Sum type
Truth table

Reminder : it is strictly FORBIDDEN to use the @ operator and the `List` module.

```
# truth_table "a|b|c|d|e";;
 a b c d e
 T T T T T   T
 T T T T F   T
 T T T F T   T
 T T T F F   T
 T T F T T   T
 T T F T F   T
 T T F F T   T
 T T F F F   T
 T F T T T   T
 T F T T F   T
 T F T F T   T
 T F T F F   T
 T F F T T   T
 T F F T F   T
 T F F F T   T
 T F F F F   T
 F T T T T   T
 F T T T F   T
 F T T F T   T
 F T T F F   T
 F T F T T   T
 F T F T F   T
 F T F F T   T
 F T F F F   T
 F F T T T   T
 F F T T F   T
 F F T F T   T
 F F T F F   T
 F F F T T   T
 F F F T F   T
 F F F F T   T
 F F F F F   F
- : unit = ()
```

Stage 0 - Bundle

Sum type

```
# type 'a bundle
    | Empty
    | Item of 'a * 'a bundle ;;
```

0.1 Empty

Write `empty_bundle`, which creates an empty bundle.

```
val empty_bundle : unit -> 'a bundle = <fun>

# empty_bundle();;
- : 'a bundle = Empty
```

0.2 Is Empty ?

Write `empty_bundle`, which tests if bundle is empty.

```
val is_empty : 'a bundle -> bool = <fun>

# is_empty Empty;;
- : bool = true
# is_empty (Item ('a', Empty));;
- : bool = false
```

0.3 Constructor

Write `cons`, which adds an element on top of our bundle.

```
val cons : 'a bundle -> 'a -> 'a bundle = <fun>

# cons Empty 4;;
- : int bundle = Item (4, Empty)
# cons (Item (4, Empty)) 6;;
- : int bundle = Item (6, Item (4, Empty))
```

0.4 The first

Write `head`, which returns the last element added to our bundle if it is possible, an exception otherwise.

```
val head : 'a bundle -> 'a = <fun>

# head Empty;;
Exception: Failure "Head failed: empty bundle".
# head (Item (5, Empty));;
- : int = 5
```

0.5 The others

Write `tail` which returns all the elements except the last one added to our bundle if it is possible, an exception otherwise.

```
val tail : 'a bundle -> 'a bundle = <fun>

# tail (Item ('a', (Item ('r', Empty))));;
- : char bundle = Item ('r', Empty)
```

Stage 1 - Boolean expression

Sum type

```
# type boolean =
    | True | False
    | Var of string
    | Not of boolean
    | And of boolean * boolean
    | Or  of boolean * boolean ;;


(* a || b || c || d || e ||f *)
# Or (Or (Or (Or (Or (Var "f", Var "e"), Var "d"), Var "c"), Var "b"), Var "a")
```

1.1 Value ?

Write `value` which returns the first value corresponding to the requested identifier in a cartesian (identifier,value) list, an error if not possible.

```
val value : string -> (string * 'a) list -> 'a = <fun>

# value "a" [("b", False); ("a", True)];;
- : boolean = True

# value "c" [("b", False); ("a", True)];;
Exception: Failure "Unbound Var: c".
```

1.2 Extraction

Write `extract`, which extracts the list of unique identifiers from a `boolean` expression.

```
val extract : boolean -> string list = <fun>

# extract (Or( (And (Var "a", Var "b")), (And (Var "c", Var "a"))));;
- : string list = ["c"; "b"; "a"]
```

1.3 Generator

Write `generate`, which generates from an identifiers list all combinations of values `boolean True` or `boolean False`.

```
val generate : 'a list -> ('a * boolean) list list = <fun>

# generate ["a"];;
- : (string * boolean) list list = [[("a", True)]; [("a", False)]]

# generate ["a";"b"];;
- : (string * boolean) list list =
[[("a", True); ("b", True)]; [("a", True); ("b", False)];
 [("a", False); ("b", True)]; [("a", False); ("b", False)]]
```

1.4 Evaluation

Write `eval`, which evaluates or `boolean` expression with the given values for our identifiers.

```
val eval : boolean -> (string * boolean) list -> boolean = <fun>

# eval (And (Var "a", Var "b")) [("b", False); ("a", True)];;
- : boolean = False
```

Write `evaluate`, which generates the truth table of our expression. We will have to test all combinations of values for our identifiers.

```
val evaluate : boolean -> ((string * boolean) list * boolean) list = <fun>

# evaluate (And (Var "a", Var "b"));;
- : ((string * boolean) list * boolean) list =
[([("a", True); ("b", True)], True); ([("a", False); ("b", True)], False);
 ([("a", True); ("b", False)], False); ([("a", False); ("b", False)], False)]
```

1.5 Display

Write `display`, which properly displays the truth table as the example shows.

```
# display ([([("a", True); ("b", True)], True); ([("a", True); ("b", False)], True);
 ([("a", False); ("b", True)], True); ([("a", False); ("b", False)], False)]);;
 a b
 T T   T
 T F   T
 F T   T
 F F   F
- : unit = ()
```

Stage 2 - Parsing

The purpose of this section will be to convert a *simple* boolean expression (represented as a `string`) to its boolean expression (represented as a `boolean`).

We will use the following syntax :
- an identifier is represented by a single character `['a'..'z']` or `['A'..'Z']`
- the logical negation will be represented by the character `'!'`, its priority will be 3
- the logical conjunction will be represented by the character `'&'`, its priority will be 2
- the logical disjunction will be represented by the character `'|'`, its priority will be 1
- the input string will contain only valid characters (identifiers or operators), no space whatsoever
- the input string will contain only a syntactically valid expression

We will do this in two steps. The first step will be to build the polish notation of our expression using a character by character analysis. Remember to use the operators priority (3 is the maximum priority). It will build us a `bundle` of characters. The second step will be to use this `bundle` to build our `boolean` expression.

We will use two `bundle`, one for the operators, the other for the result (output). Both will be `Empty` at the beginning. The following explanation is one way to do the parsing :[1]

1. While there are characters to be read :
   (a) If the character is an identifier, then add it to the output `bundle`
   (b) If the character is an operator
      i. While there is an operator on the top of our operators `bundle` with a priority greater than our current operator :
         A. Add the operator from the operator `bundle` to our ouput `bundle`
         B. Remove the operator from the top of the operators `bundle`
      ii. Add the current operator to the top of our operators `bundle`

2. Where there are no more characters to read :
   While there are still operators in our operators `bundle` :
   (a) Add the operator from the operators `bundle` to our ouput `bundle`
   (b) Remove the operator from the top of the operators `bundle`

3. Return the output `bundle`

[1] based on `http://en.wikipedia.org/wiki/Shunting-yard_algorithm`

2.1 Parse

Write `parse` which parses a string and generates the corresponding characters `bundle`.

```
val parse : string -> char bundle = <fun>

# parse "a&b";;
- : char bundle = Item ('&', Item ('b', Item ('a', Empty)))
```

2.2 Builder

Write `builder` which builds the `boolean` expression from our PN `bundle`.

```
val builder : char bundle -> boolean = <fun>

# builder (Item ('|', Item ('|', Item ('c', Item ('b', Item ('a', Empty))))));;
- : boolean = Or (Or (Var "c", Var "b"), Var "a")
```

2.3 Truth table

Write `truth_table` which displays the corresponding truth table from the given input string.

```
val truth_table : string -> unit = <fun>
```

Stage 3 - Bonus

Here comes the fun. This is a non-exhaustive and unordered list of things you could add :

- Handle spaces and layout characters
- Handle parenthesis
- Identifiers with name longer than one character
- Handle operators with two characters `&&` and `||`
- Handle static value in the expression `"a|True"`
- Allow input in polish notation and reverse polish notation directly `"a b |"`
- Handle more operators : ⊕, ...[2]
- Handle input errors like syntax or unknown operators
- Manage the simplification of the expression, with a display of the correct one.
- ...

[2] `http://en.wikipedia.org/wiki/List_of_logic_symbols`