Top-down parsing

Here are important aspects of the top-down parsing strategy we discussed last time.

1. There is a “parser state”, or configuration of the parser, that keeps track of
   (a) the remainder of the input string which needs to be accounted for
   (b) the remainder of the yield of the currently constructed parse tree
   (c) the productions used so far (from which the current parse tree can be constructed)
   \( \langle s, \alpha, p \rangle \)

2. There were two key operations—predict and scan—that affected the state of the parser as follows.

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

3. The initial configuration on a string \( s \) is
   \( \langle s, S, \epsilon \rangle \)
   where \( S \) is the start symbol of our grammar

4. The final configuration is one in which we’ve scanned the entire string, and have no non-terminal leaves left to expand (predict) and have a sequence of production rules:
   \( \langle \epsilon, \epsilon, p_1p_2 \cdots p_n \rangle \)

Implementing top-down parsing in Ocaml

First we define a configuration:

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

Exercise 1 Define a function `scan` which takes a configuration as input and returns a configuration as output if scan can apply and otherwise raises an exception Failure(“Scan cannot apply”). It is not necessary to match the longest common prefix. Write the function so that if the first element of the input matches the first element of the stack, they are dropped.

```ocaml
let scan conf = match (conf.input,conf.stack) with
  | x::xs,T(y)::ys when x = y -> {input=xs;stack=ys;output=conf.output}
  | _ -> raise Failure("Scan cannot apply");;
```

Ocaml comes pre-defined with the `option` type.

```ocaml
type 'a option = Some of 'a | None
```
**Exercise 2** Rewrite the *scan* function above to take a configuration as input and outputs a configuration option.

```
let scan conf = match (conf.input, conf.stack) with
    | x::xs, T(y)::ys when x = y ->
        Some {input = xs; stack = ys; output = conf.output}
    | _ -> None
```

**Exercise 3** Implement *predict* by giving it an argument telling it which production to use. Therefore, *predict* should take a configuration and production as input and output a configuration option.

```
let predict prod conf = match conf.stack, prod with
    | N(x)::xs, (N(y), ys) when x = y ->
        Some {stack = ys @ xs; input = conf.input; output = prod::conf.output}
    | _ -> None
```

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 scan or predict to it:

```
let getNexts g conf =
    match scan conf with
    | Some (d) -> [d]
    | None ->
        let foldFunction p a = match predict p conf with
            | None -> a
            | Some (d) -> d::a
        in
        ProdSet.fold foldFunction g.p []
```

**Example 1** Let \( G = \langle \{ S \}, \{ a, b \}, S, \{ S \rightarrow aSb, S \rightarrow \epsilon \} \rangle \)

Let us try to parse the string `aabb`.

1. The initial configuration:

\[
\{ \text{stack = [N "S"]}; \text{input = ["a"; "a"; "b"; "b"]}; \text{output = [\]} \}
\]

2. we get the configurations accessible from this one (to be known as `conf`) in a single step:

```
# getNexts anbn conf;;
- : configuration list =
  [{\text{stack = [T "a"; N "S"; T "b"]}; \text{input = ["a"; "a"; "b"; "b"]};
    \text{output = [(N "S", [T "a"; N "S"; T "b"])]}};
  {\text{stack = []}; \text{input = ["a"; "a"; "b"; "b"]}; \text{output = [(N "S", [])]}]
```

3. We repeat this procedure, applied to the first of the possible results:
4. This time there is only one next step:

```ocaml
# getNexts anbn conf;;
- : configuration list =
  [{stack = [T "a"; N "S"; T "b"; T "b" ];
   input = ["a"; "b"; "b" ];
   output = [(N "S", [T "a"; N "S"; T "b"])]}]
```

5. We repeat this procedure, applied again to the first of the possible results:

```ocaml
# getNexts anbn conf;;
- : configuration list =
  [{stack = [T "a"; N "S"; T "b"; T "b" ];
   input = ["a"; "b"; "b" ];
   output = [(N "S", [T "a"; N "S"; T "b"]); (N "S", [T "a"; N "S"; T "b"])]}]
```

6. The next step:

```ocaml
# getNexts anbn conf;;
- : configuration list =
  [{stack = [T "a"; N "S"; T "b"; T "b" ];
   input = ["a"; "b"; "b" ];
   output = [(N "S", [T "a"; N "S"; T "b"]); (N "S", [T "a"; N "S"; T "b"]);
   (N "S", [T "a"; N "S"; T "b"])]}];
```

7. This time, we want the second result:

```ocaml
# getNexts anbn conf;;
- : configuration list = [{stack = [T "b" ];
   input = ["b" ];
   output = [(N "S", []); (N "S", [T "a"; N "S"; T "b"])])}]
```

8. The next step:

```ocaml
# getNexts anbn conf;;
- : configuration list = [{stack = [ ];
   input = [ ];
   output = [(N "S", []); (N "S", [T "a"; N "S"; T "b"])])}]
```

9. This configuration is a final configuration (stack and input are both empty). The parse tree is contained in the output field of the configuration, in reverse order.
# displayTree (List.rev conf.output);;
- : tree =
  Node (N "S",
    [Leaf (T "a");
     Node (N "S",
       [Leaf (T "a");
        Node (N "S", []);
        Leaf (T "b")]);
    Leaf (T "b")])

Which configuration to use?

- At some of the steps in the previous example there were multiple configurations which could be got to from the current one.

- How can we decide which configuration to explore next?

- Recall that (getNexts anbn) maps configurations to lists of configurations:

# getNexts anbn;;
- : configuration -> configuration list = <fun>

- Note that we can define trees so as to have a similar structure:

  type 'a tree = Node of 'a * 'a tree list;;

- We can view (getNexts anbn) as telling us about parent-child relations in a configuration tree!

  let rec makeTree nexts conf =
    let nextConfs = nexts conf
    in
    match nextConfs with
    | [] -> Leaf conf
    | _ -> Node (conf, List.map (makeTree nexts) nextConfs);;

Searching Trees: Finding one answer

**Breadth-first** search explores a tree of possibilities one level at a time. All nodes equidistant from the root are explored before any further ones are explored. Distance from the root is measured in “generations” invoking the parent-child relation.
Figure 1: The search space for the string “aabb”
Depth-first Depth first search explores a branch until it gets to a leaf, backtracks minimally to explore another branch to the next leaf and so on.

Problems:

1. finding all parses (easy)

   let rec explore update next stop confs = match confs with
   | c::cs -> if stop c then (Some (List.rev c.output))
   |     else explore update next stop (update (next c) cs)
   | [] -> None;;

Breadth-first

   let bfs = explore (fun x y -> y@x);;

   let bfParser g s = bfs (getNexts g) is_final
   [make_initial g s];;

Depth-first

   let dfs = explore (@);;

   let dfParser g s = dfs (getNexts g) is_final
   [make_initial g s];;

2. left-recursion

Example 2 Consider the grammar \(ab^* = \langle \{S\}, \{a, b\}, S, \{S \rightarrow Sb, S \rightarrow a\} \rangle\). This is what happens when we try to make the tree of parser configurations when parsing the sentence “a”:

   # makeTree (getNexts ab_star) (make_initial ab_star ["a"]);
   Stack overflow during evaluation (looping recursion?).

   Why?