Parsing-Algorithmen

Es gibt verschiedene Parsing-Algorithmen, die sich bezüglich ihrer Verarbeitungsstrategie unterscheiden:

Top-Down (oder zielgesteuert): Beginne mit dem Startsymbol und ersetze so lange Nichtterminale durch rechte Regelseiten, bis die Elemente des Strings erreicht sind (spielt die Ableitung nach)

Bottom-Up (oder datengesteuert): Beginne mit dem String aus Terminalsymbolen und ersetze sie solange mittels der rechten Regelseiten, bis das Startsymbol erreicht ist.

Depth- vs. Breadth-First versuche entweder, so schnell wie möglich einen hohen Baum aufzubauen oder baue zuerst alle kleineren Bäume auf

Uni- oder Bidirektional Suche in eine Richtung (links nach rechts oder umgekehrt) oder unterschiedlich (z.B. je nachdem, wo in einer Regel der linguistische Kopf zu finden ist).

Top-Down Parser

Ein Top-Down Parser geht folgendermaßen vor:

- Starte den Parse mit dem kompletten String als input und dem Startsymbol als Ziel.
- Das derzeitige Ziel ist ein Terminalsymbol:
  - es entspricht dem ersten Zeichen in der Eingabe, dann haben wir dieses Ziel erfolgreich abgeleitet und wir melden den abzuarbeitenden Teil der Eingabe zurück
  - dieses Ziel schlägt fehl
- das derzeitige Ziel ist ein Nichtterminal: versuche nacheinander für alle Regeln, die das Ziel als linke Seite haben, alle Teilziele abzuarbeiten. Zunächst mit der gegebenen Eingabe das erste Teilziel, mit der übriggebliebenen Eingabe das zweite Teilziel und so weiter bis wir alle Teilziele abgeleitet haben. Der Rest der Eingabe, die für die Ableitung der Teilziele nicht gebraucht wurde, wird zurückgegeben.

Ein Top-Down Parser versucht, die Linksableitung einer Kette zu finden bzw. den Parsebaum von der Wurzel aus in Präordnung zu berechnen.

Top-Down Parser in LISP

Datenstrukturen (I)

```
(defparameter *grammar*   '((S NP VP)
                            (NP Det N)
                            (VP V) (VP V NP)
                            (Det the) (Det a)
                            (N woman) (N man) (N ball) (N scope)
                            (V snored) (V saw)))
(defun rule-lhs (rule)  (first rule))
(defun rule-rhs (rule)  (rest rule))

(defun rules-for-category (category)
  (loop for rule in *grammar*
      when (eq (rule-lhs rule) category)
      collect rule))
```


Top-Down Parser in LISP

Datenstrukturen (II)

```
(defstruct parse  analyzed
                  unanalyzed)

(defun complete-parses (parses)
  (remove-if-not #'null parses :key #'parse-unanalyzed))

(defun make-tree (root daughters)  (cons root daughters))

(defun tree-root (tree)  (first tree))

(defun tree-daughters (tree)  (rest tree))
```

Top-Down Parser in LISP

Parse-Funktion

```
(defun parse (input goal)
  ;; versuche, goal mit einem Praefix von input abzuleiten
  (if (eq (first input) goal)
      ;; goal laesst sich mit dem ersten Zeichen ableiten: erzeuge ein
      ;; (Teil-)Resultat, das den Parsebaum in :analyzed und den Rest der
      ;; noch zu verarbeitenden Eingabe in :unanalyzed enthaelt
      (list (make-parse :analyzed (first input) :unanalyzed (rest input)))
    ;; expandiere goal durch alle Regeln, die goal auf der linken Seite haben
    ;; und versuche, die einzelnen Teilziele auf den entsprechenden rechten
    ;; Seiten nacheinander abzuleiten: das tut extend-parse
    (loop for rule in (rules-for-category goal)
        nconc (extend-parse (rule-lhs rule) nil (rule-rhs rule) input))))
```


Hilfs-Funktion extend-parse

```
(defun extend-parse (lhs analyzed unanalyzed input)
  (if (null unanalyzed)
      ;; Alle Teilziele wurden abgeleitet: Erzeuge ein (Teil-)Resultat,
      ;; das den lokalen Parsebaum in :analyzed enthaelt und in :unanalyzed
      ;; den noch zu verarbeitenden input
     (list (make-parse :analyzed (make-tree lhs analyzed) :unanalyzed input))
    ;; versuche, das erste Teilziel abzuleiten. Fuer alle Teilresultate, die
    ;; erzeugt werden, versuche, die weiteren Teilziele abzuleiten
    (loop for parse in (parse input (first unanalyzed))
        nconc (extend-parse
               lhs
               (append analyzed (list (parse-analyzed parse)))
               (rest unanalyzed)
               (parse-unanalyzed parse)))))
```

Bottom-Up Parser

Ein Bottom-Up Parser geht folgendermassen vor:

- Suche die Regeln, deren rechte Seite mit einem Teilstring der Eingabe matcht und ersetze den Teilstring.
- Iteriere für die so entstandenen neuen Ketten bis eine Kette gefunden ist, die nur aus dem Startsymbol besteht, oder keine Ersetzungen mehr möglich sind.

Bottom-Up Parser in LISP / Datenstrukturen (I)

```
(defparameter *grammar*  '((S NP VP)
                           (NP Det N) (NP NP PP)
                           (VP V) (VP V NP) (VP VP PP)
                           (PP P NP)
                           (Det the) (Det a)
                           (P with)
                           (Adj fat) (Adj cute)
                           (N Adj N) (N woman) (N man) (N ball) (N scope) (N hat)
                           (V snored) (V saw)))
(defun rule-lhs (rule)   (first rule))
(defun rule-rhs (rule)   (rest rule))
(defun rules-starting-in (category)  (loop for rule in *grammar*
                                         when (eq (first (rule-rhs rule)) category)
                                         collect rule))
```

Bottom-Up Parser in LISP / Datenstrukturen (II)

```
(defstruct parse   tree
                   remaining-input
                   start
                   span)

(defun complete-parses (parses)
  (remove-if-not #'null parses :key #'parse-remaining-input))

(defun make-tree (root daughters)  (cons root daughters))
(defun tree-root (tree)  (first tree))
(defun tree-daughters (tree)  (rest tree))
```


Bottom-Up Parser in LISP / Parse-Funktion

```
(defun parse (input &key (position 0))
  ;; while there is (remaining) input, find all rules that have the next input
  ;; word as their first (i.e. leftmost) right-hand side symbol and then try to
  ;; instantiate the right-hand side in extend-parse() calls; here, the initial
  ;; dot position (the borderline between the .right-of-dot. and .left-of-dot.
  ;; parts of the rule right-hand side) is _behind_ the first symbol which at
  ;; the same time consumes the current input token.  extend-parse() results
  ;; are accumulated in a single list by means of 'nconc'
  (when input  (loop for rule in (rules-starting-in (first input))
                 nconc (extend-parse (rule-lhs rule)
                                     (list (first (rule-rhs rule)))
                                     (rest (rule-rhs rule))
                                     (rest input)
                                     :start position :span 1))))
```

Bottom-Up Parser in LISP / Hilfs-Funktionen (I)

```
(defun snoc (item list)
  ;;
  ;; inverse cons(): add .item. as new last element to .list.
  ;;
  (append list (list item)))
```


Bottom-Up Parser in LISP / Hilfs-Funktionen (II)

```
(defun extend-parse (lhs left-of-dot right-of-dot input &key start span)
  (if (null right-of-dot)
      (let ((parse (make-parse :tree (make-tree lhs left-of-dot)
                               :remaining-input input
                               :start start :span span)))
        (cons parse
              (loop
                  for rule in (rules-starting-in lhs)
                  nconc (extend-parse (rule-lhs rule)
                                      (list (parse-tree parse))
                                      (rest (rule-rhs rule))
                                      input
                                      :start start :span span))))
    ;;else
```

Bottom-Up Parser in LISP / Hilfs-Funktionen (III)

```
(defun extend-parse (lhs left-of-dot right-of-dot input &key start span)
  (if (null right-of-dot)
    ......  ;; then (siehe letzte Folie)
    ;; else:
    (loop for parse in (parse input :position (+ start span))
        nconc
          (when (eq (tree-root (parse-tree parse))
                    (first right-of-dot))
            (extend-parse lhs
                          (snoc (parse-tree parse) left-of-dot)
                          (rest right-of-dot)
                          (parse-remaining-input parse)
                          :start start
                          :span (+ span (parse-span parse))))))
```