目 次

1 はじめに

このレポートは、CommonLisp というプログラミング言語と八杉満利子教授の計算機数学 I,II という授業を基に、特別研究としてまとめたものです。
今回は、真理値表と Wang のアルゴリズムについてのプログラムとその解説について書きました。

これらを始めた動機は、計算機数学 II の主な目的である命題論理と Wang のアルゴリズムの理解を深めることと、今後、計算機数学 I、II を勉強される方々の手助けとなるものを作りたいと思ったことです。

私たちの作ったプログラムには、要領の悪い所や改良の余地があるかもしれ知れませんが、時間が足りなかったため御了承下さい。
また、何かお気づきの点がございましたら、私たちが卒業するまでに御指摘、御指導の方よろしくお願い致します。

プログラムを断片的に動かしても正常に動かない場合があります。

2　命題論理の式

命題論理の式（論理式）とは、基本命題（命題変数と呼ばれる記号）およびそれから命題結合子と呼ばれる接続詞で合成される合成（論理）式です。

・命題論理の記号と式

- 命題結合子
  not , and , or , ⇒

  を使いました。not , and , or は Lisp の関数の中にありますが、⇒ は自分たちで定義しました。
- 基本（論理）式
  X,Y... などで表す。これらを命題変数ともいう。正式には、例えば X に添字をつけて、X_0,X_1,X_2,... で表します。
- 合成（論理）式
  A,B がすでに定義された式ならば (not A) , (A and B) , (A or B) , (A ⇒ B) も式である。
- シークエント
  シークエントを見易くするために！を使いました。
  例えば (A and B) をシークエントにするには、(! (A and B)) と入力します。

3 CommonLisp について

Lisp は 1959 年に MIT で開発された、初期の IBM マシン上で Lisp1.5 が最初の" 公共 (public)" の Lisp システムであった。その後、Lisp1.5 と十分互換性のあるものが様々な機械上で作成された。次の数年の間に、二つの主要な表現が発生し、また Lisp1.5 の修正や改良が生じた。

BNN-Lisp は PDP-1 上で最初開発され、次に SDS-940 にのせられた。また、除々に PDP-10 上の Interlisp へと成長していった。最近では VAX 上の Interlisp があり、また Xerox によってつくられた一連のパーソナルマシンがある。

一方、MIT においては、PDP-6 に対して新しい Lisp 処理系の開発が始まった。それは Maclisp として知られるようになり、今日 DEC-20 上で動いている。Maclisp の初期の版はスタンフォードへいき、Lisp1.6 となった。それは再び二つに分かれた。カリフォルニア大学アーバイン校の UCI-Lisp とユタ大学の Standard Lisp である。Maclisp の開発は MIT で独立して続けられた。そしてその後継者はかなり互換性のある方言に影響を与えた。主だったものの一つは Lisp マシン Lisp であり、これは、MIT で設計されつくられたパーソナル Lisp マシンへの、言語の大きな拡張であった。Lisp マシンに対しては二つの製造会社が生まれ、二つの副次的な方言が開発された。加えてバークレーで VAX(UNIX のもとで) 上に Franz Lisp が、MIT で VAX(VMS のもとで) 上に Maclisp のもう一つの後継者である NIL が開発された。すべての Maclisp の後継者は今は事の大小を問わず相互に互換性がなくなった。そして、ついに 1981 年に Common Lisp 委員会が、Maclisp の世界の混乱をしずめる試みとしてつくられた。すべての現在の Maclisp 後継グループが協力して同じ方向を目指すことを合意して、彼らはある Lisp を定義した。

この新しい Lisp が Common Lisp である。Common Lisp:The Language とよぶ Guy L.Steele によるマニュアルと言語仕様書がある。Commom Lisp は政府と多くの産業会の組織により人工知能システム用の言語として採用されつつある。多くの企業 (Lisp マシン企業を含めて) はいま、Common Lisp を自分のコンピュータ上で提供している。いくつかのパーソナルコンピュータ上でも存在するし、また日本では、Prolog の流行にもかかわらず、いくつかの Common Lisp 処理系も存在している。

Common Lisp には次の二つの鍵となる性質がある。

- 初期の Lisp 方言の一貫性のない記法や非論理的な記法の多くを、取り除くかあるいは合理的なものにした。

Common Lisp は Lisp のたいへんきれいな方言である。

- 言語の完全な定義と公開されたマニュアルがあり、これによりプログラムは移植可能となる。Common Lisp で書かれ、一つの機械上でデバッグされたプログラムは、変更せずに他の機械上で実行できるはずである。

4 Lispの特徴

すべてのLispオブジェクトはコンスかアトムである。アトムはもっとも基本的なオブジェクトです。またLispで扱うものはすべて、リストの形をしています。
（例）

- アトム
  3 , 5.7 , -2 , A , CAR , B12 , NIL etc

またLispの特徴に、下記の（例）でも書いていますけれども、自分自身で関数を定義できるというものがあります。そして自分自身で定義した関数をbodyに入れ込んで、再帰呼び出しをすることが出来ます。
（例）

```
(defun my-kansuu （引数）
    ⋮
 (my-kansuu ···
    ⋮
        ))
```

5　真理値

5.1　真理値の説明

文字と記号の列で表される式で問えるのは、真か偽かということである。この真と偽を真理値という。これらをそれぞれ T, F と書く。基本式である命題変数はTかFを値としている。各式は結合子の意味にしたがってTかFを値とする。

たとえば、(X and Y) についてXがT、YがFを値にとれば、and が「かつ」という意味であることから、「X かつ Y」はFである。他の結合子についてもこのように考えて、真理値を決定することができる。また式Aの真理値をv(A) とおく。v(A) はTかFのうち、ちょうど1つを値としてとる。

・式の恒真性

式Aに現れる命題変数を X_1,X_2,. . .,X_n とする。v(X_1),v(X_2),. . .,v(X_n) がそれぞれ任意にT、Fを値にするとき、つねに v(A)=Tであるならば、A は恒真である。

5.2　式が恒真かどうか判定するアルゴリズム

論理式が恒真かどうか判定するアルゴリズムとして私たちは2つのものを勉強しました。一つは真理値表でもう一つは **Wang** のアルゴリズムです。以下でそれぞれの説明をしていきます。

6　真理値表

6.1　真理値表

一般の真理値表の形式はいかのようなものです。

原子命題	この原子命題から成る合成命題
状況: 原子命題の真理値の 可能な組合せすべて	それぞれの状況に対応する真理値

私たちが作った真理値表は、
例えば (((A and B) or (not A)) ⇒ C) のような論理式を入力すると、

```
 TRUTH-TABLE
A:(T T T T F F F F)
B:(T T F F T T F F)
C:(T F T F T F T F)
  (T T T T F F T F)
```

と出力されます。

6.2　真理値表のプログラム

truth-table のプログラムを作成する際に、次のように考えています。全体像としては λ 式を使い、そこに必要な関数のプログラムを組み込んでいきます。

λ　(変数）(式）(真理値）

まず第一に、論理記号を決めます。(1 ）のプログラムは、implication のプログラムですが、その他の and , or , not については そのまま使用します。

第二に、入力する論理式から、命題変数だけを抜き出します。なお、同じものは除きます。

第三に、代入する真理値のリストを作ります。

最後に、λ　(変数）(式）(真理値）に相当する、プログラム truth-table を作ります。

このプログラムは、**implication** のプログラムです。

```
>(defun => (a b)
   (or (not a ) b))

=>
```

このプログラムは、**and** , **or** , **not** , ⇒ , **T** , **F** , **nil** 以外のものを変数とし、それらを出力します。また、同じ変数があった 場合は、それを除きます。

```
>(defun vars (exp)
  (if (atom exp)
      (if (and (not (eq 'and exp))
               (not (eq 'or exp))
               (not (eq 'not exp))
               (not (eq '=> exp))
               (not (eq 'nil exp))
               (not (eq 'T exp))
               (not (eq 'F exp)))
          (list exp)
          'nil)
      (do ((rem (append (vars (car exp))
                        (if (null (cdr exp))
                        'nil
                        (vars (cdr exp))))
                (cdr rem))
           (sofar 'nil (if (equal (member (car rem) (cdr rem)) 'nil)
                       (append sofar (list (car rem)))
                       sofar)))
      ((null rem)
       sofar))))

VARS
```

このプログラムは、間置記法を用いて表現することを可能にしています。（通常 **Lisp** では、前置記法を用いています。）ただし、**(A and B or C)** では、正しい答えは返ってきません。**((A and B) or C)** または、**(A and (B or C))** のように、書き換えてください。

```
>(defun narabekae (exp)
     (cond ((atom exp) exp)
           ((or (eq 'and (cadr exp))
                (eq 'or (cadr exp))
                (eq '=> (cadr exp)))
            (list (cadr exp) (narabekae (car exp))
                             (narabekae (caddr exp))))
           ((or (eq 'and (car exp))
                (eq 'or (car exp))
                (eq '=> (car exp)))
            (list (car exp) (narabekae (cadr exp))
                            (narabekae (caddr exp))))
           ((eq 'not (car exp))
            (list (car exp) (narabekae (cadr exp))))
           (T 'nil)))

NARABEKAE
```

このプログラムは、割算の商だけを出力するものです。

```
>(defun sho (x y)
   (if (< 0 y)
       (if (< x y)
           0
           (do ((lst x (- lst y))
                (count 0 (+ 1 count)))
               ((< lst y)
                count)))
       nil))

SHO
```

このプログラムは、巾乗を計算するプログラムです。

```
>(defun bekijyo (kisuu beki)
   (if (zerop beki)
      1
      (do ((rem beki (- rem 1))
           (sofar kisuu (* sofar kisuu)))
          ((eq rem 1)
           sofar))))

BEKIJYO
```

このプログラムは、関数 **TF** の補助関数です。**T** と **F** の数を合わすプログラムです。

```
>(defun tf-aux (num lst)
   (if (eq num (length lst))
       lst
       (tf-aux num (append lst lst))))

TF-AUX
```

このプログラムは、命題変数の数に合わせて **T** と **F** を並べるものです。

```
>(defun TF (num)
  (do ((rem num (sho rem 2))
       (sofar 'nil (append sofar
                      (list (tf-aux (* 2 num) (append
            (do ((rem1 rem (- rem1 1))
        (sofar1 'nil (cons 't sofar1)))
       ((zerop rem1)
        sofar1))
            (do ((rem2 rem (- rem2 1))
        (sofar2 'nil (cons 'nil sofar2)))
       ((zerop rem2)
        sofar2))))))))
      ((eq 0 rem)
       sofar)))

TF
```

このプログラムは 関数 **TF** を使って引数に論理式をとり、**T** と **F** を並べるプログラムです。

```
>(defun tf-form (exp)
  (TF (bekijyo 2 (- (length (vars exp)) 1))))

TF-FORM
```

このプログラムは、先ほどの **tf-form** で出力されたものを入れることで、縦に並べ換えるプログラムです。
例えば、命題変数を **A**、**B** とすると
A=t t f f , B=t f t f を
(A,B)=(t,t)
(A,B)=(t,f)
(A,B)=(f,t)
(A,B)=(f,f) に並べ換えます。

```
>(defun tf-reform (exp)
  (let* ((head (car exp))
         (tail (cdr exp))
         (ln (length head)))
    (maplist
    #'(lambda (job-point)
        (cons
         (car job-point)
         (mapcar
                 #'(lambda (given-list)
                               (nth (- ln (length job-point)) given-list))
                           tail)))
  head)))

TF-REFORM
```

このプログラムは、**subst** という ある式の中の ある変数 をある値に置き換えるというプログラムを拡張して、ある式の中の ある変数のリストを ある値のリストに置き換えるプログラムです。

```
>(defun gsubst (new-list old-list exp)
  (do ((fst new-list (cdr fst))
       (snd old-list (cdr snd))
       (thd exp (subst (car fst) (car snd) thd)))
      ((null fst)
       (eval thd))))

GSUBST
```

このプログラムは、今までの各プログラムを組み合わせて、論理式を入れると 真理値表を出力するプログラムです。

```
>(defun truth-table (exp)
   (format T "~%TRUTH-TABLE")
    (do ((1st (vars exp) (cdr 1st))
         (2nd (subst ’f ’nil
              (tf-form exp))
              (cdr 2nd)))
        ((null 1st)
         (format T "~%"))
        (format T "~%~S:~S"
         (car 1st)
         (append (do ((rem1 (car 2nd) (cdr rem1))
                      (sofar1 ’nil (append sofar1 (list (car rem1)))))
                     ((equal (- (length (car (tf-form exp))) 1)
                             (length sofar1))
                       sofar1))
                 (list (car (last (car 2nd)))))))
   (format T "~2T~S"
    (do ((rem (tf-reform (tf-form exp)) (cdr rem))
         (sofar ’nil (append sofar
                          (list (gsubst (car rem)
                                   (vars exp)
                                   (narabekae exp))))))
        ((null rem)
         (append (do ((rem2 (subst ’f ’nil sofar) (cdr rem2))
                      (sofar2 ’nil (append sofar2 (list (car rem2)))))
                     ((equal (- (length (car (tf-form exp))) 1)
                             (length sofar2))
                      sofar2))
                 (list (car (last (subst ’f ’nil sofar))))))))

TRUTH-TABLE
```

6.3 真理値表のプログラム使用例

問題 1 菜々が風邪をひいているとき、絵里香は風邪をひいていません。
では、絵里香が風邪をひいているとき、菜々は風邪をひいていない、と言えるのでしょうか ?

問題 1 の解答
P:菜々は風邪をひいている。
Q:絵里香は風邪をひいている。
と置くと、問題 1 は以下の論理式に書き直すことができる。
$(P \to \neg Q) \to (Q \to \neg P) \cdots (f)$
(f) が恒真であるかを調べれば解答が得られる。
(f) を以下の様に書き直し、関数 truth-table で解く。

```
>(truth-table '((P => (not Q)) => (Q => (not P))))

TRUTH-TABLE
Q:(T T F F)
P:(T F T F)
  (T T T T)
NIL
```

問題 2 $\neg A \wedge (B \Rightarrow (C \vee D))$

```
>(truth-table '((not A) and (B => (C or D))))

TRUTH-TABLE
A:(T T T T T T T T F F F F F F F F)
B:(T T T T F F F F T T T T F F F F)
C:(T T F F T T F F T T F F T T F F)
D:(T F T F T F T F T F T F T F T F)
  (F F F F F F F F T T T F T T T T)
NIL
```

問題 3 $X \wedge (\neg X \vee Y) \Rightarrow Y \vee Z$

```
>(truth-table '((X and ((not X) or Y)) => (Y or Z)))

TRUTH-TABLE
X:(T T T T F F F F)
Y:(T T F F T T F F)
Z:(T F T F T F T F)
  (T T T T T T T T)
NIL
```

7 Wangのアルゴリズム

7.1 Wangのアルゴリズムとは

任意に与えられた式が恒真であるかどうかは、真理値表を使ってすべての場合をチェックすればば判定できる。しかしその作業は一般にかなりの計算量を必要とする。もう少し見通しのよい方法に**Wang**のアルゴリズムというのがある。Wangのアルゴリズムのアイディアは、結合子の種類にしたがって式を分解し、左右2つのグループに分けていく。分解が終ってすべての場合に左右両方に同じ変数があればもとの式は恒真、そうでなければ恒真でないというものである。Wangのアルゴリズムの恒真性を判定するのに便利なものに、分解の木というものがあります。

・分解の木

与えられたsequent S_0 からはじめて分解を続けたもの全体を (S_0) の「分解の木」と よび $\mathrm{T}(S_0)$ と書く。簡単にいうと、これはシークエントにある論理記号を順番にはずして分解していくものです。シークエントとは、｜の左右に式がならんでコンマ「,」で区切られているものをいう。

○ シークエントの分解

- 論理式がシークエントの左にある場合

$$\frac{\circ\ ,\ \triangle\ |\ \square\ ,\ \mathrm{A}}{\circ\ ,\mathit{not}\ \mathrm{A},\ \triangle\ |\ \square}\ \mathit{not}\ \text{左}$$

$$\frac{\circ\ ,\ \mathrm{A},\ \mathrm{B},\ \triangle\ |\ \square}{\circ\ ,\ \mathrm{A}\ \mathit{and}\ \mathrm{B},\ \triangle\ |\ \square}\ \mathit{and}\ \text{左}$$

$$\frac{\circ\ ,\ \mathrm{A},\ \triangle\ |\ \square\ ,\ \circ\ ,\ \mathrm{B},\ \triangle\ |\ \square}{\circ\ ,\ \mathrm{A}\ \mathit{or}\ \mathrm{B},\ \triangle\ |\ \square}\ \mathit{or}\ \text{左}$$

$$\frac{\circ\ ,\ \mathrm{B},\ \triangle\ |\ \square\ ,\ \circ\ ,\ \triangle\ |\ \square\ ,\ \mathrm{A}}{\circ\ ,\ \mathrm{A}\ =>\ \mathrm{B},\ \triangle\ |\ \square}\ =>\ \text{左}$$

- 論理式がシークエントの右にある場合

$$\frac{\mathrm{A},\ \bullet\ |\ \blacktriangle\ ,\ \blacksquare}{\bullet\ |\ \blacktriangle\ ,\mathit{not}\ \mathrm{A},\ \blacksquare}\ \mathit{not}\ \text{右}$$

$$\frac{\bullet\ |\ \blacktriangle\ ,\ \mathrm{A},\ \blacksquare\ ,\ \bullet\ |\ \blacktriangle\ ,\ \mathrm{B}\blacksquare}{\bullet\ |\ \blacktriangle\ ,\ \mathrm{A}\ \mathit{and}\ \mathrm{B},\ \blacksquare}\ \mathit{and}\ \text{右}$$

$$\frac{\bullet\ |\ \blacktriangle\ ,\ \mathrm{A},\ \mathrm{B},\ \blacksquare}{\bullet\ |\ \blacktriangle\ ,\ \mathrm{A}\ \mathit{or}\ \mathrm{B},\ \blacksquare}\ \mathit{or}\ \text{右}$$

$$\frac{\mathrm{A},\ \bullet\ |\ \blacktriangle\ ,\ \mathrm{B},\ \blacksquare}{\bullet\ |\ \blacktriangle\ ,\ \mathrm{A}\ =>\ \mathrm{B},\ \blacksquare}\ =>\ \text{右}$$

上の法則で

○ , A $*$ B, △ ｜□ または ● ｜▲ , A $*$ B, ■

は次のことを示す。*は結合子の一つであり、

○ , △ , □ , ● , ▲ , ■ ,

はそれぞれコンマで区切って書かれた式の有限列である。ただし空な列も認める。○ , ● , ▲ , は*を一番外の結合子としてもつ式を含まない。

シークエントの分解には、関係する結合子と ｜の左右の区別によって、not 左 とか and 右 などと名前をつける。分解は下から上にむかっておこなわれる。

7.2 Wangのアルゴリズムのプログラム

論理式が恒真であるかどうかをWangのアルゴリズムに従って判定する際に、論理式がシークエントの右にあるか、左にあるかで論理式の分解を分けて考える。ここでは説明をわかりやすくするために、具体例を挙げて説明を進めることにする。

$X \wedge (\neg X \vee Y) \mid Y \vee Z$

を関数wangで分解するために以下の様に書き直す。

$$(\underline{(X \text{ and } ((\text{not } X) \text{ or } Y))}\ !\ (Y \text{ or } Z))\cdots(a)$$

⇑

(1) 関数l-wang-aux

関数l-wang-auxはシークエントの左側にある論理式を分解するプログラムで、(1)で分解する際、分解している論理式より後にある変数や論理式をリストとして出力するためには(a)のリスト全体のCDRをとれば良い。

・(a)を(1)で分解すると、(a)のCDRが(! (Y or Z))になるので、(6)より

$$(X\ ((\text{not } X) \text{ or } Y)\ !\ \underline{(Y \text{ or } Z)})\cdots(b)$$

となる。 ⇑

(2) 関数r-wang-aux

関数r-wang-auxはシークエントの右側にある論理式を分解するプログラムで、(2)で分解する際、シークエントの左側にある変数や論理式をリストとして出力するためには新たに関数を作らねばならない。その関数がleftである。

・(b)を(2)で分解すると、(b)の関数leftが(X ((not X) or Y))になるので、(7)より

$$\underline{(X\ ((\text{not } X) \text{ or } Y)}\ !\ Y\ Z)\cdots(c)$$

となる。 ⇑

(3) 関数l-wang-aux

(3)で分解する際、シークエントの左側にorがあるので、(c)が2つのリストに分かれる。

・(c)を(3)で分解すると、(8)より

$$\underline{((X\ (\text{not } X)\ !\ Y\ Z)\ (X\ Y\ !\ Y\ Z))}\cdots(d)$$

となる。 ⇑

(4) 関数wang-aux2

関数wang-aux2はリストが2つに分かれてもうまく論理式が分解でき、format

をきれいに出力するためのプログラムである。
・(d) を (4) で分解すると、(9) より

$\underline{((X\ !\ Y\ Z\ X)\ (X\ Y\ !\ Y\ Z))}\cdots(e)$

となる。　⇑

(5) 関数 atom-lst

関数 atom-lst は変数が全てアトムならば (e) のリスト全体を出力し、そうでなければ nil を出力するプログラムである。
・(e) を (5) で評価すると、(10) より ((X ! Y Z X) (X Y ! Y Z))···(f) が出力されて、(e) と (f) は関数 wang の (11) より等しいので終了が判定できる。

これらのプログラムを組み合わせて論理式の分解を自動的にし、分解の木を出力するプログラムが関数 wang である。但し、通常の分解の木は下から上に書くが、このプログラムでは format の都合上、上から下に出力することにする。

関数 l-wang-aux のプログラム

```
>(defun l-wang-aux (exp)
   (do ((1st exp (cdr 1st))
        (2nd 'nil (append 2nd (list (car 1st))))
        (3rd 'nil (if (atom (car 1st))
                      (atom-1st 1st)
                      (cond ((eq (cadar 1st) 'and)
                             (append 2nd (list (caar 1st))
                                     (list (caddar 1st)) (cdr 1st)))
                                                    ···(6)
                            ((eq (cadar 1st) 'or)
                             (list (append 2nd (list (caar 1st)) (cdr 1st))
                                   (append 2nd (list (caddar 1st)) (cdr 1st))))
                                                    ···(8)
                            ((eq (cadar 1st) '=>)
                             (list (append 2nd (list (caddar 1st)) (cdr 1st))
                                   (append 2nd (cdr 1st) (list (caar 1st)))))
                            ((eq (caar 1st) 'not)
                             (append 2nd (cdr 1st)
                                     (cdar 1st)))
                            (T 'nil)))))
       ((not (atom 3rd))
        3rd)))

L-WANG-AUX
```

関数 **r-wang-aux** のプログラム

```
>(defun r-wang-aux (exp)
   (do ((4th (cdr (member '! exp)) (cdr 4th))
        (5th 'nil (append 5th (list (car 4th))))
        (6th 'nil (if (atom (car 4th))
                      (atom-lst exp)
                      (if (eq (length (car 4th)) 1)
                          'nil
                          (cond ((eq (cadar 4th) 'and)
                                 (list (append (left exp) '(!) 5th
                                        (list (caar 4th)) (cdr 4th))
                                       (append (left exp) '(!) 5th
                                        (list (caddar 4th)) (cdr 4th))))
                                ((eq (cadar 4th) 'or)
                                       (append (left exp) '(!) 5th
                                        (list (caar 4th))
                                        (list (caddar 4th)) (cdr 4th)))
                                                     …(7)
                                ((eq (cadar 4th) '=>)
                                       (append (list (caar 4th))
                                        (left exp) '(!) 5th
                                        (list (caddar 4th)) (cdr 4th)))
                                ((eq (caar 4th) 'not)
                                       (append (cdar 4th)  (left exp)
                                         '(!) 5th (cdr 4th)))
                                (T 'nil))))))
       ((not (atom 6th))
        6th)))
R-WANG-AUX
```

関数 **left** のプログラム

```
>(defun left (exp)
    (do ((1st exp (cdr 1st))
         (2nd 'nil (append 2nd (list (car 1st)))))
        ((eq (car 1st) '!)
         2nd)))
LEFT
```

関数 wang-aux1 のプログラム

このプログラムは関数 wang の補助関数で、関数 l-wang-aux と関数 r-wang-aux を組み合わせたプログラムである。

```
>(defun wang-aux1 (exp)
    (do ((1st (left exp) (cdr 1st))
        (2nd (cdr (member ’! exp)) (cdr 2nd))
         (3rd ’nil (if (atom (car 1st))
                       (if (atom (car 2nd))
                           (atom-1st exp)
                           (if (or (eq (cadar 2nd) ’and)
                                   (eq (cadar 2nd) ’or)
                                   (eq (cadar 2nd) ’=>)
                                   (eq (caar 2nd) ’not))
                               (append 3rd (r-wang-aux exp))
                               ’nil))
                       (if (or (eq (cadar 1st) ’and)
                               (eq (cadar 1st) ’or)
                               (eq (cadar 1st) ’=>)
                               (eq (caar 1st) ’not))
                           (append 3rd (l-wang-aux exp))
                            ’nil))))
        ((not (atom 3rd))
         3rd)))

WANG-AUX1
```

関数 **wang-aux2** のプログラム

```
(defun wang-aux2 (exp)
   (if (equal (atom-lst (car exp)) (car exp))
       (if (equal (atom-lst (cadr exp)) (cadr exp))
           exp
           (if (eq (length (cadr exp)) 2)
               (cond ((eq (caadr exp) '!)
                      (list (car exp) (wang-aux1 (cadr exp))))
                     ((eq (cadadr exp) '!)
                      (list (car exp) (wang-aux1 (cadr exp))))
                     ((eq (length (caadr exp)) 2)
                      (cond ((eq (caaadr exp) '!)
                             (list (car exp) (list (wang-aux1 (caadr exp))
                                                   (cadadr exp))))
                            ((eq (cadr (caadr exp)) '!)
                             (list (car exp) (list (wang-aux1 (caadr exp))
                                                   (cadadr exp))))
                            ((eq (length (cadadr exp)) 2)
                             (cond ((eq (car (cadadr exp)) '!)
                                    (list (car exp)
                                          (list (caadr exp)
                                                (wang-aux1 (cadadr exp)))))
                                   ((eq (cadr (cadadr exp)) '!)
                                    (list (car exp)
                                          (list (caadr exp)
                                                (wang-aux1 (cadadr exp)))))
                                   (T (list (car exp)
                                            (list (wang-aux2 (caadr exp))
                                                  (wang-aux2 (cadadr exp)))))))
                            ((equal (atom-lst (cadadr exp)) (cadadr exp))
                             (list (car exp) (list (wang-aux1 (caadr exp))
                                                   (cadadr exp))))
                            (T (list (car exp) (list (caadr exp)
                                                     (wang-aux1 (cadadr exp)))))))
                     ((equal (atom-lst (caadr exp)) (caadr exp))
                      (list (car exp) (list (caadr exp)
                                            (wang-aux1 (cadadr exp)))))
                     (T (list (car exp) (list (wang-aux1 (caadr exp))
                                              (cadadr exp)))))
```

```
               (list (car exp) (wang-aux1 (cadr exp)))))
       (if (eq (length (car exp)) 2)
           (cond ((eq (caar exp) '!)
                  (list (wang-aux1 (car exp)) (cadr exp)))
                 ((eq (cadar exp) '!)
                  (list (wang-aux1 (car exp)) (cadr exp)))
                 ((eq (length (caar exp)) 2)
                  (cond ((eq (caaar exp) '!)
                         (list (list (wang-aux1 (caar exp))
                                     (cadar exp)) (cadr exp)))
                        ((eq (cadaar exp) '!)
                         (list (list (wang-aux1 (caar exp))
                                     (cadar exp)) (cadr exp)))
                        ((eq (length (cadar exp)) 2)
                         (cond ((eq (caadar exp) '!)
                                (list (list (caar exp) (wang-aux1 (cadar exp)))
                                      (cadr exp)))
                               ((eq (cadr (cadar exp)) '!)
                                (list (list (caar exp) (wang-aux1 (cadar exp)))
                                      (cadr exp)))
                               (T (list (list (wang-aux2 (caar exp))
                                              (wang-aux2 (cadar exp)))
                                        (cadr exp)))))
                        ((equal (atom-lst (cadar exp)) (cadar exp))
                         (list (list (wang-aux1 (caar exp)) (cadar exp))
                               (cadr exp)))
                        (T (list (list (caar exp) (wang-aux1 (cadar exp)))
                                 (cadr exp)))))
                 ((equal (atom-lst (caar exp)) (caar exp))
                  (list (list (caar exp) (wang-aux1 (cadar exp)))
                        (cadr exp)))
                 (T (list (list (wang-aux1 (caar exp)) (cadar exp))
                          (cadr exp))))
           (list (wang-aux1 (car exp)) (cadr exp)))))
WANG-AUX2
```

関数 **atom-lst** のプログラム

リストが１つ以上の場合に対して使える。

```
>(defun atom-lst (exp)
    (if (eq (length exp) 2)
        (if (eq (car exp) '!)
            (atom-lst-aux2 exp)
            (if (eq (cadr exp) '!)
                (atom-lst-aux2 exp)
                (if (equal (atom-lst-aux1 (car exp)) (car exp))
                    (if (equal (atom-lst-aux1 (cadr exp)) (cadr exp))
                        exp ···(10)
                        (list (car exp) (atom-lst (cadr exp))))
                    (list (atom-lst (car exp)) (atom-lst (cadr exp))))))
        (atom-lst-aux2 exp)))

ATOM-LST
```

関数 atom-lst-aux1 のプログラム

このプログラムは関数 atom-lst の補助関数で、リストが 1 つか 2 つの場合に対してのみ使える。

```
>(defun atom-lst-aux1 (exp)
    (if (eq (length exp) 2)
        (if (eq (car exp) ’!)
            (atom-lst-aux2 exp)
            (if (eq (cadr exp) ’!)
                (atom-lst-aux2 exp)
                (list (atom-lst-aux2 (car exp))
                      (atom-lst-aux2 (cadr exp)))))
        (atom-lst-aux2 exp)))

ATOM-LST-AUX1
```

関数 atom-lst-aux2 のプログラム

このプログラムは関数 atom-lst と関数 atom-lst-aux1 の補助関数で、リストが 1 つの場合に対してのみ使える。

```
>(defun atom-lst-aux2 (exp)
    (do ((1st exp (cdr 1st))
         (2nd ’nil (if (atom (car 1st))
                       (append 2nd (list (car 1st)))
                       ’nil)))
        ((not (atom (car 1st)))
         ’nil)
        (if (null 1st)
            (return 2nd))))

ATOM-LST-AUX2
```

関数 wang のプログラム

```
>(defun wang (exp)
    (format T "~%~S" exp)
    (if (equal (atom-lst exp) exp) …(11)
        'nil
        (if (eq (length exp) 2)
            (if (eq (car exp) '!)
                (wang (wang-aux1 exp))
                (if (eq (cadr exp) '!)
                    (wang (wang-aux1 exp))
                    (wang (wang-aux2 exp))))
            (wang (wang-aux1 exp)))))

WANG
```

7.3 Wangのアルゴリズムのプログラム使用例

問題1　菜々が風邪をひいているとき、絵里香は風邪をひいていません。では、絵里香が風邪をひいているとき、菜々は風邪をひいていない、と言えるのでしょうか？

問題1の解答

P:菜々は風邪をひいている。

Q:絵里香は風邪をひいている。

と置くと、問題1は以下の論理式に書き直すことができる。

$(P \to \neg Q) \to (Q \to \neg P) \cdots (f)$

(f)が恒真であるかを調べれば解答が得られる。

(f)を以下の様に書き直し、関数wangで分解する。

```
>(wang '(! ((P =>(not Q)) => (Q => (not P)))))

(! ((P => (NOT Q)) => (Q => (NOT P))))
((P => (NOT Q)) ! (Q => (NOT P)))
(((NOT Q) ! (Q => (NOT P))) (! (Q => (NOT P)) P))
((! (Q => (NOT P)) Q) (! (Q => (NOT P)) P))
((Q ! (NOT P) Q) (! (Q => (NOT P)) P))
((P Q ! Q) (! (Q => (NOT P)) P))
((P Q ! Q) (Q ! (NOT P) P))
((P Q ! Q) (P Q ! P))
NIL
```

シークエントの左右にそれぞれQ、Pがあるので恒真であることわかる。

従って、絵里香が風邪をひいているとき、菜々は風邪をひいていない、と言える。

問題 2　「女の子は、競歩が好きでないなら、旅行が好きである。」
これが真実なら、旅行も競歩も好きでない者のなかに、女の子は、
1. いる、
2. いない、
どちらでしょうか ?

問題 2 の解答
P:女の子は競歩が好きである。
Q:女の子は旅行が好きである。
と置くと、1. は以下の論理式に書き直すことができる。
$(\neg P \to Q) \to (\neg P \land \neg Q)\cdots(g)$
また、2. は以下の論理式に書き直すことができる。
$(\neg P \to Q) \to \neg(\neg P \land \neg Q)\cdots(h)$
(g) か (h) のどちらが恒真であるかを調べれば解答が得られる。
(g) 、(h) を以下の様に書き直し、関数 wang で分解する。

(g) の場合

```
>(wang ’(! (((not p) => q) => ((not p) and (not q)))))

(! (((NOT P) => Q) => ((NOT P) AND (NOT Q))))
(((NOT P) => Q) ! ((NOT P) AND (NOT Q)))
((Q ! ((NOT P) AND (NOT Q))) (! ((NOT P) AND (NOT Q)) (NOT P)))
(((Q ! (NOT P)) (Q ! (NOT Q))) (! ((NOT P) AND (NOT Q)) (NOT P)))
(((P Q !) (Q ! (NOT Q))) (! ((NOT P) AND (NOT Q)) (NOT P)))
(((P Q !) (Q Q !)) (! ((NOT P) AND (NOT Q)) (NOT P)))
(((P Q !) (Q Q !)) ((! (NOT P) (NOT P)) (! (NOT Q) (NOT P))))
(((P Q !) (Q Q !)) ((P ! (NOT P)) (! (NOT Q) (NOT P))))
(((P Q !) (Q Q !)) ((P P !) (! (NOT Q) (NOT P))))
(((P Q !) (Q Q !)) ((P P !) (Q ! (NOT P))))
(((P Q !) (Q Q !)) ((P P !) (P Q !)))
NIL
```

(h) の場合

```
>(wang '(! (((not p) => q) => (not ((not p) and (not q))))))

(! (((NOT P) => Q) => (NOT ((NOT P) AND (NOT Q)))))
(((NOT P) => Q) ! (NOT ((NOT P) AND (NOT Q))))
((Q ! (NOT ((NOT P) AND (NOT Q))))
 (! (NOT ((NOT P) AND (NOT Q))) (NOT P)))
((((NOT P) AND (NOT Q)) Q !) (! (NOT ((NOT P) AND (NOT Q))) (NOT P)))
(((NOT P) (NOT Q) Q !) (! (NOT ((NOT P) AND (NOT Q))) (NOT P)))
(((NOT Q) Q ! P) (! (NOT ((NOT P) AND (NOT Q))) (NOT P)))
((Q ! P Q) (! (NOT ((NOT P) AND (NOT Q))) (NOT P)))
((Q ! P Q) (((NOT P) AND (NOT Q)) ! (NOT P)))
((Q ! P Q) ((NOT P) (NOT Q) ! (NOT P)))
((Q ! P Q) ((NOT Q) ! (NOT P) P))
((Q ! P Q) (! (NOT P) P Q))
((Q ! P Q) (P ! P Q))
NIL
```

(g) の場合、シークエントの左右に同じ変数がないので恒真でないことがわかる。
(h) の場合、シークエントの左右にそれぞれ Q、P があるので恒真であることがわかる。

従って、2. いない、が正解になる。

8　さいごに

やっと これらのプログラムを完成させることが出来ました。Lisp は今までやったプログラミング言語とは全く雰囲気が違い 最初はとまどっていましたが、ひとつひとつ関数を作っていくうちに、すごく勉強になりました。このように勉強しながら作っていったものなので、要領が悪い所や改良できる点もたくさんあると思います。しかし自分達でこれらのように動くものが出来たことは嬉しいことであり、自信にもなりました。また、ここまでこれたのも八杉満利子教授の御指導に多分によるもので、大変感謝しております。ありがとうございました。

私達も卒業するにあたり、何かを形にして残せていけたらと思い、はじめたものなので、今後 八杉教授もしくは他の先生方にも授業の中で使っていただけるなら、それが一番嬉しいことです。

さいごに、４年間 何とかここまで来られたことに 八杉教授をはじめ各先生方、事務の方、先輩方にも 感謝致しまして この卒業論文を締めくくらせていただきます。

川原 大祐　　實藤 法道　　田中 久雄