In this assignment, we build an LL(1) parser. We use the following OCaml types to define the notion of grammar:

```
type terminal = string

type non_terminal = string

type symbol =
  | Terminal of terminal
  | NonTerminal of non_terminal

type production = symbol list

type rule = non_terminal * production

type grammar = {
  start : non_terminal;
  rules: rule list;
}
```

Subsequently, we always assume that all non-terminals are accessible and productive. We also assume that the starting symbol S' is associated with a single rule of the form $S' \to S'\#$ where $\#$ is a terminal symbol not appearing in the other rules of the grammar. Thus, the example given in class

$$\begin{array}{rcl} E & \to & TE' \\ E' & \to & +\,TE' \\ & | & \varepsilon \\ T & \to & FT' \\ T' & \to & *\,FT' \\ & | & \varepsilon \\ F & \to & (E) \\ & | & \texttt{int} \end{array}$$

will be represented by the OCaml value:

```
let g_arith =
  { start = "S'";
    rules = [ "S'", [ NonTerminal "E"; Terminal "#" ];
              "E", [ NonTerminal "T"; NoTerminal "E'"; ];
              "E'", [Terminal "+"; NoTerminal "T"; NoTerminal "E'"; ];
              "E'", [ ];
              "T", [ NonTerminal "F"; NoTerminal "T'"; ];
              "T'", [Terminal "*"; NoTerminal "F"; NoTerminal "T'"; ];
              "T'", [ ];
              "F", [Terminal "("; NonTerminal "E"; Terminal ")"; ];
              "F", [ Terminal "int" ]; ] }
```

1 Fixed-point calculation

1.1

To facilitate the fixed point calculations in the following questions, write a function

```
val fixpoint: ('a -> 'a * bool) -> 'a -> 'a
```

such that `fixpoint f x` iterates the function `f` from the value `x` as long as the boolean returned by `f` is `true`.

2 Calculating the nulls

We are given the following OCaml module to represent sets of non-terminals:

```
module Ntset = Set.Make(String)
type nulls = Ntset.t
```

2.1

Write a function `is_null_production` that, given the set of non-terminals that can be derived into the empty word, determines whether a word can be derived into the empty word.

```
val is_null_production: nulls -> production -> bool
```

(This is the NULL(α) function from the course.) We can use `List.for_all`.

2.2

Deduce a function `null : grammar -> nulls` which calculates the set `nulls` of non-terminals of a given grammar. We will use the `fixpoint` function in the following way:

```
let null g =
  let step nulls =
   (* ...we calculate the new set of nulls and
       we indicate by a boolean whether there has been a change... *)
  in
  fixpoint step Ntset.empty
```

Note: It is of course possible to use `Ntset.equal` to determine whether the set `nulls` has changed, but it is also possible to be more precise by using `is_null_production` only on rules whose left-hand side is not yet in `nulls` and then detecting whether `nulls` needs to be modified.

To visualize the result, we can use the following code which defines a pretty-printer `pp_nulls` for all nulls:

```
let pp_non_terminal fmt s = Format.fprintf fmt "%s" s

let pp_iter iter pp_elt fmt =
  let first = ref true in
  iter (fun elt ->
      if not !first then Format.fprintf fmt ",@ " else first := false;
      pp_elt fmt elt)

let pp_nulls fmt =
  Format.fprintf fmt "@[%a@]" (pp_iter Ntset.iter pp_non_terminal)
```

We can test by comparing the result of:

```
let () =
  let nulls_arith = null g_arith in
  Format.printf "null: %a@." pp_nulls nulls_arith
```

with the current calculated set.

3 Calculation of the firsts

We are given the following OCaml modules to represent sets of terminals (`Tset`) and dictionaries indexed by non-terminals (`Ntmap`):

```
module Ntmap = Map.Make(String)
module Tset = Set.Make(String)
```

The sets of primes that we will calculate in this question will therefore have the following type:

```
type firsts = Tset.t Ntmap.t
```

that is, a dictionary that associates each non-terminal symbol with a set of terminal symbols.

3.1

Write a function `val empty_map : grammar -> Tset.t Ntmap.t` which constructs a dictionary associating each non-terminal in the grammar with an empty set of terminals.

3.2

Write a function

```
val first_production_step: nulls -> firsts -> production -> Tset.t
```

which computes the set of primes of a production, given the set of zero non-terminals and a dictionary for the primes of each non-terminal. (This is the FIRST(α) function from the course.)

Deduce a function `val first: grammar -> nulls -> firsts` which calculates the set of firsts of the non-terminals of a grammar, given the set `nulls` of non-terminals of this grammar (the latter being passed as an argument to avoid being recalculated several times). We will use the `fixpoint` function and determine whether the set of firsts is modified using the `Tset.subset` function which determines the set inclusion.

To visualize the result, we can use the following code which defines a pretty-printer `pp_firsts` for all the firsts:

```
let pp_iter_bindings iter pp_binding fmt =
  let first = ref true in
  iter (fun key elt ->
      if not !first then Format.fprintf fmt "@\n" else first := false;
      pp_binding fmt key elt)

let pp_terminal fmt s = Format.fprintf fmt "%s" s

let pp_firsts fmt =
  Format.fprintf fmt "@[%has@]"
  @@ pp_iter_bindings Ntmap.iter (fun fmt nt ts ->
         Format.fprintf fmt "@[%a -> {%a}@]" pp_non_terminal nt
           (pp_iter Tset.iter pp_terminal)
           ts)
```

We can test by comparing the result of:

```
let () =
  let nulls_arith = null g_arith in
  let firsts_arith = first g_arith nulls_arith in
  Format.printf "first: %a@." pp_firsts firsts_arith
```

with the current calculated set.

4 Calculating the follows

Finally, we will calculate the follows of the grammar. The set of follows is represented by the same OCaml type as the first:

```
type follows = Tset.t Ntmap.t
```

4.1

Write a function

```
val follows: grammar -> nulls -> firsts -> follows
```

which calculates the following of a grammar, given its null non-terminals and its firsts.

We can adopt the following scheme:

```
let follow g nulls firsts =
  let update (follows,b) nt follow_nt =
    (* ...
     ... updates the follows table with nt -> follow_nt
     ...and replaces b with true if the table has been modified
     ... *)
  in
  let rec update_prod ((follows,b) as acc) nt p =
     (*...
     ... examines the production nt -> p of the grammar
     ... and updates the pair (follows,b) for any non-terminal X of p
     ... *)
  in
  let step follows =
    List.fold_left
      (fun acc (nt,p) -> update_prod acc nt p)
      (follows,false) g.rules
  in
  fixpoint step (empty_map g)
```

To visualize the result, we can reuse the `pp_firsts` function for all the following ones, and compare the result of

```
let pp_follows = pp_firsts

let () =
  let nulls_arith = null g_arith in
  let firsts_arith = first g_arith nulls_arith in
  let follows_arith = follow g_arith nulls_arith firsts_arith in
  Format.printf "follow: %a@." pp_follows follows_arith
```

with the current calculated set.

5 Construction of the LL(1) table

We give ourselves the following OCaml types to represent the dictionaries indexed by terminals and the sets of productions:

```
module Tmap = Map.Make(String)
module Pset = Set.Make(struct type t = production let compare = compare end)
```

We then define the following type for the top-down analysis tables:

```
type expansion_table = Pset.t Tmap.t Ntmap.t
```

Such a table is therefore a dictionary associating a set of productions with each non-terminal and then with each terminal. Both dictionaries are sparse: when a row or column of the table is empty, there is no corresponding entry in the table.

5.1

Write a function

```
val add_entry: expansion_table -> non_terminal -> terminal -> production ->
↪  expansion_table
```

which adds an entry to the table. Care will be taken to correctly handle the case of a first occurrence of a row or column.

5.2

Write a function

```
val expansions: grammar -> expansion_table
```

which calculates the top-down parsing table of a given grammar.

We can test with the following grammar (which characterizes words containing as many a as b):

```
let g1 = {
  start = "S'";
  rules = ["S'", [NonTerminal "S"; Terminal "#"];
           "S", [];
           "S", [Terminal "a"; NoTerminal "A"; NoTerminal "S"];
           "S", [Terminal "b"; NoTerminal "B"; NoTerminal "S"];
           "A", [Terminal "a"; NoTerminal "A"; NoTerminal "A"];
           "A", [Terminal "b"];
           "B", [Terminal "b"; NoTerminal "B"; NoTerminal "B"];
           "B", [Terminal "a"];
         ] }

let table1 = expansions g1
```

To visualize the result, we can use the following code which defines a pretty-printer `pp_table` for the expansion tables:

```
let pp_symbol fmt = function
  | Terminal s -> Format.fprintf fmt "\"%s\"" s
  | NonTerminal s -> Format.fprintf fmt "%s" s

let rec pp_production fmt = function
  | [] -> ()
  | [x] -> pp_symbol fmt x
  | x :: l -> Format.fprintf fmt "%a %a" pp_symbol x pp_production l

let pp_table fmt t =
  let print_entry cp =
```

```
    Format.fprintf fmt " %s: @[%a@]@\n" c pp_production p in
  let print_row nt m =
       Format.fprintf fmt "@[Expansions for %s:@\n" nt;
       Tmap.iter(fun c rs -> Pset.iter(print_entry c) rs) m;
       Format.fprintf fmt "@]" in
  Ntmap.iter print_row t
```

In the example above, the result of

```
  let () = Format.printf "%a@." pp_table table1
```

must be as follows:

```
Expansions for A:
  a: "a" AA
  b: "b"
Expansions for B:
  a: "a"
  b: "b" BB
Expansions for S:
  #:
  a: "a" AS
  b: "b" BS
Expansions for S':
  #: S "#"
  a: S "#"
  b: S "#"
```

Also test with the `g_arith` grammar given at the very beginning of this topic.

```
let table_arith = g_arith expansions
let () = Format.printf "%a@." pp_table table_arith
```

6 LL(1) characterization

6.1

Write a function `is_ll1 : expansion_table -> bool` which determines whether the expansion table contains at most one rule per case (which, by definition, then characterizes the grammar as belonging to the LL(1) class).

Test with

```
let() = assert(is_ll1 table1)
let () = assert (is_ll1 table_arith)
```

Also test with a grammar of your choice that is not LL(1).

7 String recognition

7.1

Write a function

```
val analyze: non_terminal -> expansion_table -> string list -> bool
```

which determines whether a word is recognized by a grammar, given its starting non-terminal and its expansion table. (We will not worry about whether the table corresponds to an LL(1) grammar and, in case of ambiguity, we will choose a production at random with `Pset.choose`.) We will not forget to add the symbol `"#"` at the end of the word. Note: the analysis is successful if and only if we arrive at a stack and an entry both equal to the empty list `[]` , since we have added a rule $S' \to S\#$.

We can test with the `g1` grammar using the following code

```
let explode s =
  let n = String.length s in
  let rec make i = if i = n then [] else String.make 1 s.[i] :: make (i+1) in
  make 0

let test1 s = analyze g1.start (g1 expansions) (explode s)

let() = assert(test1 "")
let() = assert(test1 "ab")
let() = assert(test1 "ba")
let() = assert(test1 "abab")
let() = assert(test1 "aaabbb")
let() = assert(test1 "aaababababbabab")

let () = assert (not (test1 "a"))
let () = assert (not (test1 "b"))
let () = assert (not (test1 "aab"))
let () = assert (not (test1 "aaabbba"))
let () = assert (not (test1 "aaabbbaabab"))
let () = assert (not (test1 "aaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbb"))
```

8 Bootstrap: the grammar of grammars

We are given the following grammar:

```
let g_gram =
  { start = "S'";
    rules = [ "S'", [ NonTerminal "S"; Terminal "#" ];
              "S", [ NonTerminal "R" ];
              "S", [NonTerminal "R"; Terminal ";"; NonTerminal "S" ];
              "R", [Terminal "ident"; Terminal "::="; NoTerminal "P"];
```

```
              "P", [ NonTerminal "W" ];
              "P", [ NonTerminal "W"; Terminal "|"; NonTerminal "P" ];
              "W", [ ];
              "W", [ NonTerminal "C"; NoTerminal "W";];
              "C", [Terminal "ident"];
              "C", [Terminal "string"];
            ] }
```

This is the grammar of grammars, with the following meaning for the different non-terminals:

- `S` = sequence of rules separated by semicolons;
- `R` = rules for a non-terminal;
- `P` = productions separated by vertical bars;
- `W` = production right member;
- `C` = terminal or non-terminal symbol.

Check that it is not LL(1):

```
let table_gram = g_gram expansions
let () = Format.printf "%a@." pp_table table_gram
let () = assert (not (is_ll1 table_gram))
```

Propose an LL(1) grammar recognizing the same language.