Bottom-up parsing

The basic idea:

1. We will attempt to build a tree for a sentence $s$.
2. Whereas the top-down method was prediction oriented, with a great number of steps being made independently of the data, the bottom-up method is more data-driven.
3. We begin by assigning syntactic categories to words.
4. Then, whenever we have adjacent categories making up the right-hand side of some rule, we can infer that they go together to make the left-hand category of that rule.
5. If we can build up a constituent of type $S$ over the string $s$ in this manner, we have successfully parsed $s$.

Example 1 Let $G = \langle \{S, NP, VP, V, Det, N\}, \{john, tiger, killed, a\}, S, P\rangle$, where $P$ is:

\[
S \rightarrow NP \ VP \\
NP \rightarrow \ \text{john} \mid \text{Det} \ \text{N}
\]

\[
\text{Det} \rightarrow a \\
N \rightarrow tiger \\
VP \rightarrow V \ NP \\
V \rightarrow killed
\]

Let us try to parse the string $john \ killed \ a \ tiger$.

1. We begin by labeling $john$ as a $NP$:

\[
NP \\
\mid \\
john
\]

2. as there are no rules with just an $NP$ as their right-hand side, we look to the next word, and assign it an appropriate label:

\[
NP \quad V \\
\mid \quad \mid \\
john \quad \text{killed}
\]

3. Again, $NP \ V$ doesn’t constitute any right hand rule side, so we look at the next word:
4. Still, no right hand side looks like either $NP \ V \ Det$ or $V \ Det$ or $Det$, so we look again at the next word:

$$NP \ V \ Det \ N$$

$$\begin{align*}
&john \\
&killed \\
&a \\
&tiger
\end{align*}$$

5. Now there is a rule whose right hand side is $Det \ N$, so we can reduce two trees to one labeled $NP$:

$$NP \ V \ Det \ N$$

$$\begin{align*}
&john \\
&killed \\
&Det \\
&a \\
&tiger
\end{align*}$$

6. We reduce the two trees $V$ and $NP$ to $VP$:

$$NP \ V \ NP$$

$$\begin{align*}
&john \\
&killed \\
&Det \\
&a \\
&tiger
\end{align*}$$

7. Finally, we reduce the $NP$ and $VP$ trees to $S$. As we have made an $S$ tree spanning the entire input, we have successfully parsed the string!

$$S \ NP \ VP \ NP$$

$$\begin{align*}
&john \\
&killed \\
&Det \\
&a \\
&tiger
\end{align*}$$
Formulating bottom-up parsing incrementally

- Our parser minimally needs to keep track of:
  1. what parts of the input remain to be looked at
  2. what symbols the trees we have thus far constructed are rooted in

- We represent this in terms of a parser state \( c = \langle s, \gamma \rangle \), which contains:
  1. a list of input words \( s \)
  2. a list of symbols \( \gamma \)

- We can thus re-formulate our bottom-up parsing strategy more economically:

<table>
<thead>
<tr>
<th>operation</th>
<th>effect</th>
<th>condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce</td>
<td>( \langle s, \delta' \gamma \rangle \Rightarrow \langle s, A\gamma \rangle )</td>
<td>( A \rightarrow \delta \in G )</td>
</tr>
<tr>
<td>shift</td>
<td>( \langle as, \gamma \rangle \Rightarrow \langle s, a\gamma \rangle )</td>
<td></td>
</tr>
</tbody>
</table>

- the initial configuration on a string \( s \) is \( \langle s, \epsilon \rangle \)

- a final configuration is one in which we’ve seen the entire string, and have a single tree of category \( S \) left: \( \langle \epsilon, S \rangle \)

Example 2 Let \( G = \langle \{ S, NP, VP, V, Det, N \}, \{ john, tiger, killed, a \}, S, P \rangle \), where \( P \) is:

\[
S \rightarrow NP \ VP \\
NP \rightarrow john \mid Det \ N \\
Det \rightarrow a \\
N \rightarrow tiger \\
VP \rightarrow V \ NP \\
V \rightarrow killed
\]

Let us try to parse the string \( john \ killed \ a \ tiger \).

start: \( \langle john \ killed \ a \ tiger, \epsilon \rangle \)

shift: \( \langle killed \ a \ tiger, \ john \rangle \)

reduce: \( \langle killed \ a \ tiger, NP \rangle \)

shift: \( \langle a \ tiger, killed \ NP \rangle \)

reduce: \( \langle a \ tiger, V \ NP \rangle \)

shift: \( \langle tiger, a \ V \ NP \rangle \)

reduce: \( \langle tiger, Det \ V \ NP \rangle \)

shift: \( \langle \epsilon, tiger \ Det \ V \ NP \rangle \)
reduce: \( \langle \epsilon, NP \ V \ NP \rangle \)
reduce: \( \langle \epsilon, VP \ NP \rangle \)
reduce: \( \langle \epsilon, S \rangle \)

- This gives us a way to check whether a string has a parse
- How do we recover that parse?
- We extend our parser state to include a record of which rules were used to rewrite the left-most non-terminal at each step:

<table>
<thead>
<tr>
<th>operation</th>
<th>effect</th>
<th>condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce</td>
<td>( \langle s, \delta \gamma, \omega \rangle \implies \langle s, A\gamma, \omega(A, \delta) \rangle )</td>
<td>( A \rightarrow \delta \in G )</td>
</tr>
<tr>
<td>shift</td>
<td>( \langle as, \gamma, \omega \rangle \implies \langle s, a\gamma, \omega \rangle )</td>
<td></td>
</tr>
</tbody>
</table>

- The tree recovered by a bottom-up parser is represented *differently* than the one recovered by its top-down counterpart.

**TD**: derivation is left-most (‘LL’)

**BU**: derivation is right-most, but backwards (‘LR’)

### Implementing the bottom-up parser

We can use the SAME configuration we defined before.

```ocaml
type configuration = { stack : label list; input : string list; output : production list };;
```

**Exercise 1** Define a function implementing *shift*.

```ocaml
let shift conf = match conf.input with
  | y::ys        -> Some {input=ys;stack=T(y)::conf.stack;output=conf.output}
  | _            -> None;;
```

We can implement *reduce* by giving it an argument telling it which production to use:

```ocaml
let reduce prod conf = match prod with
  | (lhs,[])     -> Some {input=conf.input;stack=lhs::conf.stack;output=prod::conf.output}
  | (lhs,rhs)    -> match check_n conf.stack (List.rev rhs) with
                     | Some (rest)   -> Some {input=conf.input;stack=lhs::rest;output=prod::conf.output}
                     | None          -> None;;

  val reduce : production -> configuration -> configuration option = <fun>
```

4
This relies on the auxiliary function \texttt{check\_n} which checks whether the right hand side of a rule (its second argument) is identical with the top of a stack (its first argument):

\begin{verbatim}
let rec check_n stack rev_rhs = match stack,rev_rhs with
    | L(x)::xs,y::ys when x = y -> check_n xs ys
    | _,[] -> Some stack
    | _ -> None;;
\end{verbatim}

val check_n : stack\_symbol list \rightarrow label list \rightarrow stack\_symbol list option = <fun>

More generally, we can implement a function that, given a grammar, maps a parser configuration to the list of all configurations that result from applying either shift or reduce to it:

\begin{verbatim}
let getNextsBU g conf =
    let shifted =
        match shift conf with
            | Some (c) -> [c]
            | None -> []
    in
    let foldFunction p conf_list = match reduce p conf with
        | None -> conf_list
        | Some (c) -> c::conf_list
    in
    ProdSet.fold foldFunction g.p shifted;;
\end{verbatim}

val getNextsBU : grammar \rightarrow configuration \rightarrow configuration list list = <fun>

Problems:

TD: \hspace{1em} left-recursion

BU: empty categories

\textbf{Example 3} Consider the grammar \textit{anbn}. When we try to make the tree of parser configurations when parsing the sentence “ab” here is what happens:

\begin{verbatim}
# makeTree (getNextsBU anbn) (make\_initial anbn ["a";"b"]);
Stack overflow during evaluation (looping recursion?).
\end{verbatim}

Why?
Figure 1: The search space for the string “aacbb”