Shift-Reduce-Parsing

Übersicht

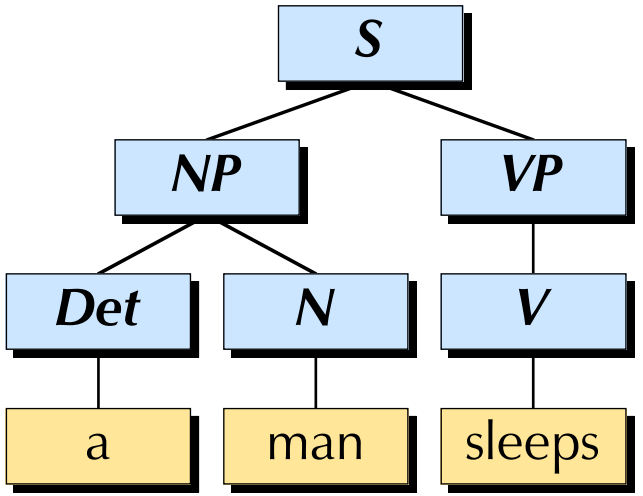
- Shift-Reduce-Parsing: Ein einfaches Bottom-Up-Verfahren
- Keller
- Shift- und Reduce-Schritte
- Vorgehen des Parsers
- Implementation in Prolog
- Probleme
  - Tilgungsregeln
  - Zyklische Regeln

Shift-Reduce-Parsing

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

- Nimm ein Wort — es ist *»a«*
- *»a«* ist ein Det
- Nimm ein weiteres Wort — es ist *»man«*
- *»man«* ist ein N
- Det und N bilden zusammen eine NP
- Nimm ein weiteres Wort — es ist *»sleeps«*
- *»sleeps«* ist ein V
- V bildet (für sich alleine) eine VP
- NP und VP bilden zusammen ein S

$S \rightarrow NP\ VP$	$Det \rightarrow the$	$V \rightarrow loves$
$NP \rightarrow Det\ N$	$Det \rightarrow a$	$V \rightarrow sleeps$
$VP \rightarrow V\ NP$	$N \rightarrow man$	$V \rightarrow sees$
$VP \rightarrow V$	$N \rightarrow woman$	$V \rightarrow thinks$



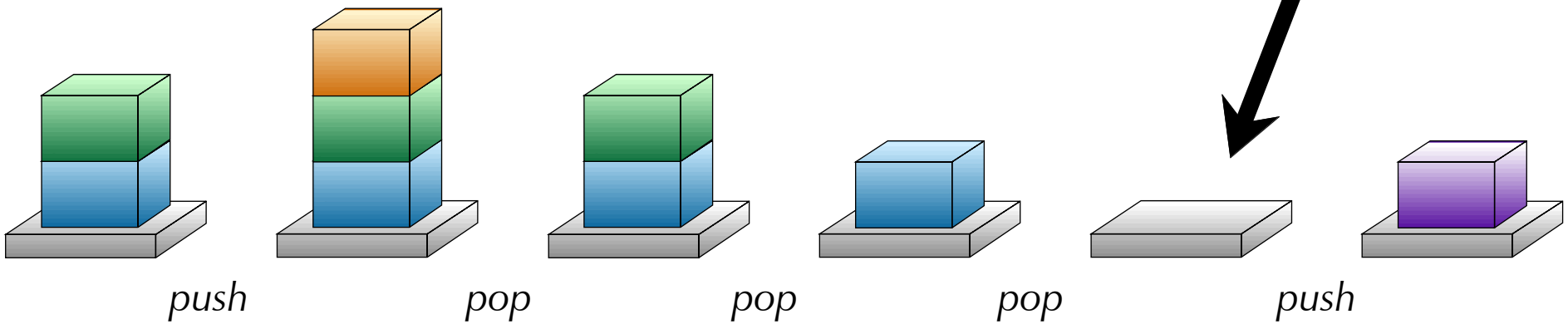
$S \Rightarrow NP\ VP \Rightarrow NP\ V \Rightarrow NP\ sleeps \Rightarrow Det\ N\ sleeps \Rightarrow Det\ man\ sleeps \Rightarrow a\ man\ sleeps$

Keller

Keller (auch: Stapel), engl. *Stack*

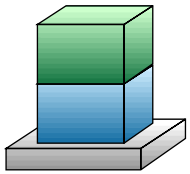
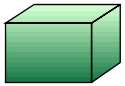
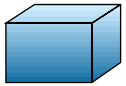
- wichtige Datenstruktur
- zwei Operationen:
  - *push* — etwas oben auf den Stapel darauflegen
  - *pop* — oberste Schicht vom Stapel wegnehmen
- Zugang immer von oben
- Keller im Alltag: Bücherstapel; Mensatellerwärmer

leerer Keller
empty stack



Keller in Prolog

Stacks können in Prolog einfach als Listen programmiert werden:

-  = [ , ]
-  = [ ]
- oberstes Element auf Stapel = erstes Element in Liste
- push und pop als Listenoperationen

Shift-Reduce-Parsing

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

- Shift
  - »nimm ein Wort«
  - verschiebe ein Wort auf den Keller
- Reduce
  - »X und Y bilden zusammen ein Z«, »X bildet (für sich alleine) ein Z«, »X ist ein Z«
  - wenn die obersten Kellerelemente gleich der rechten Seite einer Regel sind, ersetze sie durch die linke Seite der Regel (Reduktion)

Shift-Reduce-Parsing

Vorgehen beim Shift-Reduce-Parsing:

- Verschiebe ein Wort von der Eingabekette auf den Keller (»Shift«)
- Reduziere die obersten Keller-Elemente (»Reduce«)
  - so lange, bis die obersten Kellerelemente nicht mehr der rechten Seite einer Grammatikregel entsprechen
- Sind noch mehr Wörter in der Eingabekette?
  - ja: nochmals von vorne
  - nein: Stop

Shift-Reduce-Parsing

Schritt	Aktion	Keller (Stack)	Eingabe-Kette
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

Shift-Reduce-Parser in Prolog

Um einen Shift-Reduce-Parser in Prolog zu implementieren, empfiehlt sich:

- Keller »rückwärts« als Liste darstellen
  - oberstes Keller-Element = erstes Element in der Liste
  - Det N ⇒ `[n, det]`
- Regeln so darstellen, dass sie mit Keller unifizieren
  - auch rückwärts
  - *NP* → *Det N*
    ⇒ `brule([n, det | RestKeller], [np | RestKeller]).`

Grammatikregeln, Lexikon

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

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

word(det, the).
word(n, dog).
word(n, cat).
word(v, sees).
```

Ein einzelner Reduce-Schritt

Durch diese Darstellung von Regeln und Keller ist ein einzelner Reduce-Schritt sehr effizient:

$NP \rightarrow Det\ N$

Aktion	Keller (Stack)
	NP V Det N
reduce	*NP V NP*

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

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

Mehrere Reduce-Schritte: reduce / 2

Der Keller wird so lange reduziert, bis keine Grammatik-regel mehr anwendbar ist.

Rekursiver Fall

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

Abbruch

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

shift / 2

Ein einzelner Shift-Schritt

- nimmt erste Wort von der Eingabekette weg
- »pusht« es zuoberst auf den Keller

```
shift(Stack, [Word|S], [Word|Stack], S).
```

alter Keller *alte Kette* *neuer Keller* *neue Kette*

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

Schritt	Aktion	Keller	Eingabe-Kette
2		*Det*	*man sleeps*
3	shift	*Det man*	*sleeps*

shift_reduce / 3

Das Prädikat shift_reduce

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

Abbruch

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

Rekursiver Fall

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

Tilgungsregeln

Reine Bottom-Up-Verfahren (z.B. Shift-Reduce-Parsing) kommen nicht mit Tilgungsregeln zurecht.

- wenn die Grammatik eine Regel der Form $X \rightarrow \varepsilon$ enthält
  - kann der Reduce-Schritt beliebig oft angewendet werden
  - dabei wächst der Keller immer weiter
  - ohne dass der Parser näher zur Lösung kommen würde

Schritt	Aktion	Keller (Stack)	Eingabe-Kette
0	—	ε	*the man sleeps*
1	shift	*the*	*man sleeps*
2	reduce	*X the*	*man sleeps*
3	reduce	*X X the*	*man sleeps*
4	reduce	*X X X the*	*man sleeps*
5	reduce	*X X X X the*	*man sleeps*
6	reduce	*X X X X X the*	*man sleeps*

Zyklische Regeln

Reine Bottom-Up-Verfahren (z.B. Shift-Reduce-Parsing) kommen nicht mit Regeln zurecht, die zu Zyklen führen.

- Beispiel: $A \to B \quad \cdot \quad B \to C \quad \cdot \quad C \to A$
- wie bei Tilgungsregeln kann der Reduce-Schritt beliebig oft durchgeführt werden
- somit ist wiederum nicht garantiert, dass der Parser terminiert

Aktion	Keller (Stack)
reduce	B …
reduce	A …
reduce	C …
reduce	B …
reduce	A …
reduce	C …
reduce	B …
reduce	A …
reduce	C …
reduce	B …
reduce	A …

Problematische Grammatiken

Welche Komplikationen tauchten bisher auf?

- Links-Rekursion
  - Beispiel: $NP \rightarrow Det\ N \cdot Det \rightarrow NP\ Poss$ (für »the kings's crown«)
  - Beispiel: $S \rightarrow S\ Conj\ S$ (für »Eine Katze singt und ein Hund bellt«)
  - problematisch für Top-Down-Parser, die von links nach rechts parsen
- Rechts-Rekursion
  - Beispiel: $S \rightarrow S\ Conj\ S$ (für »Eine Katze singt und ein Hund bellt«)
  - problematisch für Top-Down-Parser, die von rechts nach links parsen
- Tilgungsregeln: $X \rightarrow \varepsilon$
  - problematisch für Bottom-Up-Parser
- Zyklische Kettenregeln: $A \rightarrow B \cdot B \rightarrow C \cdot C \rightarrow D \cdot D \rightarrow A$
  - problematisch für Bottom-Up-Parser; für Top-Down wg. Links-/Rechtsrekursion

Aufgaben: Shift-Reduce-Parsing

Programmiertechniken der Computerlinguistik 1 · Wintersemester 1998/99

1. Keller-Operationen

Schreibe Prolog-Prädikate für die Operationen auf einem Keller. Es sind jeweils Beispiele für die Benutzung dieser Prädikate angegeben.

a) push/3 — Nimmt ein Element und einen Keller; liefert einen neuen Keller zurück, bei dem das Element zuoberst auf dem Eingabekeller liegt.

```
?- push(marmelade, [maus, kartoffel, assel], NeuerKeller).
NeuerKeller = [marmelade, maus, kartoffel, assel]
```

b) pop/3 — Nimmt einen Keller; liefert das oberste Element des Kellers und den neuen Keller zurück.

```
?- pop([maus, kartoffel, assel], Gepoppt, NeuerKeller).
Gepoppt = maus,
NeuerKeller = [kartoffel, assel]
```

Was passiert, wenn Du versuchst, das oberste Element von einem leeren Keller wegzunehmen?

2. Shift-Reduce-Parsing

Führe auf dem Papier ein Shift-Reduce-Parsing der folgenden Eingabeketten durch. Gib für jeden Shift- oder Reduce-Schritt an, wie Eingabekette und Keller aussehen.

a) the cat sees a dog

b) the dog the dog

3. Strukturbäume

Der bisherige Shift-Reduce-Parser ist eigentlich gar kein Parser, sondern lediglich ein Akzeptor: Er gibt nur aus, ob ein Satz der Grammatik entspricht oder nicht. Für die meisten computerlinguistischen Anwendungen wird jedoch die *Struktur* der erkannten Sätze gebraucht.

Welche Änderungen sind nötig, damit die Struktur in der selben Form wie früher bei den DCGs ausgegeben wird?

Beispiel (die Einrückungen sind nur zur Verdeutlichung; Prolog wird dagegen alles in einer einzigen Zeile schreiben):

```
?- parse([the,dog,sees,the,cat], [s(Struktur)]).
Struktur = s(np(det(the),
                n(dog)),
             vp(v(sees),
                np(det(the),
                   n(cat))))
```