Shift-Reduce-Parsing

Übersicht

- Parsing-Richtungen
  - Top-Down: hypothesengesteuert
  - Bottom-Up: datengesteuert
- Shift-Reduce-Parsing als Bottom-Up-Verfahren
  - Stapel als Datenstruktur
  - Der Algorithmus: Shift- und Reduce-Schritte
- Implementation in Prolog
  - Eingabekette konsumieren und Stapel aufschichten: shift/4
  - Grammatikregeln und Lexikon: brule/2 und word/2
  - Reduktion mittels Grammatikregeln: reduce/2
- Parsing-Algorithmus: shift_reduce/3
  - Terminierungsprobleme
  - Tilgungsregeln und zyklische Regeln

Top-Down

Ein Top-Down-Parser für eine kontextfreie Grammatik

- fängt mit dem Startsymbol an.
- führt wiederholt Ableitungsschritte durch.
  - expandiert jeweils LHS durch RHS.
- Ziel: Ableiten der zu analysierenden Kette

S → NP VP	*Det → a*
NP → Det N	*N → man*
VP → V	*V → sleeps*

S ⇒ ***NP*** *VP*
⇒ ***Det*** *N* VP
⇒ a **N** VP
⇒ a *man* **VP**
⇒ a man ***V***
⇒ a man *sleeps*

Legende

fett: Nicht-Terminal-Symbol, das im nächsten Schritt expaniert wird.

kursiv: Symbol, das durch letzten Schritt expandiert wurde

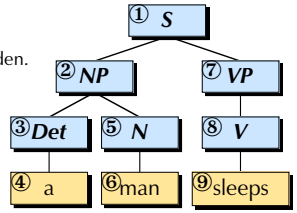
Top-Down

Vorgehen beim Top-Down-Parsing

- Ich suche ein S.
- Um S *(1)* zu erhalten, brauche ich eine NP und eine VP.
- Um NP *(2)* zu erhalten, brauche ich ein Det und ein N.
- Um Det *(3)* zu erhalten, kann ich das Wort *»a« (4)* verwenden.
- Um N *(5)* zu erhalten, kann ich das Wort *»man« (6)* verwenden.
- Damit ist die NP vollständig.
- Um VP *(7)* zu erhalten, brauche ich ein V.
- Um V *(8)* zu erhalten, kann ich das Wort *»sleeps« (9)* verwenden.
- Damit ist die VP vollständig.
- Damit ist das S vollständig.

S → NP VP	*Det → a*
NP → Det N	*N → man*
VP → V	*V → sleeps*

➥ *hypothesengesteuert*



Bottom-Up

Ein Bottom-Up-Parser für eine kontextfreie Grammatik

- fängt mit der zu analysierenden Kette an.
- führt wiederholt Ableitungsschritte *»rückwärts«* durch.
  - reduziert jeweils RHS auf LHS.
- Ziel: Erreichen des Startsymbols

S → NP VP	*Det → a*
NP → Det N	*N → man*
VP → V	*V → sleeps*

S ⇐ **NP** ***VP***
⇐ NP ***V***
⇐ *NP* **sleeps**
⇐ **Det** ***N*** sleeps
⇐ *Det* **man** sleeps
⇐ **a** man sleeps

Legende

fett: Nicht-Terminal-Symbol, das im nächsten Schritt reduziert wird

kursiv: Symbol, das durch letzten Schritt reduziert wurde

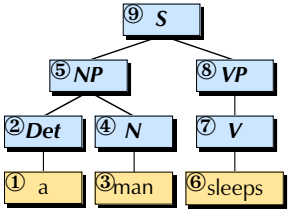
Bottom-Up

Vorgehen beim Bottom-Up-Parsing

- Nimm ein Wort — es ist *»a« (1).*
- *»a«* ist ein Det *(2).*
- Nimm ein weiteres Wort — es ist *»man« (3).*
- *»man«* ist ein N *(4).*
- Det und N bilden zusammen eine NP *(5).*
- Nimm ein weiteres Wort — es ist *»sleeps« (6).*
- *»sleeps«* ist ein V *(7).*
- V bildet (für sich alleine) eine VP *(8).*
- NP und VP bilden zusammen ein S *(9).*

➥ *datengesteuert*

$S \rightarrow NP\ VP$	$Det \rightarrow a$
$NP \rightarrow Det\ N$	$N \rightarrow man$
$VP \rightarrow V$	$V \rightarrow sleeps$



Shift-Reduce-Parsing

Das Shift-Reduce-Parsing ist ein einfaches Bottom-Up-Verfahren

- **Daten**
  - Eingabekette: Was muss noch verarbeitet werden? – **Liste**
  - Abarbeitungs-Stapel: Was haben wir alles schon erkannt? – **Stapel**
- **Aktionen**
  - Eingabekette konsumieren – **shift**
    - *Nimm ein Wort*
  - Grammatische Regeln anwenden – **reduce**
    - Lexikalische Regeln (*»a« ist ein Det*)
    - Syntaktische Regeln (*Det und N bilden zusammen eine NP*)

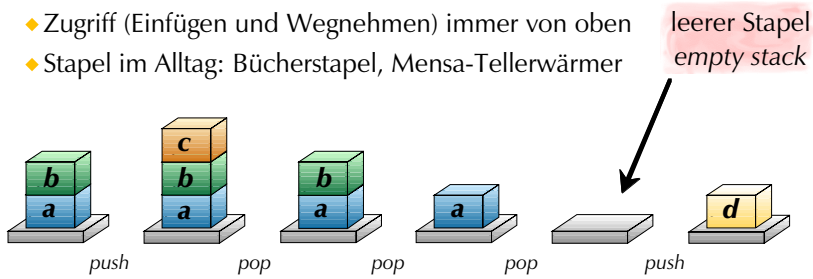
Shift-Reduce-Parsing

Schritt	Aktion	Stapel	Eingabekette
0	—	*ε*	*the man sleeps*
1	shift	*the*	*man sleeps*
2	reduce	*Det*	*man sleeps*
3	shift	*Det man*	*sleeps*
4	reduce	*Det N*	*sleeps*
5	reduce	*NP*	*sleeps*
6	shift	*NP sleeps*	*ε*
7	reduce	*NP V*	*ε*
8	reduce	*NP VP*	*ε*
9	reduce	*S*	*ε*

Die Daten: Stapel

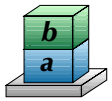
Die Datenstruktur "Stapel" (auch: Keller), engl. *Stack*

- zwei Operationen
  - i. ***push*** — ein Element auf den Stapel darauflegen
  - ii. ***pop*** — oberstes Element vom Stapel wegnehmen
- Zugriff (Einfügen und Wegnehmen) immer von oben
- Stapel im Alltag: Bücherstapel, Mensa-Tellerwärmer



Stapel als Listen

Stapel können als Listen betrachtet werden.



= [b , a]

oberstes Element auf Stapel
= erstes Element in Liste

= [ ]

leerer Stapel
= leere Liste

push und pop als Prädikate über Listen

```
?- push(c, [b,a], Stack)).
Stack = [c,b,a]
```

```
?- pop(E, [c,b,a], New)).
E = c
New = [b,a]
```


Die Aktionen

In jedem Schritt führt ein Shift-Reduce-Parser eine von zwei möglichen Aktionen durch

- **Shift**
  - »Nimm ein Wort«
  - verschiebe ein Wort auf den Stapel (*schiebe*)
- **Reduce**
  - »X und Y bilden zusammen ein Z«
  - »X bildet (für sich alleine) ein Z«
  - »X ist ein Z«
  - wenn die obersten Stapelelemente gleich der rechten Seite einer Regel sind, ersetze sie durch die linke Seite der Regel (*reduziere*)

Shift-Reduce-Algorithmus

Ablauf beim Shift-Reduce-Parsing

I. **Shift**: Verschiebe ein Wort von der Eingabekette auf den Stapel

II. **Reduce**: Reduziere den Stapel so lange mit Hilfe der lexikalischen und syntaktischen Regeln, bis keine weiteren Reduktionen mehr möglich sind.

III. Sind noch mehr Wörter in der Eingabekette?
  - ja: Gehe zum Schritt I.
  - nein: Stop.

Das Resultat der syntaktischen Analyse befindet sich auf dem Stapel.

Implementierungstechniken I

Implementation des Stapels und der Eingabekette

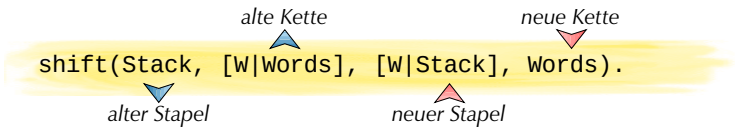
- Stapel als Liste darstellen
  - Notationswechsel: Als Liste wächst Stapel nach links: *Det man* ≡ `[man,det]`
- Eingabekette ebenfalls als Liste (Stapel) darstellen

Schritt	Aktion	Stapel	Eingabekette
0	—	`[]`	`[the,man,sleeps]`
1	shift	`[the]`	`[man,sleeps]`
2	reduce	`[det]`	`[man, sleeps]`
3	shift	`[man,det]`	`[sleeps]`
...	...	...	...
9	reduce	`[s]`	`[]`

shift/2

Ein einzelner Shift-Schritt

- nimmt das erste Wort von der Eingabekette weg (*pop*)
- setzt es zuoberst auf den Stapel (*push*)




```
?- shift([det], [man,sleeps], NewStack, NewString).
NewStack = [man,det],
NewString = [sleeps]
```

Schritt	Aktion	Stapel	Eingabekette
2		[det]	[man,sleeps]
3	shift	[man,det]	[seeps]

Implementierungstechniken II

Effiziente Implementierung der Grammatikregeln

- RHS wird "rückwärts" als offene Liste notiert, damit sie mit Stapel unifiziert! (***b**ackward **rule***)

NP → Det N ←

```
brule([n,det|X], [np|X]).
```

Aktion	Stapel
reduce	*Det N*
reduce	*NP*

Aktion	Stapel
reduce	[n,det]
reduce	[np]

- Aufruf der Regeln reduziert Stapel!

```
?- brule([n,det|Rest], NewStack).
NewStack = [np|Rest]
```


Syntax und Lexikon

Syntaktische Regeln

```
brule([vp,np|X], [s|X]).
brule([n,det|X], [np|X]).
brule([v|X], [vp|X]).
```

S → NP VP
NP → Det N
VP → V

- Lexikonregel

```
brule([Word|X], [Cat|X]) :-
  word(Word, Cat).
```

Lexikon

```
word(a,det).
word(man, n).
word(sleeps, v).
```

Det → a
N → man
V → sleeps

Reduktion: reduce/2

Rekursionsschritt

- Reduziere den Stapel so oft mit einer passenden Grammatikregel, wie es geht.

```
reduce(Stack, ReducedStack) :-
  brule(Stack, Stack2),
  reduce(Stack2, ReducedStack).
```

Abbruchbedingung

- Wenn keine Regel passt, lass den Stapel unverändert.

```
reduce(Stack, Stack).
```

"Catch-All"-Klausel

reduce/2 berechnet die transitive Hülle der brule-Relation!

Parsen mit shift_reduce/3

Abbruchbedingung

- Eingabekette ist leer

```
shift_reduce([], Stack, Stack).
```

Rekursionsschritt

- führt einen einzelnen Shift-Schritt mit shift/2 durch
- benutzt reduce/2, um den Stapel so weit wie möglich zu reduzieren

```
shift_reduce(String, Stack, Result) :-
   shift(Stack, String, NewStack, NewString),
   reduce(NewStack, ReducedStack),
   shift_reduce(NewString, ReducedStack, Result).
```


Problematische Regeln

Terminierungsprobleme von Bottom-Up-Verfahren wie dem Shift-Reduce-Parsing

- bei **Tilgungsregeln** $X \to \varepsilon$

```
brule(Rest, [x|Rest]).
```

  - Regel kann immer angewendet werden
  - Keller wächst immer weiter

- bei **zyklischen Regeln** $A \to B$, $B \to A$

```
brule([b|Rest], [a|Rest]).
brule([a|Rest], [b|Rest]).
```

  - Regeln können zyklisch folgend immer wieder angewendet werden
  - der Keller wird nie kleiner (allenfalls bleibt er immer gleich gross)