ソフトウェア技法 ミニプロジェクト (後半)

亀山幸義 (筑波大学情報科学類)

[2017/07/31 改訂版]

[2017/08/02 改訂版; NNF の記述がまるごと落ちていたので追加]

[2017/08/02 16:36 改訂版; 誤植修正]

プログラミングの授業は、言語の機能を個別に見ていくボトムアップ方式だけでは面白くない。何か目標となる，まとまったプログラムを書いてみたい。そうすることにより、断片的な知識が整理されて「使える」ものとなっていく。

この授業では、例年、授業の最終週に「ミニプロジェクト」と称して、まとまったプログラムを書いてきたが、今年は趣向を変えて、授業の途中と最後に，ミニプロジェクトを 1 つずつ (合計 2 つ) やることにした。後半の課題は、論理式に対する処理である。

1 命題論理式の真理値

命題論理の論理式は、原子論理式 p,q,... を、「かつ (and)」、「または (or)」、「ならば (imp)」、「でない (not)」などの論理記号で結合したものである。ここでは、簡単のため、3 つ以上の論理式の「かつ」や「または」を取ることは考えない。つまり、「A かつ B」のように、論理記号の引数は 2 以下であるものとする．(「かつ」、「または」、「ならば」は 2 引数、「でない」は 1 引数である。) 3 つ以上の論理式の「かつ」を取りたいときは、「(A かつ B) かつ C」のように入れ子にする。

このような論理式のデータ型は、OCaml の代数データ型で簡単にあらわすことができる。ここでは formula という名前にする。

```
type formula =
  | Atom of string             (* 例 Atom("p"), Atom("q123")  *)
  | Not  of formula            (* 例 Not(f1)                  *)
  | And  of formula * formula  (* 例 And(f1, f2)              *)
  | Or   of formula * formula  (* 例 Or(f1, f2)               *)
;;
```

ここで、`Atom("p")` というのは、原子論理式 p のことである。また、Not, And, Or は論理記号のことである。必要ならば、上記の代数データ型に Imp (ならば) など他の論理記号を加えてもよい。

たとえば、以下のものは、formula 型の要素である。

```
let f0 = Atom("p") ;;
let f1 = And(Not(Atom("p")),Or(Atom("q"),Atom("p"))) ;;
let f2 = Not(And(Not(Atom("p")),Or(Atom("q"),Atom("p")))) ;;
```

なお、この形式は、内部処理には適しているが、人間が入力するのはとても大変である。人間は、$\neg p \vee (q \wedge p)$ といった表記を好むが、こういった自由な表記を許すためには、構文解析器が必要になるので、この授業では、そこまではやらない。つまり、皆さんの「論理式処理プログラム」に対する入力は、上記の f0, f1, f2 といった formula 型のデータとする。

構文解析に興味がある人は，自動生成するためのツールとして，ocamllex, ocamlyacc などがあるし，自分で書いてみるための手がかりとして，パーサコンビネータなどがあり，それぞれわかりやすい解説がインターネット上にいろいろあるので，調べてみるとよい．ocamllex, ocamlyacc の使い方は 3 年生の主専攻実験 S8 でも簡単に解説している．

1.1 論理式に対する簡単な処理

与えられた論理式に含まれる原子命題の名前 (文字列"p"など) を集めたリストを返す関数 `get_atom` を定義しよう。たとえば、

```
get_atom (And(Not(Atom("p")),Or(Atom("q"),Atom("p")))) = ["p"; "q"]
```

といった答えが返ってくるとよい。

ここで大事なのは、原子命題 p は 2 回含まれているが、出力のリストには 1 回だけ出てくることである。つまり、出力のリストは集合であって欲しい (要素の重複がないリストであってほしい)。集合であればよいので、要素の順番はどうでもよい。なお、2 つの文字列が等しいかどうかの判定は、=を使えばよい。

演習課題 1-1 このような関数 get_atom を定義せよ。

例. `get_atom (And(Not(Atom("p")),Or(Atom("q"),Atom("p")))) = ["p"; "q"]`

例. `get_atom (And(Atom("r"),And(Atom("q"),Not(Atom("p"))))) = ["p"; "q"; "r"]`

これらの例で、リストの中の文字列の順番は上記の通りでなくて良い. たとえば, 1 番目の例では, `["q"; "p"]` を返しても良い。ただし、同じ要素が重複することはないようにせよ。

ヒント: 要素とリストを与えられたとき、その要素がリストに出現するかどうかを調べるには、List.mem という関数を使うとよい。たとえば、`List.mem "p" ["q"; "p"; "r"]` = true であり, `List.mem "s" ["q"; "p"; "r"]` = false である.

```
(* 補助関数 union
   2つのリストをもらい、重複なく合併したリストを返す。要素の順番は気にしない。*)
let rec union lst1 lst2 =
  match lst1 with
  | [] → lst2
  | h::t → ...  (* ここで List.mem を使う *)
;;
(* 関数 get_atom
   formula 型の引数に対して、それに含まれる原子命題のリスト（同じ要素が
   2回以上あらわれないもの）を返す *)
let rec get_atom (form1 : formula) : string list =
  match form1 with
  ... (* 上記の union を使う *)
;;
```

1.2 論理式に対する割当て

次に、原子命題に対する真理値の割当て (assignment) を考える。(以下では、単に「割当て」と呼ぶ。) これは、1 つ 1 つの原子命題に対して true か false を定めたものであり、ここでは, 原子命題の名前 (文字列) と、true または false を 2 つ組にしたもののリストとする。

割当ての例 1: `[("p", true); ("q", false)]`

割当ての例 2: `[("p", true); ("q", true); ("r", false)]`

このような形にしておくと嬉しい理由は、「割当て a のもとでの原子命題 p の真理値を計算したい」という時、List.assoc という関数を使うだけで答えが得られるからである。

```
let _ = List.assoc "p" [("p", true); ("q", false)] ;;
  - : bool = true

let _ = List.assoc "q" [("p", true); ("q", false)] ;;
  - : bool = false

let _ = List.assoc "r" [("p", true); ("q", false)] ;;
  Exception: Not_found. （エラー）
```

一般に、[(キー 1, 値 1); (キー 2, 値 2); ...; (キー n, 値 n)] という形のリストを**連想リスト (association list)** と呼び、よく使うデータ構造[*1]である。連想というのは、「キー」と「値」の「対応表」という程度の意味である。List.assoc (OCaml の List モジュールの assoc 関数) は、キーと連想リストをもらうと，キーに対応する値を返すものであり，キーに対応する値がない場合はエラーを返す．(上記の例を参照)

ここでは、キーとして，原子命題の名前 (文字列)、対応する値として bool 型の値 (真理値) を用いた連想リストを用いる。

1.3 論理式の真理値

論理式の真理値を計算しよう。この関数も一種の評価器 (evaluator) であるので、eval という名前にしよう。

論理式が与えられたとき、その論理式に含まれる全ての原子命題に対して true/false という値が決まれば、あとは、論理記号の意味に応じて決まっていき、最終的に論理式全体の真理値 (true/false) が決まる。つまり、eval は、論理式と割当てを引数として、真理値を返す関数である。

例 1. `eval (And(Not(Atom("p")),Or(Atom("q"),Atom("p")))) [("p",true);("q",false)]` = `false`

例 2. `eval (And(Not(Atom("p")),Or(Atom("q"),Atom("p")))) [("p",false);("q",true)]` = `true`

演習課題 1-2 このような性質をもつ関数 `eval` を定義せよ。`eval` は `formula -> (string * bool) list -> bool` という型を持つはずである．(ここで第 2 引数は，割当ての型を表す．)

なお、与えられた割当てにおける原子命題が不足しているときは、エラーを出すようにせよ。逆に、原子命題が多過ぎるのは，問題ない。

例 3. `eval (And(Not(Atom("p")),Or(Atom("q"),Atom("p")))) [("p", true)]` (エラー)

例 4. `eval (And(Not(Atom("p")),Or(Atom("q"),Atom("p")))) [("p", false); ("q", true); ("r", true)]`=`true`

```
let rec eval (form1 : formula) (asgn : (string * bool) list) : bool =
  match form1 with
  | Atom(s)          → List.assoc s asgn  (* List.assoc を使えばよい *)
  | Not(form2)       → ...
  ...
```

演習課題 1-3 上記の関数 `eval` を利用して、与えられた論理式の真理値表を作成せよ。

これは、たとえば、`And(Not(Atom("p")),Or(Atom("q"),Atom("p")))` という論理式が与えられると、以下の表を印刷するというものである。

```
p q target
t t f
f t t
```

[*1] 「よく使う」というのは、連想リストの構造であって、ここでの実装方法ではない。ここでやっているように、連想リストを、単純なリストで表現すると検索効率が非常に悪くなるので、大きな連想リストに対する実際の実装は、ハッシュにもとづくものを使う。

```
t f f
f f f
```

この表の 1 行目は、見出し行であり、target formula (真理値計算の対象となる論理式) の原子命題たちを空白でくぎって印刷し、最後に target と表示している。(対象となる論理式自体をきれいに表示するのは、発展課題であるので、ここでは単に ”target”と書けばよい。) また、原子命題は 1 文字であるとは限らないが、ここでは、2 文字以上の原子命題が来ることは考えなくてよい。つまり、命題名が長いものを原子命題として持つ式の場合、以下のようにずれてしまってもよいものとする。

```
ppp qqqqq target
t t f
f t t
t f f
f f f
```

上記の表の 2 行目以降が真理値の計算結果であり、2 行目は、(p,q)=(true,true) の時、与えられた論理式の真理値が false であること、また 3 行目は、(p,q)=(false,true) の時、与えられた論理式の真理値が true であることを表している。

ヒント: 原子命題のリスト atoms をもらって、その原子命題リストに対する「すべての割当てを並べたリスト」を返す関数は、以下のように実装できる。

```
(* make_asgn_list
   原子命題のリストをもらい、それに対する「割当て」を全て生成して、
   それらを集めたリストを返す。walk というのはそれの補助関数である。
   *)
let rec walk atoms asgn =
  match atoms with
    | [] → [asgn]
    | h::t → (walk t ((h,true)::asgn)) @ (walk t ((h,false)::asgn))
;;

(* make_asign_list 本体 *)
let make_asgn_list (atoms : string list) : ((string * bool) list) list =
  walk atoms []
;;

(* make_asgn_list の使用例 *)
let _ = make_asgn_list ["p"; "q"];;
- : (string * bool) list list =
[[("q", true); ("p", true)]; [("q", false); ("p", true)];
 [("q", true); ("p", false)]; [("q", false); ("p", false)]]
(* この結果は、(p,q)=(true,true), (true,false), (false, true), (false,false)
   の４通りの割当てをリストにしたものを表している*)
```

この make_asgn_list はそのまま使ってよい。これと、1 つ前の課題の答えを使えば、「論理式 f があたえらえたとき、あらゆる割当てのもとでの f の真理値」を計算することが可能となる。結果の印刷には、print_string や print_newline を使えばよい。

```
let test s =
   print_string "Start printing ...."; print_newline ();
   print_string s; print_newline ();
   print_string "End printing ...."; print_newline ();
   ;;

let _ = test (string_of_int 135) ;;
```

ここでの注意点: OCaml では、print_string なども関数であるので、値を返す点であり、これらの印刷関数は、すべてユニット型の値 () を返すという点である。2 つ以上の印刷関数をつなげたいときは、;(セミコロン 1 つだけ) でつなげばよい。また、複雑な式の中で、印刷関数をたくさんよぶときは、かっこで囲うか、あるいは、begin .... end で囲えばよい。

なお、上記の test 関数では、print_string と print_newline のみを使ったが、ほかにも print_endline など便利な関数があるので、興味がある人、すっきりしたプログラムを書きたい人は、自分で調べるとよい。

演習課題 1-4

与えられた論理式 (formula 型のデータ) が、恒真であるかどうかを判定するプログラム is_valid を書きなさい。ただし恒真であるとは、どのような割当てのもとでも真である論理式のことであり，演習問題 1-2 の結果と、演習問題 1-3 における make_asgn_list を利用するとよい。(演習問題 1-3 における真理値表を作成する関数そのものは利用する必要はない。)

is_valid の型は formula -> bool である.

例 1. is_valid (Or(Atom("p"),Not(Atom("p")))) = true

例 2. is_valid (Or(Atom("p"),Not(Atom("q")))) = false

2 論理式の変形

命題論理式を標準形に変形する手続き、つまり、命題論理式をもらって、ある特別な形の式を出力する関数を書くことを目標にする。

ここで考える標準形は 2 つで、否定標準形 (Negation Normal Form, NNF) と論理積標準形 (Conjunctive Normal Form, CNF) である。

2.1 NNF への変形

NNF は、否定の記号 ¬ が出現する場所が、原子命題の直前のみに制限された形の命題論理式である。たとえば、$\neg(A \wedge (B \vee C))$ は NNF ではないが、それと同値な $(\neg A) \vee ((\neg B) \wedge (\neg C))$ という論理式は NNF である。なお、これと同値な、$((\neg B) \wedge (\neg C)) \vee (\neg A)$ という論理式も NNF であり、こちらを出力してもよいものとする。

NNF への変形は、以下の同値変形規則を用いる。

$$\neg(\neg A) \text{ は } A \text{ と同値である。}$$
$$\neg(A \wedge B) \text{ は } (\neg A) \vee (\neg B) \text{ と同値である。}$$
$$\neg(A \vee B) \text{ は } (\neg A) \wedge (\neg B) \text{ と同値である。}$$

左辺を右辺に書きかえていくことにより、¬ が次第に内側にはいっていき、最後には NNF が得られる。

演習課題 2-1 与えられた論理式を、それと同値な否定積標準形 (NNF) に変形する手続き to_nnf を実装せよ。ただし、この関数の型は formula -> formula とする。

ヒント: いろいろな実装方法があるが、ここでは、単純に (効率のことは気にせず) 以下のような形でいくのが一番簡単であろう。ここでは、論理記号を「上の方から 2 つ分見る」という方式で、すべての場合を書きつくしている。

```
let rec to_nnf (form1 : formula) : formula =
  match form1 with
  | Atom(_)                → form1   (* すでに NNF であるので変形不要 *)
  | Not(Atom(_))           → form1    (* すでに NNF であるので変形不要 *)
  | Not(Not(form2))        → to_nnf form2
  | Not(And(form2,form3)) → Or (to_nnf (Not(form2)), to_nnf (Not(form3)))
  | Not(Or(form2,form3))   → ...
  | And(form2,form3)       → And(to_nnf form2, to_nnf form3)
                                (*上記のパターン以外でトップが And のとき *)
  | Or(form2,form3)        → ... (*上記のパターン以外でトップが Or のとき *)
;;
```

ここで大事な点は、Not(Not(form2)) のケースで、form2 を答えとするのではなく、それをさらに関数にかけて、to_nnf form2 を計算して、その結果を返している点である。これは form2 の中に、さらに ¬ が変な位置にあって、NNF になっていない可能性があるので、変形を続けないといけないからである。

同様の考えで、Not(And(form2,form3)) のケースも書いているが、右辺が、Or(to_nnf (Not(form2)), to_nnf (Not(form3))) となっているのは非常に大事な点である。これを、Or(Not(to_nnf(form2)), Not(to_nnf(form3))) とするとうまく行かない。(なぜうまくいかないか、試してみてほしい。)

2.2 CNF への変形

CNF は、

$$(L_1^1 \vee L_2^1 \vee \cdots) \wedge (L_1^2 \vee L_2^2 \vee \cdots) \wedge \cdots$$

という形をした命題論理式である。ただし、各 L_i^j は、原子論理式 (p, q など) か、原子論理式の直前に否定をつけた形 ($\neg p$, $\neg q$ など) のいずれかとする。つまり、And (かつ) は一番外側に現れ、次の層に Or (または) が現れ、一番内側に Not (〜でない) が現れるというように、論理記号が所定の位置にのみ現れるよう、制限された形である。

この CNF への変形をしたい理由は、様々な判定が楽になる (一般の命題論理式に対する判定関数を書くのは大変でも、CNF になっていれば簡単に書ける) ということが多いからである。今回の演習では、「様々な判定」の例として「恒真であるかどうか」を判定することにする。ただし、論理式が恒真である (いつでも真である) とは、その論理式がどんな割当てのもとでも真になることである。

まず、CNF のデータ型を決めよう。CNF は 命題論理式の一種なので、formula 型をそのまま流用してもよいが、以下の 2 つの理由により、新しく cnf 型を定義することにする。

- formula 型では Or や And は 2 引数のものしかないが、CNF ではたくさんの論理式が Or や And で一気につながる形がでてくるので、それを Or(Atom("p"),Or(Atom("q"),Or(Not(Atom("p")),Not(Atom("q"))))) のように書くのではなく、効率的に表したい。
- formula 型のものがすべて CNF とは限らないので、「CNF だけをぴったり表しているデータ型」があった方が、チェックしやすい。(CNF になっていないものを出力してしまう、という間違いを防げる)。

CNF 型のデータを定義するためにはまず、上記の L_i^j に相当するデータ (論理学では、この形のデータを「リテラル」と呼ぶ) の型を定義しよう。

```
type literal =
  | Pos of string     (* 例 Atom("p") に相当するリテラルを Pos("p") と書く  *)
  | Neg of string     (* 例 Not(Atom("p"))に相当するリテラルを Neg("p") と書く*)
;;
```

さて、CNF は、「「0 個以上のリテラルを Or でつないだもの」を 0 個以上 And でつないだもの」である。ここでは、Or とか And とかは O とか A 1 文字にした以下のデータ (単なるリストに O とか A を付加したもの) で表してしまおう。

```
type clause =
   | O of literal list ;;
type cnf =
   | A of clause list ;;
```

これにより formula 型の `And(Or(X,Or(Y,Z)),Or(V,W))` という式は、cnf 型では、`A([O([X; Y; Z]); O([V; W])])` というデータで表されることになる。(なお、もうちょっと節約して `[[X; Y; Z]; [V; W]]` というデータで表しても良かったのであるが、そうすると、単なるリストのリストと混同する間違いが起きかねないので、ここでは A とか O というタグを残した。)

演習課題 2-2 与えられた NNF を、それと同値な論理積標準形 (Conjunctive Normal Form, CNF) に変形する関数 `to_cnf` を定義しなさい。

ここの関数への入力は、すでに NNF に変形されている (¬ は一番内側にしかない) と仮定してよい。

```
to_cnf : formula -> cnf
to_cnf (Or(Atom("p"),Not(Atom("p"))))
  ==> A([O([Pos("p");Neg("p")])])
to_cnf (And(Atom("p"),Or(Atom("q"),Atom("r"))))
  ==> A([O([Pos("p")]); O([Pos("q"); Pos("r")])])
to_cnf (Or(Atom("p"),And(Atom("q"),Atom("r"))))
  ==> A([O([Pos("p"); Pos("q")]); O([Pos("p"); Pos("r")])])
```

ヒント: この関数 `to_cnf` の概形は、以下の通りである。

```
let rec to_cnf (f : formula) : cnf =
match f with
| Atom(p) → A[O[Pos(p)]]
| Not(Atom(p)) → A[O[Neg(p)]]
| And(f1,f2) →
     let cnf1 = to_cnf f1 in
     let cnf2 = to_cnf f2 in
      (* cnf1 と cnf2 を And でつなげた CNF を作りたい *)
     begin
       match (cnf1, cnf2) with
       | (A(cl1), A(cl2)) →  A(cl1 @ cl2)
       (* ここで @ は2つのリストを結合したリストを返す OCaml 演算子
          (append) を使っている;
          e1 @ e2 は List.append e1 e2 と同じである。*)
     end
| Or(f1,f2) →
     let cnf1 = to_cnf f1 in
     let cnf2 = to_cnf f2 in
      ... (* ここを実装するのが、課題である *)
      (* cnf1 と cnf2 を Or でつなげた CNF を作りたい *)
| _ → failwith "This case cannot happen"  (* 入力が NNF でなかった場合、エラーにする *)
```

cnf1 と cnf2 が CNF のとき、これらを Or をつなげると、CNF にならない。(Or が一番外の記号になってしまうため。) そこで、論理式に関する「分配則」を使う。分配束は以下の通りである。

- $(A \wedge B)$ と C を Or でつなげた論理式は $(A \vee C) \wedge (B \vee C)$ と同値である。
- $(A \wedge B)$ と $(C \wedge D)$ を Or でつなげた論理式は $(A \vee C) \wedge (B \vee C) \wedge (A \vee D) \wedge (B \vee D)$ と同値である。
- 一般に、$(A_1 \wedge A_2 \wedge \cdots \wedge A_m)$ と $(B_1 \wedge B_2 \wedge \cdots \wedge B_n)$ を Or でつなげた論理式は、$\bigwedge_{1 \leq i \leq m, 1 \leq j \leq n} (A_i \vee B_j)$ と同値である。

これらを適宜使って、cnf1 と cnf2 の論理和の形の論理式を CNF に変形するアルゴリズムを実装してほしい。

演習課題 2-3 (発展課題だが、これが目標なのでできるだけ解いてほしい) CNF の恒真性の判定をする関数 `is_valid_cnf` を定義しなさい。

```
is_valid_cnf (A([O([Pos("p")]); O([Pos("q"); Pos("r")])]))
  ==> false
is_valid_cnf (A([O([Pos("p");Neg("p")]); O([Neg("r"); Pos("r")])]))
  ==> true
```

ヒント: 恒真性の判定条件は以下の通り。

1. `A([C1; C2; ...; Cn])` という形の CNF が恒真であるのは、`C1`, `C2`,..., `Cn` という形のもの (節とよばれる) がすべて恒真のとき、および、そのときに限る。(上記の `Ci` は、And でつながっているものを並べたリストであることに注意せよ。)
2. `O([L1; L2; ...; Lm])` という形の節が恒真であるのは、ある i, j とある p に対して、`Li`=`Pos("p")`, `Lj`=`Neg("p")` の形であるとき、および、その時に限る。(上記の `Li` は、Or でつながっているものを並べたリストであることに注意せよ。)

この 2 つ目の条件はちょっとわかりにくいが要するに、

```
O([ ...; Pos("p"); ....; Neg("p"); ....])
```

もしくは

```
O([ ...; Neg("p"); ....; Pos("p"); ....])
```

のどちらかの形の節は恒真であり、そのパターンが 1 つも含まれない節は恒真でないということである。(ただし、上記の"p"は何でもよい。) 例は以下の通り：

```
O([Pos("p");Pos("q");Neg("p");Neg("r")])   ... 恒真
O([Pos("p");Pos("q");Pos("p");Neg("r")])   ... 恒真でない
O([Neg("p");Pos("q");Neg("p");Pos("p")])   ... 恒真
```

演習課題 2-4 (発展課題) 上記を組み合わせて、命題論理式を NNF に変形し、さらに CNF に変形して恒真性を判定しなさい。

3 発展課題

この演習課題は、いろいろな発展課題が考えられる。以下を参考に 1 つでもチャレンジしてみてほしい。

- 論理式や CNF を「きれいに」出力する: formula 型の論理式、たとえば、
  `Not(And(Not(Atom("p")),Or(Atom("q"),Atom("p"))))` をそのまま出力すると、読みにくく、間違いやすい。そこで、formula 型の論理式を`!( !p /\ (q \/ p))` といった形で印刷する関数があるとうれしい。ディバッグの際にも役に立つだろう。このようなプログラムを pretty-printer と呼ぶ。

  ```
  pretty_print (Not(And(Not(Atom("p")),Or(Atom("q"),Atom("p")))))
  ==>   "! ( !p /\ (q \/ p))"
  ```

  なお、出力において、無駄な括弧はできるだけ印刷しないようにしたい。たとえば、`(p /\ q)` というのはあきらかに無駄な括弧を含んでいる。こういった括弧はできるだけ印刷しない方がよい。(ただし、省略しすぎると間違ったものになってしまうので、できる範囲でよい。)
- 構文解析器の実装: 論理式を formula 型の要素として入力するのは大変である。`! p /\ (q \/ r)` といった形で入力できると、とてもうれしい。このような形の文字列をもらって formula 型に変形するプログラムを構文解析器と言う。様々な構文解析の作り方があるが、(1) この範囲であれば、自分で素朴に作ってみても非常に面白いだろう。(2) また、ocamllex, ocamlyacc というツールを使うやりかたもある。(3) さらに、Parser Combinator (parsec) など面白い構文解析の書き方もある。
- DNF (論理和標準形) への変形:
  DNF は CNF の「双対 (dual)」であり、一番内側にリテラルがあり、次に And の階層があり、一番外に Or の階層があるという論理式である。
  上記の CNF 変形アルゴリズムは、ほんのわずかな変形で、DNF 変形アルゴリズムになる。また、恒真性判定アルゴリズムは、ほんのわずかな変形で、充足可能性判定アルゴリズムになる。それを実装しなさい。
  なお、充足可能性の方は、単に「充足可能」という答えだけでなく、「どの割当てのもとで真となるか」という割当ても返してほしい。(たとえば、数独ゲームを命題論理式で表した場合、「解が存在する」というだけでなく「実際にどういう解か」が知りたい。) ただし、充足可能な割当てが複数ある場合は、1 つだけ出力すればよい。
- 「小さな CNF」への変形:
  この資料で述べた「CNF への変換手続き」は、得られる CNF の大きさが、最悪の場合、もとの命題論理式の大きさの指数関数の大きさになるという欠点があった。もし、命題論理式の充足可能性にのみ興味がある場合、より小さな CNF へ変換手続きが知られている。この中で古典的なものは、1968 年に発表された Tseitin algorithm である。このアルゴリズムをインターネット等で調べて OCaml で実装しなさい。また、その性能 (変形にかかる時間、また、得られる CNF の大きさ) を調べなさい。
  参考: 命題論理の充足可能性問題は SAT 問題 (satisfiability 問題) と呼ばれ、コンピュータサイエンスの中でも非常に重要な問題である。というのは、我々がコンピュータを使って解きたい典型的な「問題」は、「たくさんの制約があって、その制約たちを全て (論理的に) 満足する解を探す」という制約解消問題であり、驚くほどたくさんの「問題」たちが、このパターンにおとしこめるからである。このような制約が命題論理の範囲で記述できれば[*2]命題論理の論理式の恒真性判定、あるい

[*2] 命題論理の範囲で記述できない制約はどうするのか？ に対する答えの 1 つとして、SMT ソルバがあり、興味がある人は海野広志先生に聞いてみてほしい。

は、充足可能性判定により、問題が解ける。SAT を解くプログラムのことを SAT ソルバと言い、SAT ソルバの高速化に関する猛烈な国際競争のおかげで、近年では、非常に巨大な SAT 問題を現実的な時間内に解けるようになってきている。興味がある人は ”SAT solver” や ”SAT competition”を検索してみるとよい。