A Bottom Up Parser in Prolog

Mohamed Balabel

University of Tübingen

June 22, 2016
Overview

Background

why BU in Prolog?
recap: BU parsing
recap: Left-Corner Parsing
recap: DCG notation

A simple BU parser

the basic idea
notation
procedure

A BU parser with lookup table

linking
left-recursion

Appendix: Our Parsers so far
Why BU in Prolog?

- **default procedure** when parsing in Prolog: top-down, left to right (as provided for DCG)

- **problem** with this procedure: **left-recursion**! For example:

  \[
  s \rightarrow s, \text{conj}, s. 
  \]

- **a naive solution** can be: rename those predicates by hand, that cause left-recursion

Q: Can you think of reasons why this is not a good solution?
Why BU in Prolog?

- some of the problems are:
  1. doesn’t work for every case
  2. not faithful to the theory
  3. increases the grammar size
- solution: implement a BU parser!
BU Parsing

**Basic idea** underlying different BU parsing techniques:
guide your parse steps with the input:

1. recognize low level units (terminals) first
2. build mid level representations
3. build higher level representations until you have the full sentence

Some **variants** include:

1. naive BU with backtracking
2. LR parser (Left-to-right, Rightmost derivation)
3. CYK parser
4. Tomita Parser

→ today: left-corner method (rather hybrid approach)
The **left-corner of a production rule** in a context-free grammar is the left-most symbol on the right side of the rule. For example, in the rule $A \rightarrow X\alpha$, $X$ is the left corner.

We will use the term **right span** to refer to the rest of the rhs of a rule after the left-corner. Eg. in the rule $A \rightarrow X\alpha$, $\alpha$ is the right span.
LC Parsing: Left-Corners

For our purpose, we will consider that the left-corner relation is:

- **reflexive**: each non-terminal is a left-corner of itself
- **transitive**: if A is a left-corner of B, B left-corner of C, then A is left-corner of C.

**Q**: Given the following grammar, determine the left-corners of S:

\[
\begin{align*}
S & \rightarrow \text{NP } \text{VP} \\
S & \rightarrow \text{ITJ} \\
\text{VP} & \rightarrow \text{V } \text{NP} \\
\text{NP} & \rightarrow \text{DET } \text{N}
\end{align*}
\]
LC Parsing: Left-Corners

For our purpose, we will consider that the left-corner relation is:

- **reflexive**: each non-terminal is a left-corner of itself
- **transitive**: if A is a left-corner of B, B left-corner of C, then A is left-corner of C.

Q: Given the following grammar, determine the left-corners of S:

\[
S \rightarrow \text{NP } \text{VP} \\
S \rightarrow \text{ITJ} \\
\text{VP} \rightarrow \text{V } \text{NP} \\
\text{NP} \rightarrow \text{DET } N
\]

A: set of left-corners of S includes: NP, ITJ (follow directly from the rules), DET (by transitivity) and S (by reflexivity).
LC Parsing

The procedure of LC parsing consists of four actions:

store
If the next word in the input is $a$, and $A \rightarrow a$ is a rule in the grammar, then read in $a$, and push $A \rightarrow a \cdot$ onto the top of the stack.

predict
If $A \rightarrow \alpha \cdot$ is on top of the stack, and the grammar contains a rule $X \rightarrow A \beta$, then replace the top item on the stack with $X \rightarrow A \cdot \beta$
LC Parsing

**complete**

If the top stack item is $Z \rightarrow \beta \cdot$ and the next item down is $X \rightarrow \alpha \cdot Z \gamma$, then replace those top two items with $X \rightarrow \alpha Z \cdot \gamma$

**successful configuration**

If all symbols are consumed, and a rule has been fully recognized, leaving the stack with no other active rules, then the parse has been successful.
Let’s work out an example together:

Given the following grammar, how can we parse the phrase ”the boy slayed the dragon”?

S → NP VP
NP → DET N
DET → the
N → boy
N → dragon
VP → VT NP
VT → slayed
As you have noticed, the left-corner order is indeed an **inorder tree traversal**!
DCG Notation

DCG

DCG notation is **syntactic sugar for working with difference lists**.
The difference lists are only hidden from us, but still there!
Ex:

```
s(Num) --> np(Num), vp(Num).
```

Q: How is this translated internally in Prolog?
DCG Notation

DCG
DCG notation is syntactic sugar for working with difference lists. The difference lists are only hidden from us, but still there!

Ex:

\[ s(\text{Num}) \rightarrow \text{np}(\text{Num}), \text{vp}(\text{Num}). \]

without DCG
The same rule, without any syntactic sugar, would look like this:

\[ s(\text{Num}, \text{A}, \text{C}) \rightarrow \text{np}(\text{Num}, \text{A}, \text{B}), \text{vp}(\text{Num}, \text{B}, \text{C}). \]
Basic Intuition behind our LC Parser

1. First thing we should encounter is a **terminal node**: we look up its category
   [the], [program, writes, a, program, that, halts].
   → consume the; now we have DET

2. Which **rules have this category as LC?**: we try them one by one
   → we find: NP → DET, N, OPTREL.
3. Now we show that the right span of this rule is compatible.
   → here the NP’s right span will consume ”program” as N, and nothing as OPTREL.
   → [the, program], [writes, a, program, that, halts].

4. Then we use the lhs of this rule (its category)
   → so now we look for rules that have NP as left-corner.
   Ex:
   S → NP, VP.

5. repeat until all the required symbols are consumed
A new operator

in order to conveniently refer to any part of the rule (lhs, LC of rhs, ...etc), we are going to introduce the following operator:

:- op(1200,xfx,--->).

Q: What does 1200 mean?
Notation

A new operator

in order to conveniently refer to any part of the rule (lhs, LC of rhs, ...etc), we are going to introduce the following operator:

:- op(1200, xfx, --->).

Q: What does 1200 mean?

A: Order of binding follows the order of numbers: * has precedence 400, + 500, and -- > 1200 (highest possible value).
Notation

Non-terminals
we will write rhs of rules of non-terminals as lists, not conjunctions:

\[ s \rightarrow [np, vp] \].
\[ optrel \rightarrow [] \].

Terminals
terminals will be represented as unit clauses

\[ \text{word(word, preterminal)} \].
Complete Example Augmented with Parse Tree

Grammar rules:

\[ s(s(NP,VP)) \rightarrow [np(NP), vp(VP)] \] .

\[ np(np(Det,N,Rel)) \rightarrow [det(Det), n(N), optrel(Rel)] . \]

\[ np(np(PN)) \rightarrow [pn(PN)] . \]

\[ vp(vp(TV,NP)) \rightarrow [tv(TV), np(NP)] . \]

\[ vp(vp(IV)) \rightarrow [iv(IV)] . \]

\[ optrel(rel(epsilon)) \rightarrow [ ] . \]

\[ optrel(rel(that,VP)) \rightarrow [relpro, vp(VP)] . \]
Lexicon entries:

word(that, relpro).
word(terry, pn(pn(terry))).
word(shrdlu, pn(pn(shrdlu))).
word(halts, iv(iv(halts))).
word(a, det(det(a))).
word(program, n(n(program))).
word(writes, tv(tv(writes))).
Let us suppose our goal is: to give Prolog a phrase, and get back its parse tree:

```prolog
?- parse(Tree, [a, program, writes, a, program, that, halts], []).

Tree = s(s(np(det(a), n(program), rel(epsilon)), vp(tv(writes), np(det(a), n(program), rel(that, vp(iv(halts))))))).
```
We can also tell Prolog explicitly which phrase type we expect, so that it will fail if it doesn’t match:

▶ ?- parse(np(Tree), [a, program, that, halts], []).

T = np(det(a), n(program), rel(that, vp(iv(halts)))).

▶ ?- parse(vp(Tree), [program, writes, a, program], []).

false.
Procedure: Part I/IV

Want to parse a phrase?
→ Then take the first possible leaf in the phrase + check that the phrase has this leaf as a possible LC.

parse(Phrase) -->
    leaf(SubPhrase),
    lc(SubPhrase, Phrase).
Procedure: Part II/IV

When do we have a leaf?
→ Of course if we have it as a word in the lexicon!

leaf(Cat) --> [Word], {word(Word, Cat)}.

Forgot something?
→ Oops! We forgot the epsilon rules!

leaf(Phrase) --> {Phrase --> []}.

In other words: consider the category you have on the lhs, without consuming any symbols / words.
How can we prove that a subphrase is an LC of the phrase?

→ The base case is: (remember we consider every phrase to be LC of itself):

\[ \text{lc}(	ext{Phrase}, \text{Phrase}) \rightarrow \text{[]} \].

makes sense, after all we are parsing bottom-up, so this should be the last step (in every iteration).
Otherwise: search for a rule of which the subphrase is an LC, see how far this rule can span to the right; if compatible, check that the lhs of this rule is itself indeed an LC of the superphrase.

\[
lc(\text{SubPhrase}, \text{SuperPhrase}) \rightarrow \\
\{\text{Phrase} \longrightarrow [\text{SubPhrase}|\text{Rest}]\}, \\
\text{parse\_rest}(\text{Rest}), \\
lc(\text{Phrase}, \text{SuperPhrase}).
\]
parse_rest is where we **iterate the whole process** again and again over the rest of the symbols

```
parse_rest([]) --> [].

parse_rest([Phrase|Phrases]) -->
    parse(Phrase),
    parse_rest(Phrases).
```

→ *let’s see an example in trace mode!*
Linking

We can still make our parser more efficient if we made use of information concerning what type of expression we are attempting to parse.

For instance, we don’t have to waste time trying the epsilon rule `optrel(rel(epsilon))` every time we are attempting to parse a sentence: this rule is simply not a left-corner of S (and in fact, no optrel rule is a left-corner of S).
Linking

For this reason, we will introduce the predicate link, which will serve as our lookup table:

\[
\begin{align*}
\text{link}(\text{np}(_), \text{s}(_)) . \\
\text{link}(\text{det}(_), \text{np}(_)) . \\
\text{link}(\text{det}(_), \text{s}(_)) . \\
\text{link}(\text{pn}(_), \text{np}(_)) . \\
\text{link}(\text{pn}(_), \text{s}(_)) . \\
\text{link}(\text{tv}(_), \text{vp}(_)) . \\
\text{link}(\text{iv}(_), \text{vp}(_)) . \\
\text{link}(\text{relpro}, \text{optrel}(_)) . \\
\text{link}(\text{Nonterminal}, \text{Nonterminal}) .
\end{align*}
\]

→ any nonterminal is an LC of itself
Parse Predicate with Linking

Now we just need to make use of our table in both the parse and the lc predicates:

`parse without linking`

`parse(Phrase) -->
    leaf(SubPhrase),
    lc(SubPhrase, Phrase).`

Q: How shall we alter this rule to make use of our link/2 predicate?
parse Predicate with Linking

Now we just need to make use of our table in both the parse and the lc predicates:

**parse without linking**

```
parse(Phrase) -->
    leaf(SubPhrase),
    lc(SubPhrase, Phrase).
```

**parse with linking**

```
parse(Phrase) -->
    leaf(SubPhrase),
    {link(SubPhrase, Phrase)},
    lc(SubPhrase, Phrase).
```
lc Predicate with Linking

**lc without linking**

```prolog
lc(SubPhrase, SuperPhrase) -->
{Phrase ---> [SubPhrase|Rest]},
parse_rest(Rest),
lc(Phrase, SuperPhrase).
```

**lc with linking**

```prolog
lc(SubPhrase, SuperPhrase) -->
{Phrase ---> [SubPhrase|Rest]},
{link(Phrase, SuperPhrase)},
parse_rest(Rest),
lc(Phrase, SuperPhrase).
```
So have we defeated left recursion already?

Let’s see live!
LC-Parser and Left-Recursion

We extend our program of LC-parser by adding:

- a new rule for sentence
  \[ s(s(S1,\text{CONJ},S2)) \rightarrow [s(S1),\text{conj}(\text{CONJ}),s(S2)]. \]

- new rule for conjunction
  \[ \text{conj}(\text{CONJ}) \rightarrow [\text{conj}(\text{CONJ})]. \]

- new lexical entry conjunction
  \[ \text{word}(\text{and}, \text{conj}(\text{conj}(\text{and}))). \]

- new link for s
  \[ \text{link}(s(_),s(_)). \]

Q: Is this last rule necessary? Can’t we rely on the rule \text{link}(\text{NT},\text{NT})?
LC-Parser and Left-Recursion

Q: Is this last rule necessary? Can’t we rely on the rule link(NT,NT)?

This last rule is necessary, because the general link rule link(NT,NT) requires both to be the same, whereas here the variables can be different.

But this program has a major flaw!
LC-Parser and Left-Recursion

The problem is not fatal if we try a query like:

?- parse(T,[terry, writes, a, program, that, halts, and, the, program, writes, a, program],[[]]).

It keeps trying the same solution again and again: annoying, but at least it gives the right solution.
Ambiguous Sentences are a Trap

The problem becomes obvious when we try to parse a syntactically ambiguous phrase:

?- parse(s(T),[terry, writes, a, program, and, the, program, writes, shrdlu, and, shrdlu, writes, a, program],[]).

Here the program gives us only one reading, then spends the rest of its life backtracking to this same reading but never comes to give the second.
LC-Parser and Left-Recursion

The Root of all Evil
Parsers will fail on rules of the form: X → X, and we have such a rule in our current grammar, namely:

\[
\text{conj(CONJ)} \longrightarrow [\text{conj(CONJ)}].
\]

- As a result, lc/4 can **infinitely apply this rule**!
- It seems we will have to refuge to our old ad hoc method: **rename some non-terminals**.
LC-Parser and Left-Recursion

Our program will undergo the following changes:

1. one of the 2 identical non-terminals will be renamed:
   \( \text{conj(CONJ)} \longrightarrow \text{[c(CONJ)]} \).

2. accordingly, we will add this info to our lookup table...
   \( \text{link(c(\_), conj(\_))} \).

3. ...and change the corresponding lexical entry
   \( \text{word(and, c(conj(and))}) \).

Problem solved! alomst at least. This will still over-generate, but the number of parses is finite and we get all readings.
Bottom-Up and Left-Recursion: the Bottom Line

The bottom line here is:

- we have defeated left-recursion with bottom-up parsing and can define rules like $A \rightarrow A, \alpha$
- though we have to be careful and avoid rules of the form $A \rightarrow A$
In this section we will try to sketch the development of the recognizers and parsers we have seen so far in this seminar.

- Recursive Top-Down (with append/3)
- Parsing as Deduction and DCG
- Simple Shift-Reduce
- Left-Corner
- Agend-based
Recursive Top-Down

This was our first attempt:

```
s(Z) :- np(X), vp(Y), append(X, Y, Z).
np(Z) :- det(X), n(Y), append(X, Y, Z).
vp(Z) :- v(X), np(Y), append(X, Y, Z).
vp(Z) :- v(Z).
```

```
det([the]).
det([a]).
n([woman]).
n([man]).
v([shoots]).
```
Soon we complained that the program does not use the input sentence to guide the search. We saw that the program chooses noun phrases and verb phrases and only afterwards checks whether these can be combined to form the sentence.
Recursive Top-Down

- We attempted to solve this problem by placing the append calls in front, ex:
  \[ s(Z) :- \text{append}(X,Y,Z), \ np(X), \ vp(Y). \]

- Even then we had many problems: using \texttt{append/3} a lot with uninstantiated variables, which is a source of inefficiency, as fewer steps are devoted to recognising than to using append. Besides, we had no possibility to write any left-recursive rules.
Parsing as Deduction

Then we looked at another technique that makes use of difference lists:

\[
\]

\[
s(S-SH) :- dappend(NP-NPH, VP-VPH, S-SH),
np(NP-NPH),
vp(VP-VPH).
\]

\[
np(NP-NPH) :-
dappend(Det-DetH, N-NH, NP-NPH),
det(Det-DetH),
n(N-NH).
\]

\[
vp(VP-VPH) :- v(VP-VPH).
\]
Parsing as Deduction

det(Det-DetH) :- word(Det-DetH, the).
n(N-NH) :- word(N-NH, dog).
v(V-VH) :- word(V-VH, walks).

word([Word|Rest]-Rest, Word).

We noticed that it is a lot more efficient than using append/3. Trace a query with this program, you will find it needs far less steps than with append/3.
Soon we discovered that we can rewrite the program without dappend/3 if we just carry out all necessary unification in the arguments.

As an example, we can rewrite this rule:

\[
\textit{s} (S\text{-}SH) \leftarrow \text{dappend} (NP\text{-}NPH, VP\text{-}VPH, S\text{-}SH), \\
\text{np} (NP\text{-}NPH), \\
\text{vp} (VP\text{-}VPH).
\]

as follows:

\[
\textit{s} (NP\text{-}VPH) \leftarrow \text{np} (NP\text{-}VP), \text{vp} (VP\text{-}VPH).
\]
Parsinig as Deduction

Our recognizer now looked like this:

\[
\begin{align*}
s(NP-VPH) & : -= np(NP-VP), \ vp(VP-VPH). \\
np(Det-NH) & : -= det(Det-N), \ n(N-NH). \\
vp(VP-VPH) & : -= v(VP-VPH).
\end{align*}
\]

\[
\begin{align*}
det(Det-DetH) & : -= word(Det-DetH, the). \\
n(N-NH) & : -= word(N-NH, dog). \\
n(N-NH) & : -= word(N-NH, cat). \\
v(V-VH) & : -= word(V-VH, walks).
\end{align*}
\]

\[
\begin{align*}
\text{word}([Word|Rest]-Rest, Word).
\end{align*}
\]
We made our code more legible by using 2 separate arguments instead of the */2 operator.

\[
\begin{align*}
s(P0,P) & : - np(P0,P1), \ vp(P1,P). \\
np(P0,P) & : - det(P0,P1), \ n(P1,P). \\
vP(P0,P) & : - v(P0,P). \\
det(P0,P) & : - \text{word}(P0,P,\text{the}). \\
n(P0,P) & : - \text{word}(P0,P,\text{dog}). \\
n(P0,P) & : - \text{word}(P0,P,\text{cat}). \\
v(P0,P) & : - \text{word}(P0,P,\text{walks}).
\end{align*}
\]

\text{word}([\text{Word}|\text{Rest}],\text{Rest},\text{Word}).
Afterwards we introduced the DCG notation; which is Prolog’s syntactic sugar for working with difference lists. Finally we could write Prolog rules just as we write linguistic rules (well, almost at least!).

```prolog
s --> np, vp.
np --> det, n.
vp --> v, np.
vp --> v.

det --> [the].
det --> [a].
n --> [woman].
n --> [man].
v --> [shoots].
```

Parses can be specified as **extra arguments embedded** in this notation.
Shift-Reduce

Although we could - using DCG - already overcome the problem of inefficiency in a very elegant way, one big problem remained: we could not write left-recursive rules. So we decided to try bottom-up approaches.

First, we developed a simple shift-reduce parser in 5 steps:
Step 1/5: Grammar Rules

In order to access any part of a grammar rule easily, we introduced the grammar_rule/3 predicate.

```prolog
grammar_rule(s, [np(Num), vp(Num)]).
grammar_rule(np(Num), [det(Num), n(Num)]).
grammar_rule(vp(Num), [v(Num), np(_)]).
grammar_rule(vp(Num), [vp(Num), pp]).
```
Step 2/5: Lexicon

The lexicon/2 predicate lists all \( \langle \text{word}, \text{category} \rangle \) tuples in our lexicon.

```
lexicon(the, det(sg)).
lexicon(the, det(pl)).
lexicon(man, n(sg)).
lexicon(men, n(pl)).
```
Shift-Reduce

Step 3/5: Shift

When we recognize a terminal (i.e. a word) we take it out from the buffer, and push its category on the stack.

sr([Head|Tail], Stack, Goal) :-
    lexicon(Head, Cat),
    sr(Tail, [Cat|Stack], Goal).

What is this third argument (Goal)? Hang on a second!
Step 4/5: Reduce

When 2 categories accumulate on top of the stack, and there is a rule that can reduce both rules, we exchange both categories with the category of that rule.

\[
\text{sr}(\text{Buffer}, [\text{Cat}_1, \text{Cat}_0 | \text{Stack}], \text{Goal}) \leftarrow \\
\quad \text{grammar_rule}((\text{Cat}, [\text{Cat}_0, \text{Cat}_1]), \\
\quad \text{sr}(\text{Buffer}, [\text{Cat} | \text{Stack}], \text{Goal}).
\]

For example, if we have Det and N on top of the stack, we can exchange them with NP.
Shift-Reduce

Step 5/5: Successful Configuration

If all symbols are consumed, and there is only one category left on the stack, then we have reached our **Goal**.

\[
\text{sr([], [Goal], Goal).}
\]

This is the **bottom of the recursion**. This goal is the **category of the phrase** we wanted to parse. It now just has to be **percolated upwards**.
We achieved a minor improvement by indexing the grammar rules by the second part of their RHS. This way, we made use of Prolog’s **first argument indexing**.

```prolog
term_expansion(grammar_rule(LHS, [RHS0, RHS1]),
               grammar_rule_(RHS1, LHS, RHS0)).
```

**Q:** Why the second, and not the first part of the RHS?
Shift-Reduce

Well, the second part of the RHS is what we will always find on top of the stack every time we want to reduce. And that's the only thing we will now alter in our program!

\[ sr(\text{Buffer}, [\text{Cat0, Cat1}|\text{Stack}], \text{Goal}) :- \]
\[ \text{grammar_rule}_{-}(\text{Cat0, Cat, Cat1}), \]
\[ sr(\text{Buffer}, [\text{Cat}|\text{Stack}], \text{Goal}). \]
A subtle problem has nevertheless pertained through all the above-mentioned programs. To illustrate the problem consider the following example rules:

\[ \text{vp} \rightarrow \text{dv}, \ \text{np}, \ \text{pp}(\text{to}). \]
\[ \text{vp} \rightarrow \text{dv}, \ \text{np}, \ \text{np}. \]

which should match "gave a book to Peter" and "gave Peter a book" respectively.

What will happen, if for example the first rule fails during a certain parse and Prolog moves to the second?
Agenda-Based Parsers

Clearly all the **work done to prove the first goals in the rule** \((dv, np)\) **will be undone**. What a loss!

In the second rule, poor Prolog will have to start proving \(dv\) and \(np\) once again from the scratch!

There must be a solution for this inefficiency.

For this reason, we will look at **agenda-based approaches** to parsing.
References
