University of Malta
BSc(Hons.) IT Year III

CSA305: Natural Language Processing using Logic Programming

Lecture 10

Parsing Algorithms

- Top Down Left Right Parsing
- Shift/Reduce Parsing
- Left Corner Parsing

Department of Computer Science and AI
1999/2000

Grammar Interpreters

The DCG system is a **compiler** which translates rules such as

```
s --> np, vp.
```

directly into clauses (i.e. "Prolog machine code")

```
s(P0,P) :- np(P0,P1), vp(P1,P).
```

We can then recognise a sentence by issuing a query

```
?- s(['John',ate,the,cat],[ ]).
```

which is executed by Prolog in the normal way. The recognition algorithm is *implicit* in Prolog's control strategy.

How can we make the algorithm *explicit*?

- Storing grammar rules in a neutral way as **facts** having the form `rule(lhs,rhs)`, e.g.

  rule(s,[np,vp]).
  rule(np,[n]).
  rule(np,[d,n]).
  etc.

- Explicitly defining a recognition predicate of the form `parse(C,P1,P)` where C is the category we want to recognise, and P1 and P are text indices.

A Top-Down Left-Right Recogniser in Prolog

```
% parse(C,P1,P)
%
% Recognise a consituent of category C over
% text segment defined by indices P1 and P.

parse(C,[Word|X],X) :-
 lex(Word,C).

parse(C,P1,P) :-
 rule(C,Cs),
 parse_list(Cs,P1,P).

parse_list([ ],P,P).

parse_list([C1|Crest],P1,P) :-
 parse(C1,P1,X),
 parse_list(Crest,X,P).
```


Results

```
5 ?-  parse(np,[the,dog],[]).
   Call:  (  7) parse(np, [the, dog], []) ? creep
   Call:  (  8) rule(np, _L126) ? creep
   Exit:  (  8) rule(np, [d, n]) ? creep
   Call:  (  8) parse_list([d, n], [the, dog], []) ? creep
   Call:  (  9) parse(d, [the, dog], _L141) ? creep
   Call:  ( 10) word(d, the) ? creep
   Exit:  ( 10) word(d, the) ? creep
   Exit:  (  9) parse(d, [the, dog], [dog]) ? creep
   Call:  (  9) parse_list([n], [dog], []) ? creep
   Call:  ( 10) parse(n, [dog], _L170) ? creep
   Call:  ( 11) word(n, dog) ? creep
   Exit:  ( 11) word(n, dog) ? creep
   Exit:  ( 10) parse(n, [dog], []) ? creep
   Call:  ( 10) parse_list([], [], []) ? creep
   Exit:  ( 10) parse_list([], [], []) ? creep
   Exit:  (  9) parse_list([n], [dog], []) ? creep
   Exit:  (  8) parse_list([d, n], [the, dog], []) ? creep
   Exit:  (  7) parse(np, [the, dog], []) ? creep
Yes
```

Question: What changes are necessary to turn the recogniser into a parser?

Left Recursion

```
% Phrase-structure rules

rule(s,[np,vp]).
rule(np,[np,pp]).
rule(np,[d,n]).
rule(vp,[v,np]).
rule(vp,[v,np,pp]).
rule(pp,[p,np]).

C:  (  7) parse(np, [the, dog, near, the, dog], []
C:  (  8) rule(np, _L141) ? creep
E:  (  8) rule(np, [np, pp]) ? creep
C:  (  8) parse_list([np, pp], [the, dog, near, th
C:  (  9) parse(np, [the, dog, near, the, dog], X)
C:  ( 10) word(np, the) ? creep
F:  ( 10) word(np, the) ? creep
R:  (  9) parse(np, [the, dog, near, the, dog], X)
C:  ( 10) rule(np, _L180) ? creep
E:  ( 10) rule(np, [np, pp]) ? creep
C:  ( 10) parse_list([np, pp], [the, dog, near, th
C:  ( 11) parse(np, [the, dog, near, the, dog], _I
```


Bottom-Up Parsing

Top down depth first parsers loop on left recursive rules like

```
np -> np pp
```

One way of dealing with this is to use a bottom up parsing algorithm:

Accept a word: it's *the*
the is a D
Accept another word: it's *dog*
dog is an N
D and N make an NP
Accept another word: it's *barks*
barks is a V
V makes a VP
NP and VP make an S

Shift Reduce Parsing

The simplest bottum-up parsing algorithm is called shift-reduce, which requires two operations (shift and reduce) and a stack.

Shift-Reduce Algorithm

1. If there is remaining input, shift one word onto the stack
2. Reduce the stack repeatedly using lexical entries and grammar rules until no futher reduction is possible.
3. Go to step 1.

N.B With S/R parsing the algorithm itself discovers which kind of constituent is present. There is no need to ask for a specific category to be parsed.

Shift Reduce in Prolog

Step	Action	Stack	Input
	(Start)		[the,dog,barked]
1	Shift	[the]	[dog,barked]
2	Reduce	[d]	[dog,barked]
3	Shift	[dog,d]	[barked]
4	Reduce	[n,d]	[barked]
5	Reduce	[np]	[barked]
6	Shift	[barked,np]	
7	Reduce	[v,np]	
8	Reduce	[vp,np]	
9	Reduce	s	

- Words are shifted to the beginning of the stack, which ends up in reverse order.
- The reduce step is simplified if we also store the rules backward, so that the rule

  ```
  s --> np vp
  ```

  is stored as the fact

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

- The term [a,b|X] matches any list whose first and second elements are a and b respectively.
- The first argument directly matches the stack to which this rule applies
- The second argument is what the stack becomes after reduction.

The Program

parse(+S,?Result):
parses input string S, where Result is list of categories to which it reduces

```
parse(S,Result) :- shift_reduce(S,[],Result).
```

shift_reduce(+S,+Stack,?Result):
parses input string S, where Stack is list of categories parsed so far.

```
shift_reduce(S,Stack,Result) :-
  shift(Stack,S,NewStack,S1),  % fails if S = [ ]
  reduce(NewStack,ReducedStack),
  shift_reduce(S1,ReducedStack,Result).

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

shift(+OldStack,+S,-NewStack,-NewS):
shifts first element from S onto NewStack

```
shift(X,[H|Y],[H|X],Y).
```

reduce(+Stack,-ReducedStack):
repeatedly reduces beginning of Stack to form fewer, larger consituents

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

reduce(Stack,Stack).
```


Grammar and Lexicon

Grammar:

```
brule([vp,np|X],[s|X]).
brule([n,d|X],[np|X]).
brule([np,v|X],[vp|X]).
brule([pp,np,v|X],[vp|X]).
brule([np,p|X],[pp|X]).
brule([np,conj,np|X],[np|X]).

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

Lexicon

The lexicon is linked to the grammar by the predicate

```
word(Cat, Word) :- lex(Word,Cat).
```

so that lexical information itself is stored, as before, in the relation `lex`.

```
lex(and,conj). lex(chase,v).
lex(chases,v). lex(dog,n).
etc.
```


Results

```
6 ?- parse([the,cat],[np]).
Call:  (  7) parse([the, cat], [np]) ? creep
Call:  (  8) shift_reduce([the, cat], [], [np]) ? creep
Call:  (  9) shift([], [the, cat], _L137, _L138) ? creep
Exit:  (  9) shift([], [the, cat], [the], [cat]) ? creep
Call:  (  9) reduce([the], _L139) ? creep
Call:  ( 10) brule([the], _L151) ? creep
Call:  ( 11) word(_G309, the) ? creep
Exit:  ( 11) word(d, the) ? creep
Exit:  ( 10) brule([the], [d]) ? creep
Call:  ( 10) reduce([d], _L139) ? creep
Call:  ( 11) brule([d], _L188) ? creep
Call:  ( 12) word(_G312, d) ? creep
Fail:  ( 12) word(_G312, d) ? creep
Fail:  ( 11) brule([d], _L188) ? creep
Redo:  ( 10) reduce([d], _L139) ? creep
Exit:  ( 10) reduce([d], [d]) ? creep
Exit:  (  9) reduce([the], [d]) ? creep
Call:  (  9) shift_reduce([cat], [d], [np]) ? creep
Call:  ( 10) shift([d], [cat], _L189, _L190) ? creep
Exit:  ( 10) shift([d], [cat], [cat, d], []) ? creep
Call:  ( 10) reduce([cat, d], _L191) ? creep
Call:  ( 11) brule([cat, d], _L203) ? creep
Call:  ( 12) word(_G315, cat) ?
```


Left Corner Parsing

- **Key Idea:** accept a word, identify the constituent it marks the beginning of, and parse the rest of the constituent top down.

- **Main Advantages:**

  1. Like a bottom-up parser, can handle left recursion without looping, since it starts each constituent by accepting a word from the input string.

  2. Like a top-down parser, is always expecting a particular category for which only a few of the grammar rules are relevant. It is therefore more efficient than a plain shift-reduce algorithm.

The Algorithm

To parse a constituent of type C:

1. Accept a word W from input and determine K, its category.

2. Complete C:

   - If K=C, exit with success; otherwise

   - Find a constituent whose expansion begins with K. Call that CC. For instance, if K=d (determiner), CC could be Np, since we have `rule(np,[d,n])`

   - Recursively left-corner parse all the remaining elements of the expansion of CC (in this case, [n]).

   - Put CC in place of K, and return to step 2

Program for Left Corner Parser

- **parse(+C,+P1,+P):** *Parse a constitutent of category C between text indices P1 and P.*

```
parse(C,[W|Rest],P) :-
 word(K,W),
 complete(K,C,Rest,P).
```

- **parse_list(Cs,+P1,+P):** *Parse a list of consitutents Cs beginning between text indices P1 and P.*

```
parse_list([],P,P).
parse_list(([C|Cs],P1,P) :-
 parse(C,P1,P2),
 parse_list(Cs,P2,P).
```

- **complete(+W,+C,+P1,-P):** *Verifies that W can be the first subconstituent of C, and then left-corner parses the rest of C*

```
complete(C,C,P,P).            % if C=W, do nothing

complete(K,C,P1,P) :-
 rule(CC,[K|Rest]),
 parse_list(Rest,P1,P2),
 complete(CC,C,P2,P).
```


Results

```
4 ?- parse(np,[the,cat],[]).
Call:  (  7) parse(np, [the, cat], []) ? creep
Call:  (  8) word(_L128, the) ? creep
Exit:  (  8) word(d, the) ? creep
Call:  (  8) complete(d, np, [cat], []) ? creep
Call:  (  9) rule(_L153, [d|_G306]) ? creep
Exit:  (  9) rule(np, [d, n]) ? creep
Call:  (  9) parse_list([n], [cat], _L155) ? creep
Call:  ( 10) parse(n, [cat], _L181) ? creep
Call:  ( 11) word(_L196, cat) ? creep
Exit:  ( 11) word(n, cat) ? creep
Call:  ( 11) complete(n, n, [], _L181) ? creep
Exit:  ( 11) complete(n, n, [], []) ? creep
Exit:  ( 10) parse(n, [cat], []) ? creep
Call:  ( 10) parse_list([], [], _L155) ? creep
Exit:  ( 10) parse_list([], [], []) ? creep
Exit:  (  9) parse_list([n], [cat], []) ? creep
Call:  (  9) complete(np, np, [], []) ? creep
Exit:  (  9) complete(np, np, [], []) ? creep
Exit:  (  8) complete(d, np, [cat], []) ? creep
Exit:  (  7) parse(np, [the, cat], []) ? creep

Yes
```