

Module Ltl

1. This is the main data structure representing an LTL formula. Notice that `Var` is of type `string` now.

```
type ltl_formula =
  | True
  | False
  | Var of string
  | Or of ltl_formula × ltl_formula
  | And of ltl_formula × ltl_formula
  | Neg of ltl_formula
  | Iff of ltl_formula × ltl_formula
  | Imp of ltl_formula × ltl_formula
  | Until of ltl_formula × ltl_formula
  | Next of ltl_formula
  | Glob of ltl_formula
  | Ev of ltl_formula
```

2. This function prints a formula `f` on the standard output.

```
let rec show_formula f =
  match f with
  | Var x → Printf.printf "Var_\"%s\" \" \" x
  | True → Printf.printf "True"
  | False → Printf.printf "False"
  | Glob x → Printf.printf "Glob_(\"; show_formula x; Printf.printf ")"
  | Ev x → Printf.printf "Ev_(\"; show_formula x; Printf.printf ")"
  | Neg x → Printf.printf "Neg_(\"; show_formula x; Printf.printf ")"
  | Next x → Printf.printf "Next_(\"; show_formula x; Printf.printf ")"
  | And (x, y) → Printf.printf "And_(\"; show_formula x; Printf.printf ",_\";
    show_formula y; Printf.printf ")"
  | Or (x, y) → Printf.printf "Or_(\"; show_formula x; Printf.printf ",_\";
    show_formula y; Printf.printf ")"
  | Until (x, y) → Printf.printf "Until_(\"; show_formula x; Printf.printf ",_\";
    show_formula y; Printf.printf ")"
  | Iff (x, y) → Printf.printf "Iff_(\"; show_formula x; Printf.printf ",_\";
    show_formula y; Printf.printf ")"
  | Imp (x, y) → Printf.printf "Imp_(\"; show_formula x; Printf.printf ",_\";
    show_formula y; Printf.printf ")"
```

3. This function takes a formula `f` and simplifies it according to the laws of Boolean algebra.

```

let rec simp f =
  match f with
  | Neg True  → False
  | Neg False → True
  | Neg (Neg e) → simp e
  | Neg e → Neg (simp e)
  | And (_, False) → False
  | And (False, _) → False
  | And (True, e) → simp e
  | And (e, True) → simp e
  | Or (True, _) → True
  | Or (False, e) → simp e
  | Or (_, True) → True
  | Or (e, False) → simp e
  | Or (a, b) → Or (simp a, simp b)
  | And (a, b) → And (simp a, simp b)
  | Imp (a, b) → simp (Or (Neg a, b))
  | Iff (a, b) → simp (And (simp (Imp (a, b)), simp (Imp (b, a))))
  | _ → f

```

4. Returns the closure of LTL formula f in the form of a list, e.g., $\text{closure } (\text{Glob } (\text{Var } \text{"a"}))$ would return $[\text{Var } \text{"a"}; \text{Glob } (\text{Var } \text{"a"})]$.

```

let rec closure f =
  match f with
  | And (a, b) → f :: (closure a)@(closure b)
  | Or (a, b) → f :: (closure a)@(closure b)
  | Var a → [Var a]
  | Neg a → f :: (closure a)
  | Imp (a, b) → closure (simp (Imp (a, b)))
  | Iff (a, b) → closure (simp (Iff (a, b)))
  | Next a → f :: (closure a)
  | Until (a, b) → f :: (closure a)@(closure b)
  | Glob a → f :: (closure a)
  | Ev a → f :: (closure a)
  | _ → [f; Neg f]

```

5. Returns a list of used variables in a formula f , e.g., if f is $\text{Glob } (\text{Var } \text{"a"})$, then $[\text{Var } \text{"a"}]$ is returned.

```

let rec variables f =
  match f with
  | And (a, b) → (variables a) @ (variables b)
  | Until (a, b) → (variables a) @ (variables b)
  | Or (a, b) → (variables a) @ (variables b)
  | Glob a → variables a
  | Neg a → variables a
  | Ev a → variables a
  | Next a → variables a
  | Var x → [Var x]
  | _ → []

```

Module Aba

6. In a sense, this function is the “heart” of the program, although one of the most easy to write bits of code, actually. *target* takes some state *f*, an LTL formula, and some list of variables, *s*. We have $s \subseteq \Sigma$, where $\Sigma = 2^{A^P}$ is an alphabet consisting of the powerset of a set of variables.

target basically computes the successor state for *f* upon processing the input *s*.¹

```

let rec target f s =
  match f with
  | Var a →
    if (List.exists ((=) (Var a)) s) then True else False
  | And (a, b) → Ltl.simp (And (target a s, target b s))
  | Or (a, b) → Ltl.simp (Or (target a s, target b s))
  | Until (a, b) →
    Ltl.simp (Or (target b s, (Ltl.simp (And (target a s, Until (a, b))))))
  | Imp (a, b) → target (Ltl.simp (Imp (a, b))) s
  | Iff (a, b) → target (Ltl.simp (Iff (a, b))) s
  | Next a → a
  | Glob a → Ltl.simp (And (target a s, Glob a))

```

¹Notice how close this function is to its corresponding mathematical definition of δ , the state transition function for alternating Buchi automata:

$$\begin{aligned}
 \delta(true, a) &= true \\
 \delta(\varphi \vee \psi, a) &= \delta(\varphi, a) \vee \delta(\psi, a) \\
 \delta(\neg\varphi, a) &= \neg\delta(\varphi, a) \\
 \delta(X\varphi, a) &= \varphi \\
 \delta(\varphi U \psi) &= \delta(\psi, a) \vee \delta(\varphi, a) \wedge \varphi U \psi \dots \text{and so on.}
 \end{aligned}$$

```

| Ev a → Ltl.simp (Or (target a s, Ev a))
| Neg a → Ltl.simp (Neg (target a s))
| - → f

```

7. *transitions* returns all the possible transitions within an alternating Buchi automaton for a set of states *states*, where each element in the list is an LTL formula. Usually *states* would contain the closure of some LTL specification. *alphabet* is the input alphabet of the alternating Buchi automaton.

```

let rec transitions states alphabet =
  match states with
  | [] → []
  | s :: st →
    let succ_states = (List.map (fun p → s, p, (target s p)) alphabet) in
    succ_states @ (transitions st alphabet)

```

8. This function takes as input a list of formulas and prints it to standard output. Notice that the argument to *show_formula_list* will usually just contain elements of the form *Var string*, since it is used to print an automaton's input symbols which in turn, are elements of an alphabet.

```

let rec show_formula_list =
  function
  | [] → Printf.printf ""
  | x :: y :: xs → Ltl.show_formula x;
    printf ";␣"; Ltl.show_formula y;
    show_formula_list xs
  | x :: xs → Ltl.show_formula x;
    show_formula_list xs

```

9. *show_transitions* takes a list of transitions, where each list element is a triple (*ltl_formula*, [*ltl_formula*], and prints it to standard output.

```

let rec show_transitions =
  function
  | [] → Printf.printf ""
  | (s, p, t) :: tt →
    Printf.printf "␣(";
    show_formula s;
    Printf.printf "),␣["; show_formula_list p; printf "],␣(";
    show_formula t;
    Printf.printf ")]\n";
    show_transitions tt

```

10. *succ_states* takes a list of transitions (of an alternating Buchi automaton) and some state $s \in LTL$, and returns all immediate successor states of s , i.e., all states reachable via taking one transition only.

```
let succ_states t s =
  let rt = List.filter (fun h → match h with (src, _, _) →
    if s = src
    then true
    else false) t in
  List.map (fun h → match h with (_, _, dst) → dst) rt
```

11. *unfold_and_or* takes a list of LTL formulas whose elements may be conjunctions or disjunctions of formulas. It returns the list of formulas, but all conjunctions and disjunctions are “unfolded” in a sense that an entry $a \vee b$ becomes $[a; b]$, respectively for \wedge .

```
let rec unfold_and_or =
  function
  | [] → []
  | (And (a, b)) :: ft → a :: b :: unfold_and_or ft
  | (Or (a, b)) :: ft → a :: b :: unfold_and_or ft
  | f :: ft → f :: unfold_and_or ft
```

12. Function returns *true* if state $q \in LTL$ is reachable in a list of transitions t with initial state $i \in LTL$, otherwise *false*.

```
let rec has_path t i q =
  if i = q then true
  else
    (* first, get all immediate successors of i: *)
    let r = succ_states t i in
    (* we have to unfold all And and Or states: *)
    let r = unfold_and_or r in
    (* we have to remove cycles, or the algorithm does not terminate: *)
    let r = List.filter ((≠) i) r in
    match r with
    | [] → false
    | _ →
      if List.mem q r
      then true
      else List.for_all (fun r → has_path t r q) r
```

13. This function “prunes away” unreachable states and their respective transitions, where t is a list of transitions of an alternating Buchi automaton, $i \in LTL$ is the initial state of the

automaton, and s a list of states of the automaton, where each $s_i \in s$ is an LTL formula.²

```
let prune_transitions  $t\ i =$ 
  (* first, compute a set of all reachable states: *)
  let  $rs = List.filter\ (\text{fun } s \rightarrow has\_path\ t\ i\ s)\ (closure\ i)\ \text{in}$ 
  (* then remove all transitions which do not cover any of those states: *)
  List.filter\ (\text{fun } t \rightarrow \text{match } t \text{ with } (s, -, -) \rightarrow
    if\ (List.mem\ s\ rs)
    then true
    else false)\ t
```

Module *Ltl_parser*

14. Bit of an ugly definition to make a lexer consisting of the keywords in the given list. *lexer* is basically a function which takes an input stream and produces an output token stream.

```
let lexer =
  Genlex.make_lexer
  ["&"; "|"; "G"; "U"; "X"; "F"; "->"; "<->"; "True"; "False"; "("; ")"; "-"]
```

15. Definition of a recursive-descent stream parser. *parse_formula* is a function which takes a token stream and returns an LTL formula. It is mutually recursive, giving rise to precedence of operators.

```
let rec parse_atom = parser
  | [ $'Kwd\ "-"; e1 = parse\_atom$ ]  $\rightarrow Neg\ (e1)$ 
  | [ $'Ident\ c$ ]  $\rightarrow Var\ c$ 
  | [ $'Kwd\ "("; e = parse\_formula; 'Kwd\ ")"$ ]  $\rightarrow e$ 
and parse_formula = parser
  | [ $e1 = parse\_and; stream$ ]  $\rightarrow$ 
    (parser
      | [ $'Kwd\ "|"; e2 = parse\_formula$ ]  $\rightarrow Or\ (e1, e2)$ 
      | [ $'Kwd\ "U"; e2 = parse\_formula$ ]  $\rightarrow Until\ (e1, e2)$ 
      | [ $'Kwd\ "->"; e2 = parse\_formula$ ]  $\rightarrow Imp\ (e1, e2)$ 
      | [ $'Kwd\ "<->"; e2 = parse\_formula$ ]  $\rightarrow Iff\ (e1, e2)$ 
      | [ $\rangle$ ]  $\rightarrow e1)\ stream$ 
```

²This shows again how sets are realised in terms of lists here.

```

and parse_and = parser
| [⟨ e1 = parse_atom; stream ⟩] →
  (parser
   | [⟨ 'Kwd "&"; e2 = parse_and ⟩] → And (e1, e2)
   | [⟨ ⟩] → e1) stream
| [⟨ e1 = parse_unary ⟩] → e1

and parse_unary = parser
| [⟨ 'Kwd "G"; e1 = parse_and ⟩] → Glob (e1)
| [⟨ 'Kwd "F"; e1 = parse_and ⟩] → Ev (e1)
| [⟨ 'Kwd "X"; e1 = parse_and ⟩] → Next (e1)

```

Module Ltl2aba

16. The program's main function. It accepts from the standard input a string which contains an LTL formula. Then it converts this string into an *ltl_formula* data type, and prints the transitions of an alternating Buchi automaton corresponding to the formula to the standard output.

```

let _ =
  let formula =
    Ltl_parser.parse_formula(lexer(Stream.of_string Sys.argv.(1))) in
  let alphabet = Set_list.powerset (Ltl.variables formula) in
  let all_transitions =
    transitions (closure formula) alphabet in
  let min_transitions = prune_transitions all_transitions formula in
  Aba.show_transitions min_transitions

```

Index

Aba (module), **6**, 16, 16
And, **1**, 3, 6
closure, **4**, 4, 13, 16
Ev, **1**, 6
False, **1**, 3, 6
Glob, **1**, 6
has_path, **12**, 12, 13
Iff, **1**, 4, 6
Imp, **1**, 3, 4, 6
Ltl (module), **1**, 6–8, 16, 16
Ltl2aba (module), **16**
ltl_formula (type), **1**, 1
Ltl_parser (module), **14**, 16, 16
Neg, **1**, 3, 4, 6
Next, **1**
Or, **1**, 3, 6
prune_transitions, **13**, 16
show_formula, **2**, 2, 8, 9
show_formula_list, **8**, 8, 9
show_transitions, **9**, 9, 16
simp, **3**, 3, 4, 6
succ_states, **10**, 12
target, **6**, 6, 7
transitions, **7**, 7, 16
True, **1**, 3, 6
unfold_and_or, **11**, 11, 12
Until, **1**, 6
Var, **1**, 4–6
variables, **5**, 5, 16