wird eine Liste mit Elementen `e1, e2, e3` konstruiert.

Listen und Funktionsaufrufe

Beispiele:

```
> (defun f (n1 n2) (* n1 (+ n2 2)))
F
> (f 3 4)
18
> (f (+ 2 3) 4)
30
> (list 'f (+ 2 3) 4)
(F 5 4)
> '(f (+ 2 3) 4)
(F (+ 2 3) 4)
> '(e1 e2 e3)
(E1 E2 E3)
> (e1 e2 e3)
>>Error: The function E1 is undefined
```

Prädikate

Lisp-Prädikate sind Funktionen, die *logische Werte* zurückliefern. In Lisp wird nil als *falsch* und alles was nicht nil ist als *wahr* interpretiert.

Bemerkung: nil ist also ein Symbol, eine Liste und ein Wahrheitswert.

Prädikate auf Listen:

`(null l)`	Ist `l` die leere Liste?
`(consp l)`	Ist `l` eine zusammengesetzte Liste
`(listp l)`	Ist `l` eine Liste (vom Typ `list`)?
`(member a l)`	Ist a ein Element der Liste `l`?

Beispiel:

```
(null nil)              ==>   T
(null '(1))             ==>   NIL
(member 'a '(a b c)) ==>   (A B C)
```

Prädikate

Zusammengesetzte **Prädikate** lassen sich mit den Standardoperatoren or, and und not realisieren; **z.B**:

```
(and (consp '()) (not (null 'x)))  ==> NIL
(and (not (member 'b nil))
(or (listp '(1 2)) nil))
==> T
```

Prädikate

Weitere Typprädikate:

```
(setf var 10)                  ==> 10
(integerp var)                 ==> T
(numberp var)                  ==> T
(atom 'x)                      ==> T
(atom var)                     ==> T
(symbolp 'x)                   ==> T
(listp var)                    ==> NIL
(typep var 'ratio)             ==> NIL
(typep var '(or number symbol)) ==> T
```

Prädikate

Darüberhinaus gibt es noch eine Unmenge weiterer vordefinierter Prädikate für eingebaute Datentypen; z.B. `=`, `/=`, `<`, `>`, `<=`, `>=` zum Vergleich numerischer Argumente, `char=`, `char<`, `char>` zum Vergleich von Zeichen, `string=`, `string<`, `string>` zum Vergleich von Zeichenreihen, . . .

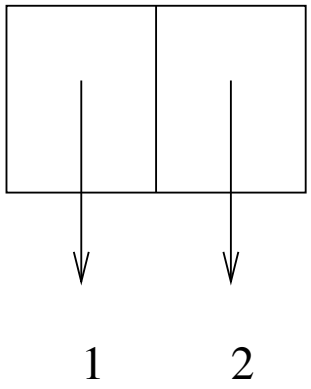
Paare und interne Darstellung von Listen

Eine **cons-Zelle** ist ein Paar von Zeigern; der 1. Zeiger heißt *car*-Zeiger und der zweite heißt *cdr*-Zeiger. Mit cons wird eine cons-Zelle allokiert und mit `car`, `cdr` auf die jeweiligen Bestandteile zugegriffen:

```
(setf  *c* (cons 1 2))
(car   *c*)      ==> 1
(cdr   *c*)      ==> 2
(consp *c*)      ==> T
```

Paare und interne Darstellung von Listen

Graphische Darstellung einer cons-Zelle:



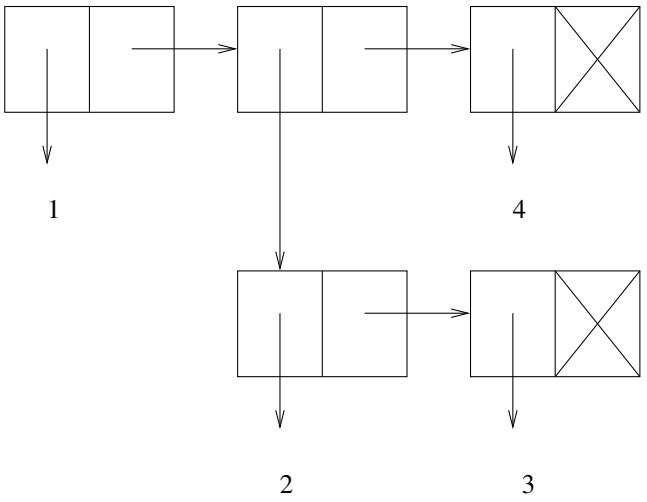
Darstellung obiger cons-Zelle als *Dotted Pair* : (1 . 2) Bemerkung: Die Funktionen `car` und first sowie `cdr` und `rest` sind synonym; die jeweils zweite Form ist aber zu bevorzugen.

Paare und interne Darstellung von Listen

Listen werden intern dargestellt als cons-Zellen; z.B. sind die Formen

```
'(1 (2 3) 4)
(cons 1 (cons (cons 2 (cons 3 nil))
(cons 4 nil)))
'(1 . ((2 . (3 . nil)) . (4 . nil)))
```

alle äquivalent.



Paare und interne Darstellung von Listen

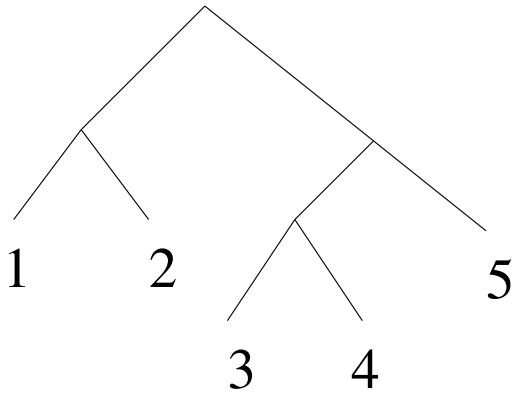
Beachte: `(a . (b . (c . nil)))` repräsentiert eine Liste, während `(a . (b . (c . d)))` keine Liste repräsentiert!

Vorsicht: Das Prädikat `listp` wird beide Varianten als Liste identifizieren.

Bemerkung: Der Lisp-Benutzer/Programmierer muß sich um Speicher-Allokation und -Freigabe nicht kümmern – das übernimmt Lisp.

Darstellung von Bäumen durch Listen

Die interne Darstellung von Listen legt eine Darstellung von Bäumen nahe (diese Bäume sind binär und tragen Werte nur an den Blättern):



Dieser Baum kann durch die Ausdrücke

```
(cons (cons 1 2) (cons (cons 3 4) 5))
((1 . 2) (3 . 4) . 5)
```

wiedergegeben werden. Zugriff auf den linken (bzw. rechten) Teilbaum erfolgt durch `first` (bzw. `rest`)

Bemerkung: Andere Baumdarstellungen [?] siehe später.

Assoziationslisten

Eine Assoziationsliste ist eine Liste von Dotted Pairs, wobei jedes Paar aus einem Schlüssel und einem Wert besteht. Mit `assoc` kann man nach bestimmten Schlüsseln suchen:

```
(setf *state-table*
  '((BW . Baden-Wuerttemberg)
    (BY . Bayern)
    (SL . Saarland)))

(assoc 'SL *state-table*)          ==> (SL . SAARLAND)
(cdr (assoc 'SL *state-table*)) ==> SAARLAND
(assoc 'NRW *state-table*)         ==> NIL
```

Assoziationslisten

Aufbau von Assoziationslisten mit `acons`:

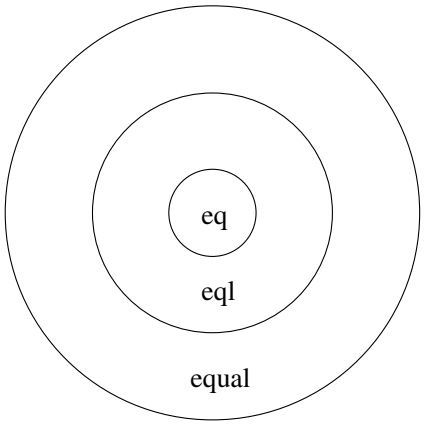
```
(acons 'BW 'Baden-Wuerttemberg
(acons 'BY 'Bayern
(acons 'SL 'Saarland nil)))
```

Bemerkung: Assoziationslisten benötigen weniger Speicher als Listen.

Identität und Gleichheit

1. `eq` testet auf Identität, d.h. auf Gleichheit der Adresse (Pointervergleich)
2. `eql` testet auf Gleichheit der Adresse (Pointervergleich) und Gleichheit von Zahlen.
3. `equal` testet auf Strukturgleichheit (Durchlaufen der Strukturen)

Bemerkung: Umgangssprachlich meint man mit `eql` “dasselbe” während man mit `equal` “das gleiche” meint.



Identität und Gleichheit

Beispiele:

```
(eq 'x 'x)                    ==> T
(eq "foo" "foo")              ==> NIL
(eq 3.5 3.5)                  ==> NIL
(eql 3.5 3.5)                 ==> T
(eql '(1 2 3) '(1 2 3))       ==> NIL
(eql "foo" "foo")             ==> NIL
(equal '(1 2 3) '(1 2 3))     ==> T
(equal "foo" "foo")           ==> T
```

Beachte: Lisp-Atome werden in einer Symboltabelle gehalten und haben daher immer die gleiche “Adresse”.

Zusammenfassung

Evaluierung

- In Lisp werden Eingaben in einer `read-eval-print`-Schleife interpretiert
- Zahlen, Zeichen, Zeichenreihen evaluieren zu sich selbst
- Symbole evaluieren zu dem an sie gebundenen Wert
- Die Evaluierung eines Ausdrucks kann durch QUOTE verhindert werden
- Komplexe Ausdrücke werden evaluiert, indem zu erst der Funktionsname und danach sämtliche Argumente evaluiert werden und dann die Funktion auf die Argumente angewendet wird.

Zusammenfassung

Listen

- Listen werden durch die Konstruktoren `nil` und `cons` gebildet.
- Die zugehörigen Testfunktionen sind `null` und `consp`
- Zugriffsfunktionen sind `first` und `rest`.

Darstellung von Listen

- Listen werden intern durch „Cons-Zellen“ dargestellt
- Gleichheitsfunktionen unterschiedlicher Stärke sind `eq`, `eql` und `equal`