informatiekunde

Natuurlijke Taalverwerking
Natural Language Processing

Gosse Bouma and Begoña Villada

3e trimester 2002/2003

Overview

1. Parsing
   - Left-recursion,
   - Top-down vs. Bottom-up strategies,
2. Shift-reduce parsing
   - Implementation in Prolog,
   - Ambiguity,
   - Epsilon-rules.

Parsing

- Prolog provides top-down, depth-first, parsing strategy as default,
- Many alternative strategies exist,
- Often more robust and efficient.

Top-down Parsing

- DCG uses Prolog top-down search strategy,
- Therefore, left-recursion leads to problems,

```
ancestor(X,Y) :-
    parent(X,Y).
ancestor(X,Y) :-
    ancestor(X,Z), parent(Z,Y).
```

Left-recursion in Grammar

- een kind, een kind in het park
- `n --> n, pp.`
- Kim slaapt, Kim slaapt tot 10 uur
- `vp --> vp, pp`
- Kim slaapt en Sandy werkt
- `s --> s, [en], s.`

Left-recursion in Grammar

- Peter's (broer's) huis
- `np --> det, n`
- `det --> np, [s].`
- een (erg) aardig kind
- `a --> int, a.`
- `int --> [erg];[heel];[vet]; [].`

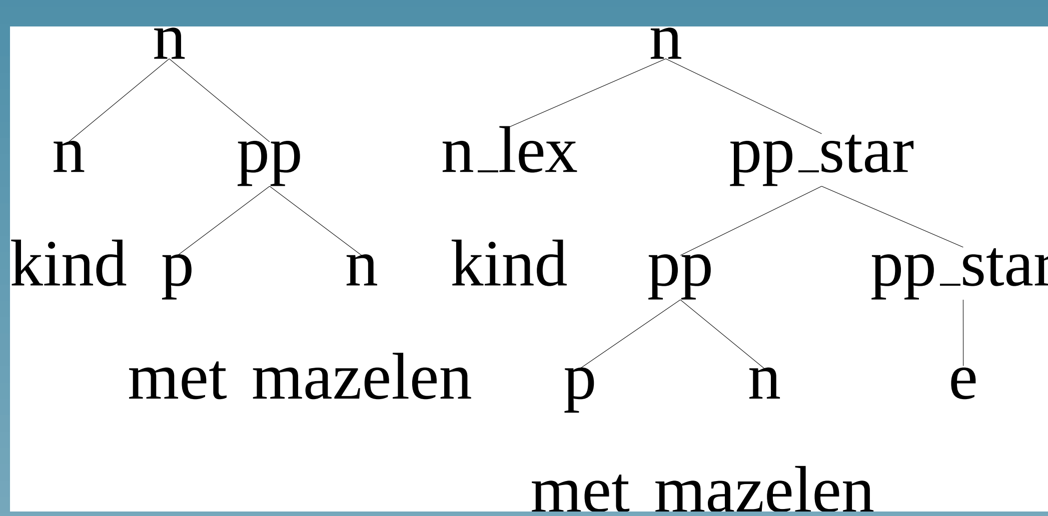
Removing Left-recursion

```
n --> n, pp
pp --> p, n

n --> n_lex, pp_star
n_lex --> [kind].
pp_star --> pp, pp_star.
pp_star --> [].
pp --> p, np.
```

Removing Left-recursion

- Changing the grammar also changes the structure assigned to input-strings,
- Although this can be fixed as well.



Removing Left-recursion

- Removing left-recursion can lead to explosion of the number of rules in the grammar. (Moore, Proceedings NAACL, 2000)

Grammar	Number of Rules
Toy	88
Paull "Best"	156
Paull "Lexicographic"	970
Paull "Worst"	5696

Alternative Parsing Methods

- Prolog searches top-down, depth-first,
- Alternatives:
  - ⋆ Bottom-up Parsing: work from the input towards the goal (s).
  - ⋆ Breadth-first (parallel): Explore all ways to expand a rule in parallel.

Alternative Parse Strategy

- Requires separation of rules (data) and parser (algorithm)

- Grammar rules as data:

```
% rule(Mother_Category,List_Daughters)
rule(s,[np,vp]).

% lex(Category,Word).
lex(np,kim).
```

Top-down Parsing in Prolog

```
top_down(Cat,P0,P1) :-
    rule(Cat,Daughters),
    find_ds(Daughters,P0,P1).
top_down(Cat,[Word|Ws],Ws) :-
    lex(Cat,Word).

find_ds([D1|Ds],P0,P2) :-
     top_down(D1,P0,P1),
     find_ds(Ds,P1,P2).
find_ds([],P0,P0).
```

Top-down vs Bottom-up

Top-down Parsing	Bottom-up Parsing
S	the dog barks
NP VP	DET dog barks
DET N VP	DET N barks
the N VP	NP barks
the dog VP	NP V
the dog V	NP VP
the dog barks	S

Shift-reduce parsing

- Bottom-up parsing!
- Start with the input, and search for lexical categories,
- Try to combine categories into phrases
- Try to combine phrases into larger phrases or a sentence.
- Bottom-up parsers do not loop on left-recursive rules.

Shift-reduce Algorithm

- Stack: for storing intermediate result,
- Shift: Remove the leftmost element of the input and add its category to the Stack,
- Reduce: Replace $C_1..C_n$ on the Stack by C_0 given a rule $C_0 \longrightarrow C_1..C_n$.

Shift-reduce Algorithm

	String	Stack	Action	Rule
1	the dog barks	[]	sh	lex(det,the)
2	dog barks	[det]	sh	lex(n,dog)
3	barks	[det,n]	red	rule(np,[det,n])
4	barks	[np]	sh	lex(v,barks)
5		[np,v]	red	rule(vp,[v])
6		[np,vp]	red	rule(s,[np,vp])
7		[s]		

Shift-reduce in Prolog

```
sr(Input,Stack) :-
  reduce(Stack,NewStack),
  sr(Input,NewStack).
sr(Input,Stack) :-
  shift(Input,Rest,Cat),
  sr(Rest,[Cat|Stack]).

shift([Wrd|Input],Input,Cat) :-
  lex(Cat,Wrd).
```

Shift-reduce in Prolog

```
reduce(Stack,[M|NewStack]) :-
   reduce_rule(M,Ds),
   append(Ds,NewStack,Stack).

reduce_rule(s,[vp,np]).
```

- Note the order of Ds in reduce_rule is reversed!

Optimized Reduce

```
reduce([vp,np|Stack],[s|Stack]).
reduce([n,det|Stack],[np|Stack]).
```

- No need for append or search,
- Rules can be automatically converted in reduce actions.

Ambiguity

- Kim bought a house with a garage.
- Kim bought a house in January.

$$\begin{array}{l} vp \longrightarrow v\ np \\ vp \longrightarrow vp\ pp \\ np \longrightarrow np\ pp \end{array}$$

Shift-reduce conflict

	String	Stack	action	rule
1.	bought a house	[]	..	..
	with a garage		..	..
m.	with a garage	[v,np]	red	vp $\longrightarrow$ v np
n.	with a garage	[vp]	...	
o.		[vp,pp]	red	vp $\longrightarrow$ vp pp
m.	with a garage	[v,np]	shift	lex(with,p)
..				
p.		[v,np,pp]	red	np $\longrightarrow$ np pp
q.		[v,np]	..	

Main and Subordinate Clauses

- Piet slaapt
- Jan denkt dat Piet slaapt
- Piet leest een boek
- Jan denkt dat Piet een boek leest

s $\longrightarrow$ np vp	bijzin $\longrightarrow$ np vpb
vp $\longrightarrow$ v np	vpb $\longrightarrow$ np v
vp $\longrightarrow$ v	vpb $\longrightarrow$ v
vp $\longrightarrow$ v [dat] bijzin	

Reduce-reduce conflict

	String	Stack	action	rule
1.	Jan denkt dat Piet slaapt	[]		
m.		[..,dat,np,vp]	red	s ⟶ np vp
n.		[..dat,s]	..	
...				
m.		[..,dat,np,vp]	red	bz ⟶ np vpb
n		[..dat,bz]	..	
..				

Advantages of shift-reduce Parsing

- Left-recursion is no problem.
- Size of the stack bounded by the number of words in the input.
- Reduce-actions terminate as long as the grammar contains no cycles.

$$\begin{array}{l} \text{np} \longrightarrow \text{n} \\ \text{n} \longrightarrow \text{np} \end{array}$$

Disadvantage of Shift-reduce

$$\boxed{\text{det} \longrightarrow \epsilon}$$

- Epsilon-rules require that a category is added to the stack, which does not correspond to a word in the input
- Size of the stack no longer bounded by the input,
- Epsilon-rules cause non-termination.

SR and Epsilon's

	String	Stack	action	rule
1	dogs bark	[]	sh	
2	dogs bark	[det]	sh	
3	dogs bark	[det,det]	sh	
...				

Removing Epsilon's

- The effect of epsilon-rules can be achieved indirectly as well:

$$\frac{\text{np} \longrightarrow \text{det n} \qquad \text{det} \longrightarrow \epsilon}{\text{np} \longrightarrow \text{n}}$$

Removing Epsilon's

- For all rules C $\longrightarrow \epsilon$ and
- all rules M $\longrightarrow$ $C_1 \ldots C_i, C, C_j \ldots C_n$,
- Add M $\longrightarrow$ $C_1 \ldots C_i, C_j \ldots C_n$,

Efficiency

- DCG and shift-reduce parsers are normally depth-first parsers:
  - ⋆ Find different solutions by backtracking,
- Depth-first parsing can be very inefficient,
- Breadth-first parsing is usually much more efficient,
  - ⋆ Chart-parsers are Breadth-first parsers.