■1 群（信号・システム）- 8 編（論理回路）

1 章 論理代数と論理関数

（執筆者：湊　真一）[2009 年 2 月 受領]

■概要■

0,1 の 2 値を扱う論理代数は，論理回路の設計や解析を行う上での数学的基礎を与えるものである．19 世紀に Boole により論理代数（いわゆるブール代数）が体系化され，更に 20 世紀中頃になり，Shannon により論理代数に基づく論理回路設計法が示された．それ以降，様々な論理設計のための技法が研究開発されている．近年では，それらの多くの技法は，計算機上にプログラムとして実装され，人手で扱うことが到底困難な大規模な論理回路を，計算機の力を借りて現実的な処理時間で設計することが可能になってきている．しかし，任意の問題に対する完全な設計自動化は困難であるため，依然として人間の関与も必要である．論理回路設計の仕組みについても設計者がある程度理解し，設計自動化プログラムを利用しながら，不満足な部分を人間が補完していく必要があると考えられる．

本章では，論理回路設計のために必要な基礎概念として，論理代数と論理関数について解説する．まず，論理代数の数学的定義を示し，次に論理関数と論理式に関する基本的事項について述べる．続いて，特徴的な性質をもつ論理関数をいくつか取り上げ，それらの数学的な諸性質を示す．更に，論理関数データを計算機上で効率よく処理するためのデータ構造とその演算処理アルゴリズムについて概観する．特に，近年盛んに用いられている BDD（二分決定グラフ）については，やや詳しく述べる．

【本章の構成】

本章では，論理代数（1-1 節），論理関数と論理式（1-2 節），論理関数の諸性質（1-3 節），論理関数の計算機上での処理（1-4 節）に関して，基礎理論と応用技術の両面から解説する．1-1〜1-3 節では，数学的な基礎概念の説明と，それらの例示を中心に解説する．一方，1-4 節では，数学的な厳密さよりも実用的，直感的な理解を重視し，最近提案されている新しい技法を中心に概観する．紙面の制約から，すべての事項を完全に網羅することはできないが，必要に応じて参考文献をあげておくので，興味ある読者は参照していただきたい．

■1 群 - 8 編 - 1 章

1-1 論理代数

（執筆者：湊　真一）[2009 年 2 月 受領]

本節では，論理回路の設計や解析の数学的基礎となる論理代数について述べる．以下では特に断らない限り，0, 1 の 2 値論理（switching logic）に関する代数を扱うものとする．

1-1-1 論理代数の定義

x が**論理値**（switching value）をもつとは，x が 0, 1 のどちらかの値を取ること，すなわち

$$x \neq 0 \text{ ならば } x = 1.$$
$$x \neq 1 \text{ ならば } x = 0.$$

が成り立つことをいう．このような 2 値の論理値を，**真理値**（truth value）あるいは**ブール値**（Boolean value）と呼ぶこともある．

論理代数（switching algebra）は，論理値 (0,1) に関する，**論理積**（AND），**論理和**（OR），**否定**（NOT）の三つの演算からなる代数系として定義される．ここで，論理積，論理和，否定は以下のように定義される二項演算及び単項演算である．

論理積 $(\cdot)$	論理和 $(+)$	否定 $(\bar{})$
$0 \cdot 0 = 0$	$0 + 0 = 0$	$\bar{0} = 1$
$0 \cdot 1 = 0$	$0 + 1 = 1$	$\bar{1} = 0$
$1 \cdot 0 = 0$	$1 + 0 = 1$	
$1 \cdot 1 = 1$	$1 + 1 = 1$	

論理積 $x \cdot y$ は，x と y がともに 1 のとき，その値が 1 となる演算である．論理和 $x + y$ は，x と y の少なくとも一方が 1 のとき，その値が 1 となる演算である．なお，本章では $\cdot, +, \bar{}$ という演算子記号を用いるが，論理演算子の記法は一定しておらず，文献により違いが見られる．算術演算子との混同を避けるために，論理積を $\wedge$, 論理和を $\vee$ とする記法もしばしば用いられている．否定演算も，$\sim x$ や x' のように記述する場合もある．

上記の論理演算子と括弧 ( )，及び，任意の個数の**論理変数**（switching variable）と，必要に応じて定数値 0,1 を組み合わせて計算手順を表した式を，**論理式**（switching expression）または**ブール式**（Boolean expression）と呼ぶ．例えば，

$$\overline{(x + y)} \cdot z$$

は，論理変数 x, y の論理和の否定を求め，更に z との論理積を計算する，という演算手順を表している．

論理変数及びその否定のことを**リテラル**（literal）と呼ぶ．否定の付かないリテラル（x, y, z など）を**正リテラル**（positive literal），否定の付いたリテラル（$\bar{x}, \bar{y}, \bar{z}$ など）を**負リテラル**（negative literal）と呼ぶ．なお，慣習として，括弧による指定がない場合には，論理積

は論理和よりも優先して計算されるものとする．また論理積の演算子（ · ）は省略されることがある．

1-1-2 論理代数の性質

論理代数は，以下に示す法則を満たす．

- **べき等則** (idempotency):
  $x \cdot x = x, \quad x + x = x.$
- **零元** (zero element):
  $x \cdot 0 = 0, \quad x + 1 = 1.$
- **単位元** (identity element):
  $x \cdot 1 = x, \quad x + 0 = x.$
- **交換則** (comutativity):
  $x \cdot y = y \cdot x, \quad x + y = y + x.$
- **結合則** (associativity):
  $(x \cdot y) \cdot z = x \cdot (y \cdot z), \quad (x + y) + z = x + (y + z).$
- **相補則** (complementation):
  $x \cdot \overline{x} = 0, \quad x + \overline{x} = 1.$
- **分配則** (distributivity):
  $x \cdot (y + z) = (x \cdot y) + (x \cdot z), \quad x + (y \cdot z) = (x + y) \cdot (x + z).$
- **吸収則** (absorption):
  $x + (x \cdot y) = x, \quad x \cdot (x + y) = x.$
- **二重否定則** (involution):
  $\overline{\overline{x}} = x.$
- **ド-モルガンの法則** (De Morgan's laws):
  $\overline{x + y} = \overline{x} \cdot \overline{y}, \quad \overline{x \cdot y} = \overline{x} + \overline{y}.$

ここで = の記号は, 両辺の計算結果が常に等しい, すなわち両辺の論理式が**等価**(equivalent)であることを表す。上記の法則が正しいことは，論理変数にすべての 0,1 の組合せ（1 変数の場合は 2 通り，2 変数の場合は 4 通り，3 変数の場合は 8 通り）をそれぞれ代入して確かめてみることにより，容易に証明できる．例えば分配則は,

x	y	z	$x\cdot(y+z)$	$(x\cdot y)+(x\cdot z)$	$x+(y\cdot z)$	$(x+y)\cdot(x+z)$
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	1	1
1	0	0	0	0	1	1
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	1	1	1

となり，すべての場合について正しいことが確かめられる．このような証明法を**完全帰納法**（perfect induction）という．

1-1-3 ド-モルガンの法則の一般化と双対原理

前に示したド-モルガンの法則は 2 変数の論理式に関するものであったが，これは任意の多変数に一般化できる．すなわち，

$$\begin{aligned}\overline{x_1+x_2+\ldots+x_n} &= \overline{x_1}\cdot\overline{x_2}\cdot\ldots\cdot\overline{x_n},\\ \overline{x_1\cdot x_2\cdot\ldots\cdot x_n} &= \overline{x_1}+\overline{x_2}+\ldots+\overline{x_n}.\end{aligned}$$

が成り立つ．このことは，$(n-1)$ 変数で成り立つことを仮定すれば n 変数で成り立つことが容易に示せるので，数学的帰納法により証明できる．

ド-モルガンの法則，分配則，及び二重否定則を繰り返し適用することにより，任意の論理式 E の否定は，次のような論理式 E^C で表現できる．

- E の論理積演算子 ($\cdot$) を論理和演算子 $(+)$ に，論理和演算子を論理積演算子に，それぞれ置き換える．
- E の正リテラル x_i を負リテラル $\overline{x_i}$ に，負リテラル $\overline{x_i}$ を正リテラル x_i に，それぞれ置き換える．
- E の定数 0 を 1 に，1 を 0 に，それぞれ置き換える．
- E の括弧と，論理変数に直接かからない否定演算子はそのまま保持する．

つまり，E の構造を保ったままで，E に含まれる積和演算子・リテラル・定数にそれぞれ置き換え規則を適用するだけで，E の否定を表す論理式 E^C を得ることができる．例えば，論理式 $E:\overline{(x+\overline{y})}\cdot z$ の否定は，$E^C:\overline{(\overline{x}\cdot y)}+\overline{z}$ と記述できる．更に，E^C から E の逆方向に対しても全く同じことが成り立つ．

上記の置換処理で得られた E^C に含まれる各リテラルの正負を反転させたものを，E に対する**双対**（dual）な論理式と呼ぶ．上記の例では，$\overline{(x\cdot\overline{y})}+z$ となる．双対な論理式は，元の論理式の 1, 0 の真，偽の意味を反転させたものであると考えることができる．論理の真を 1，偽を 0 に割り当てる場合を**正論理**（positive logic），その逆に真を 0，偽を 1 に割り当てる場合を**負論理**（negative logic）という．

本節で示した論理代数の各法則を見ると，二重否定則以外はすべて，双対な論理式に関する一対の規則により構成されていることが分かる．つまり，論理代数における種々の法則や定理は，正論理または負論理の一方について正しさが示されれば，もう一方についても全く同様のことが成り立つ．論理代数がもつこのような性質を**双対原理**（duality principle）と呼ぶ．

1-1-4 スイッチング代数とブール代数

論理代数のことを**ブール代数**（Boolean algebra）と呼ぶことがしばしばある．G. Boole が提案したブール代数は，積・和・否定の性質を公理として定め，それを満たす代数系として定義したものである．実は，本節で示した論理代数，いわゆるスイッチング代数は，ブール代数に属する代数系の一つであり，ブール代数の公理系を満たす数学的な代数系はほかにも存在する．しかしそのような代数系の族の中で，実用的に広く用いられているのはスイッチング代数だけであるため，多くの場合，ブール代数は単にスイッチング代数の同義語として使われている．ブール式（Boolean expression），ブール値（Boolean value），ブール関数（Boolean function）などの用語についても同様である．

1-1-5 排他的論理和

ここまで，論理積・論理和・否定の三つの演算からなる論理代数について述べてきたが，それ以外にも論理式・論理関数で用いられる二項演算はいくつか存在する．中でも重要なものとして**排他的論理和**（exclusive-OR: EXOR）演算がある．

排他的論理和 ($\oplus$)

$0 \oplus 0 = 0$
$0 \oplus 1 = 1$
$1 \oplus 0 = 1$
$1 \oplus 1 = 0$

排他的論理和 $x \oplus y$ は，x と y が異なる値のときに 1 を出力し，そうでないときは 0 を出力する．つまり，$x \oplus y = (x \cdot \bar{y}) + (\bar{x} \cdot y)$ とも表せる．この演算は，以下の法則を満たす．

- **交換則**（comutativity）：
  $x \oplus y = y \oplus x$.
- **結合則**（associativity）：
  $(x \oplus y) \oplus z = x \oplus (y \oplus z)$.
- **分配則**（distributivity）：
  $x \cdot (y \oplus z) = (x \cdot y) \oplus (x \cdot z)$.
- その他:
  $x \oplus 0 = x, \quad x \oplus 1 = \bar{x},$
  $x \oplus x = 0, \quad x \oplus \bar{x} = 1.$

排他的論理和は，2 の剰余系（modulo-2）における加算としても知られており，1 の個数が奇数か偶数かを計算することから，**パリティ**（parity）演算とも呼ばれている．実用的にも重要な意味をもつ．

1-1-6 種々の二項論理演算

排他的論理和のほかにも，種々の二項論理演算が存在し得る．一般に，二項論理演算は，二つの論理値から一つの論理値への関数として定義される．そこで，

x	y	$x \circ y$
0	0	a_{00}
0	1	a_{01}
1	0	a_{10}
1	1	a_{11}

のように，$a_{00}, a_{01}, a_{10}, a_{11}$ の四つの値を考えると，それぞれに 0,1 を割り当てることにより，以下に示すような 16 種類の論理演算を考えることができる．

a_{00}	a_{01}	a_{10}	a_{11}	演算を表す論理式	演算の通称
0	0	0	0	0	恒偽（inconsistency）
0	0	0	1	$x \cdot y$	論理積（AND）
0	0	1	0	$x \cdot \bar{y}$	
0	0	1	1	x	
0	1	0	0	$\bar{x} \cdot y$	
0	1	0	1	y	
0	1	1	0	$x \oplus y$	排他的論理和（EXOR）
0	1	1	1	$x + y$	論理和（OR）
1	0	0	0	$\overline{x + y}$	否定論理和（NOR）
1	0	0	1	$\overline{x \oplus y}$	同値（equivalence）
1	0	1	0	$\bar{y}$	
1	0	1	1	$x + \bar{y}$	含意（implication）
1	1	0	0	$\bar{x}$	否定（NOT）
1	1	0	1	$\bar{x} + y$	含意（implication）
1	1	1	0	$\overline{x \cdot y}$	否定論理積（NAND）
1	1	1	1	1	恒真（tautology）

これらのうち，常に定数値を返すものが 2 種類，1 変数にのみ依存するものが 4 種類あるので，真の意味での二項論理演算は 10 種類である．更にそれらをよく吟味してみると，論理変数の一方または演算結果に否定をつけただけの変化形が多く含まれていることが分かる．そのような変化形を一つのグループとしてまとめると，論理積（4 種類），論理和（4 種類），排他的論理和（2 種類）の三つのグループに分類でき，それですべてであるということが分かる．

■参考文献

1) Donald E. Knuth, “Section 7.1: Zeros and Ones,” In “The Art of Computer Programming,” vol.4, Fascicle 0, Addison-Wesley, pp.47-133, 2008.
2) Zvi Kohavi, “Switching and Finite Automata Theory,” second edition, McGraw-Hill, 1978.
3) 笹尾　勤, “ 論理設計 スイッチング回路理論, ” 第 4 版, 近代科学社, 2005.

■1 群 - 8 編 - 1 章

1-2 論理関数と論理式

（執筆者：湊　真一）[2009 年 2 月 受領]

本節では，論理関数とそれを表現する論理式に関する基本的事項について述べる．

1-2-1 論理関数

$\{0,1\}^n \to \{0,1\}$ というかたちの関数を，n-入力の**（二値）論理関数**（logic function）と呼ぶ．**スイッチング関数**（switching function），あるいは**ブール関数**（Boolean function）と呼ばれることもある．

n 個の論理変数 $x_1, x_2, \ldots x_n$ を扱う論理式 E は，各々の論理変数に 0,1 の論理値を代入する任意の組合せ（全部で 2^n 通り）に対して，論理式の計算手順に従い，0,1 いずれかの計算結果をもたらす．つまり論理式 E は，ある論理関数 $F(x_1, x_2, \ldots x_n)$ を定義していることになる．

例えば，論理関数 F が論理式 $(\bar{x}+y)\cdot z$ により定義されるとき，

$$F(x,y,z) = (\bar{x}+y)\cdot z$$

または単に

$$F = (\bar{x}+y)\cdot z$$

と書く．このとき，論理関数の入力を表す x, y, z を**入力変数**（input variable），論理関数の計算結果を表す F を**出力変数**（output variable）と呼んで区別する場合がある（F は関数名であると同時に，関数の出力値を表す変数としても使用されることがある）．

上記の例では，$x=0, y=0, z=1$ に対しては，$F(0,0,1) = (\bar{0}+0)\cdot 1 = 1$ となる．同様にして，すべての 8 通りの組合せに対する F の出力値は，以下の表に示すとおりとなる．

$F = (\bar{x}+y)\cdot z$

x	y	z	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

このように，論理関数のすべての入力組合せに対する出力値を列挙した表を，**真理値表**（truth table）と呼ぶ．真理値表は，論理関数を**一意**（unique）に表現する．すなわち，一つの真理値表はただ一つの論理関数に対応し，一つの論理関数はただ一つの真理値表により表現される．

一方，論理式は論理関数の一意な表現ではない．一つの論理式は，ただ一つの論理関数を表現するが，一つの論理関数は，何通りもの**等価**（equivalent）な論理式で表現できる．例えば，論理式 $(\bar{x}\cdot z)+(y\bar{z})$ の各入力変数に 0,1 を代入すると，上記と全く同じ真理値表が得られる．したがって，$(\bar{x}+y)\cdot z$ と $(\bar{x}\cdot z)+(y\cdot z)$ は，論理式のかたちは異なるが等価であり，同じ論理関数を表している．

論理式は，ある一つの論理関数を何通りにも表せるが，これによって表せない論理関数はない．つまり任意の論理関数に対して，それを表す論理式が少なくとも一つは存在する．すなわち，論理式は論理関数の**完全**（complete）（または**万能**（universal））な表現であるといえる．論理式の完全性は，次の 1-2-2 節に示す積和標準形を用いて，任意の論理関数が表現できることにより示される．

一つの論理関数を表す論理式は，冗長なものを許せば，いくらでも多数の論理式をつくることができる．与えられた論理式に対して，冗長な部分があればこれを取り除き，より簡単な等価な論理式を得ることは，論理式の**簡単化**（simplification）と呼ばれ，実用上重要な技術である．前に示した種々の論理代数の法則を適用することにより，逐次的な改善を行うことは可能である．しかし，演算回数が最小の論理式を現実的な探索時間で見つけることは，自明な例題を除いては，一般には困難な問題である．入力変数の個数が多くなると最小化を行うことは急速に難しくなる．

ところで，論理式中に現れる個々の二項演算子に着目すると，その演算子が適用されている二つの部分論理式を抽出することができる．それらの部分論理式は，それぞれが一つの論理関数を表現していることになる．したがって，論理式の演算子は，0,1 の論理値の演算手順を表すと同時に，二つの論理関数どうしの論理演算結果が，別の論理関数に対応する，ということも表現している．例えば，論理式 E が，二つの部分論理式 E_1, E_2 を用いて，$E = (E_1) + (E_2)$ というかたちをしていたとすると，E_1 が表す論理関数と E_2 が表す論理関数との論理和を計算したものが，E が表す論理関数になる，ということを示している．つまり，論理代数は，論理値から論理値を求める「論理演算」に関する代数であると同時に，論理関数から論理関数を計算する「論理関数演算」に関する代数ととらえることもできる．

1-2-2 論理式の標準形

上述のとおり，任意の論理関数は論理式により表現できるが，同じ論理関数を表現する論理式は多数存在する．そこで，論理式のかたちに制約を与えて，論理関数と論理式が 1 対 1 に対応するようにしたものを，論理式の標準形と呼ぶ．つまり，標準形の論理式で論理関数を表現・操作することにより，論理関数の等価性判定が容易になる．

以下では，まず積和形の論理式と，その標準形を定義する．

- 1 個のリテラル，または複数個の互いに異なる変数のリテラルの論理積を積項（product term）と呼ぶ．
- 1 個の積項，または複数の異なる積項の論理和を**積和形**（sum-of-products form または disjunctive form）と呼ぶ（定数関数を表すときには，0 または 1 の定数値を書くこととする）．
- n 変数論理関数において，n 個のリテラルからなる積項を**最小項**（minterm）と呼ぶ．
- 最小項だけからなる積和形を**積和標準形**（canonical sum-of-products form）または**最小項展開**（minterm expansion）と呼ぶ．

積和標準形は，論理式の変数順序や積項の順序を適当な辞書順で揃えれば，論理関数に対

して一意に定まる．例えば，1-2-1 節で示した $F(x,y,z)=(\bar{x}+y)\cdot z$ について考えよう．まず，論理関数の真理値表において，出力が 1 となる行を抽出し，それぞれの行に対応する（その入力組合せに対して 1 になる）最小項を列挙する．上記の例では，$(0,0,1)$ が積項 $\bar{x}\cdot\bar{y}\cdot z$ に，$(0,1,1)$ が $\bar{x}\cdot y\cdot z$ に，$(1,1,1)$ が $x\cdot y\cdot z$ に対応する．これらを集めた積和形 $(\bar{x}\cdot\bar{y}\cdot z)+(\bar{x}\cdot y\cdot z)+(x\cdot y\cdot z)$ が積和標準形となる．これが論理関数を一意に表現することは，その構築方法から明らかである．

最小項という名前は，その積項が 1 になる入力組合せが，最小の 1 通りしか存在しないことに由来する．積項の名前は「最小」項であるが，積項に含まれるリテラル数は最大である．最小項だけを集めてつくる積和標準形は，表現は一意であるものの，論理式の長さが n に対して指数的に大きくなるため，n が少し大きくなると実用的ではない．例えば恒真関数は，一般の論理式ではわずか 1 文字で表現可能であるが，積和標準形では $n\cdot 2^n$ 個のリテラルを必要とする．

論理積と論理和の構成を反転させた「和積形」も存在する．以下に定義を記す．論理代数の双対原理により，和積形の各種性質は積和形の場合と全く同様である．

- 1 個のリテラル，または複数個の互いに異なる変数のリテラルの論理和を和項（sum term）と呼ぶ．
- 1 個の和項，または複数の異なる和項の論理積を**和積形**（product-of-sums form または conjunctive form）と呼ぶ（定数関数を表すときには, 0 または 1 の定数値を書くこととする）．
- n 変数論理関数において，n 個のリテラルからなる和項を**最大項**（maxterm）と呼ぶ．
- 最大項だけからなる和積形を**和積標準形**（canonical product-of-sums form）または**最大項展開**（maxterm expansion）と呼ぶ．

1-2-3 シャノン展開

任意の論理関数 $F(x_1,x_2,x_3,\ldots,x_n)$ は，以下のとおり展開できる．

$$F(x_1,x_2,x_3,\ldots,x_n)=\overline{x_1}\cdot F(0,x_2,x_3,\ldots,x_n)+x_1\cdot F(1,x_2,x_3,\ldots,x_n)$$

この展開式は**シャノン展開**（Shannon's expansion）と呼ばれている．実は，Shannon より前に，既に Boole の論文でこの展開式が示されているので，本来はブールの展開と呼ぶべきであるが，一般にはシャノン展開という呼び名が定着している．Shannon は，現在では情報理論の分野でむしろ有名であるが，彼は自身の修士論文で，リレー（継電器）を組み合わせた論理回路の設計に論理代数が利用できることを示し，その後，論理設計の分野で多大な貢献をしたことから，この呼び名が広まったものと思われる．

この展開式が正しいことは，$x_1=0$ のときと $x_1=1$ のときに場合分けして，両辺を比較すれば容易に確かめられる．

シャノン展開は，論理関数の分解や解析を行うための最も基本的な論理式変形法の一つで

ある．シャノン展開を 1 回行うことにより，n 変数論理関数を，二つの $(n-1)$ 変数の部分関数（sub-function）に展開できる*．得られた二つの部分関数それぞれについて，さらに別の入力変数でシャノン展開を行うと，それぞれが二つずつの部分関数に分解できる．この手順を可能な限り繰り返すと，最終的に 0 または 1 の定数関数が得られ，n 段の入れ子状の展開式となる．これを分配則により積和形にして，0 定数関数となった積項を消去すると，得られた論理式は，積和標準形となる．例えば，任意の 3 変数論理関数 $F(x,y,z)$ について考えると，

$$
\begin{aligned}
F(x,y,z) &= \bar{x}\ F(0,y,z) + x\ F(1,y,z) \\
&= \bar{x}\ (\bar{y}\ F(0,0,z) + y\ F(0,1,z)) + x\ (\bar{y}\ F(1,0,z) + y\ F(1,1,z)) \\
&= \bar{x}\ (\bar{y}\ (\bar{z}\ F(0,0,0) + z\ F(0,0,1)) + y\ (\bar{z}\ F(0,1,0) + z\ F(0,1,1))) + \\
&\quad x\ (\bar{y}\ (\bar{z}\ F(1,0,0) + z\ F(1,0,1)) + y\ (\bar{z}\ F(1,1,0) + z\ F(1,1,1))) \\
&= \bar{x}\ \bar{y}\ \bar{z}\ F(0,0,0) + \bar{x}\ \bar{y}\ z\ F(0,0,1) + \bar{x}\ y\ \bar{z}\ F(0,1,0) + \bar{x}\ y\ z\ F(0,1,1) + \\
&\quad x\ \bar{y}\ \bar{z}\ F(1,0,0) + x\ \bar{y}\ z\ F(1,0,1) + x\ y\ \bar{z}\ F(1,1,0) + x\ y\ z\ F(1,1,1)
\end{aligned}
$$

となり，積和標準形が得られることがわかる．

なお，双対原理により，論理積と論理和を交換したシャノン展開も同様に得られる．

$$F(x_1,x_2,x_3,\ldots,x_n) = (\overline{x_1}+F(1,x_2,x_3,\ldots,x_n))\cdot(x_1+F(0,x_2,x_3,\ldots,x_n))$$

この展開式を元に和積標準形が得られることも同様である．

1-2-4 ダビオ展開

シャノン展開は，論理積と論理和を用いた展開であるが，論理和の代わりに排他的論理和を用いると，また別の展開ができる．任意の論理関数 $F(x_1,x_2,x_3,\ldots,x_n)$ は，$F_0 = F(0,x_2,x_3,\ldots,x_n)$, $F_1 = F(1,x_2,x_3,\ldots,x_n)$, $F_2 = F_0 \oplus F_1$ としたとき，以下のとおり展開できる．

- シャノン展開
  $F(x_1,x_2,x_3,\ldots,x_n) = \overline{x_1}\ F_0\ \oplus\ x_1\ F_1$
- **正極性ダビオ展開**（positive Davio's expansion）
  $F(x_1,x_2,x_3,\ldots,x_n) = F_0\ \oplus\ x_1\ F_2$
- **負極性ダビオ展開**（negative Davio's expansion）
  $F(x_1,x_2,x_3,\ldots,x_n) = \overline{x_1}\ F_2\ \oplus\ F_1$

ダビオ展開という名前は，排他的論理和を含む論理式に関して多くの貢献があった M. Davio から取られている．これらの展開式が正しいことは，x_1 に 0,1 をそれぞれ代入して，両辺を

* 定義域の一部だけ定義された関数を部分関数（partial function）と呼ぶこともあるが，ここでは，シャノン展開で得られた二つの部分的な関数を表しており，同じ部分関数でも意味が異なる．

比較することにより，容易に確かめられる．なお，シャノン展開では正リテラルと負リテラルの両方を用いて展開するが，ダビオ展開は正リテラルまたは負リテラルのいずれか一方しか使用しないことに注意してほしい．

シャノン展開と同様に，正極性ダビオ展開を 1 回行うことにより，n 変数論理関数を，二つの $(n-1)$ 変数の部分関数 F_0, F_2 を用いた式に展開できる．更に F_0, F_2 それぞれに対して，別の入力変数に関する正極性ダビオ展開を繰り返し適用していくと，最終的に 0 または 1 の定数関数が得られ，やはり n 段の入れ子状の展開式となる．これを，排他的論理和に関する分配則により括弧を取り除き，定数関数 0 になった積項を消去して整理すると，正リテラルのみの組合せからなる積項（または定数 1）を排他的論理和で結合した論理式が得られる．この論理式は，論理関数を一意に表すことが知られており，**環和標準形**または**リード・マラー標準形**（Reed-Muller canonical form）と呼ばれている．なお，環和標準形は次のようにしても求められる．まず積和標準形を求め，その中に出現する負リテラル $\overline{x_k}$ を $1 \oplus x_k$ にすべて置き換え，分配則などを用いて式を整理すると，環和標準形となる．

以上の議論では，正極性ダビオ展開だけを用いたが，負極性ダビオ展開だけを用いれば，負リテラルのみからなる論理式が同様に得られる．これらを区別するときには，正極性（または負極性）リード・マラー論理式と呼ぶ．

1-2-5 論理演算集合の完全性

論理演算集合の**完全性（万能性）**について述べる．前に示した通り，論理積（AND），論理和（OR），否定（NOT）による論理式は完全であり，この 3 種類の論理演算を組み合わせることにより，任意の論理関数をつくることができる．しかし，OR 演算 $x+y$ は，AND と NOT だけを使って，$\overline{\overline{x} \cdot \overline{y}}$ とすれば作ることができるので，AND と NOT だけでも完全である．更に，否定論理積（NAND）演算さえあれば，NOT は，NAND(x,x) とすればつくることができ，NAND と NOT から AND が作れる．したがって，NAND だけでも完全である．同様に否定論理和（NOR）だけでも完全であることが示せる．

一方，AND と OR の組合せだけで，NOT が使えなければ，表現できない論理関数が存在するため完全ではない．排他的論理和（EXOR）も，それだけでは完全とはならない．一般に，ある演算集合が完全であることを示すのは，例えば NAND 演算がつくれることを示せばよいので比較的容易であるが，完全でないことを示すのは簡単ではない．与えられた演算集合では，ある一定の性質をもつ論理関数の部分集合だけに閉じていることを示す必要がある．この問題に関しては文献 3) で詳しく解説されている．

■参考文献

1) Donald E. Knuth, “Section 7.1: Zeros and Ones,” In “The Art of Computer Programming,” vol.4, Fascicle 0, Addison-Wesley, pp.47-133, 2008.
2) Zvi Kohavi, “Switching and Finite Automata Theory,” second edition, McGraw-Hill, 1978.
3) 笹尾　勤 :“ 論理設計 スイッチング回路理論, ” 第 4 版，近代科学社，2005.

■1 群 - 8 編 - 1 章

1-3 論理関数の諸性質

（執筆者：湊　真一）[2009 年 2 月 受領]

本節では，論理関数の諸性質について述べる．

1-3-1 単調関数とユネイト関数

以下は，前節で示したシャノン展開の式である．

$$F(x_1, x_2, x_3, \ldots, x_n) = \overline{x_1} \cdot F(0, x_2, x_3, \ldots, x_n) + x_1 \cdot F(1, x_2, x_3, \ldots, x_n)$$

シャノン展開で得られる二つの部分関数について，論理関数 F が条件

$$F(0, x_2, x_3, \ldots, x_n) \leq F(1, x_2, x_3, \ldots, x_n)$$

を常に満たすとき，論理関数 F は入力変数 x_1 について**単調増加**（monotone increasing）であるという（ここでは，論理値 1 は 0 より大きいという意味で不等号を用いている）．

F が x_1 について単調増加であれば，シャノン展開の負リテラル $\overline{x_1}$ は不要となり，以下のとおり，正リテラルだけで展開できる．

$$F(x_1, x_2, x_3, \ldots, x_n) = F(0, x_2, x_3, \ldots, x_n) + x_1 \cdot F(1, x_2, x_3, \ldots, x_n)$$

このことは，x_1 に 0,1 をそれぞれ代入してみれば，容易に確かめられる．したがって，この場合は，負リテラル $\overline{x_1}$ を含まない積和形で，F を表現可能であることが分かる．

更に，$F(x_1, x_2, x_3, \ldots, x_n)$ が，すべての入力変数に対して単調増加であるとき，F を**単調増加関数**（monotone increasing function）または**正関数**（positive function）と呼ぶ．なお，定数関数も単調増加関数に含まれる．

単調増加関数は，否定演算を含まずに論理積と論理和だけを用いた論理式で表現できる．例えば，上に示したとおり，正リテラルだけを用いたシャノン展開を，すべての入力変数に対して繰り返し適用すれば，正リテラルのみを組み合わせた積項による積和形をつくることができる．この積和形は明らかに否定演算を含まない．

一方，単調増加と同様に，単調減少な関数も考えることができる．論理関数 F が以下の条件

$$F(0, x_2, x_3, \ldots, x_n) \geq F(1, x_2, x_3, \ldots, x_n)$$

を常に満たすとき，論理関数 F は入力変数 x_1 について**単調減少**（monotone decreasing）であるという．このとき，シャノン展開は負リテラルだけで表現できる．

$$F(x_1, x_2, x_3, \ldots, x_n) = \overline{x_1} \cdot F(0, x_2, x_3, \ldots, x_n) + F(1, x_2, x_3, \ldots, x_n)$$

したがって，このときは，正リテラル x_1 を含まない積和形で，F を表現可能である．

論理関数 $F(x_1, x_2, x_3, \ldots, x_n)$ が，すべての入力変数に対して単調減少であるとき，F

を**単調減少関数**（monotone decreasing function）または**負関数**（negative function）と呼ぶ．なお，定数関数も単調減少関数に含まれる．単調減少関数は，負リテラルのみを含む積和形で表現可能である．

単調増加関数と単調減少関数を総称して**単調関数**（monotone function）と呼ぶ．単調関数を更に一般化した概念として，ユネイト関数が定義されている．論理関数 $F(x_1, x_2, x_3, \ldots, x_n)$ が，それぞれの入力変数に対して，単調増加または単調減少のいずれかであるとき，F は**ユネイト**（unate）であるという．定数関数もユネイト関数に含まれる．単調関数は，すべての入力変数について単調増加または単調減少のいずれかに揃っていなければならないが，ユネイト関数では，入力変数によって単調増加と単調減少が混在していても構わない．ユネイト関数は，それぞれの入力変数が正負いずれか片方のリテラルしか出現しない積和形で表現できる．

1-3-2 双対関数と自己双対関数

論理関数 $F(x_1, x_2, \ldots, x_n)$ のすべての入力と出力に否定演算を加えてできる関数 $\overline{F(\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n})}$ を，F の**双対関数**（dual function）と呼ぶ．

例えば，$(x \cdot y) + (\bar{x} \cdot z)$ の双対関数は，$\overline{(\overline{x} \cdot \overline{y}) + (x \cdot \overline{z})} = (x + y) \cdot (\bar{x} + z)$ となる．この例から分かるように，論理式が，論理積，論理和，否定及び括弧からなる場合，括弧を省略しないようにした後，論理和を論理積に，論理積を論理和に，それぞれ置き換えたものは双対関数となる（定数 0 があれば 1 に，定数 1 があれば 0 に置き換える）．

さて，ここで $(x \cdot y) + (y \cdot z) + (z \cdot x)$ を考えると，その双対関数は，$(x+y) \cdot (y+z) \cdot (z+x) = (x \cdot y) + (y \cdot z) + (z \cdot x)$ となり，元の論理関数と一致する．このように，論理関数 F の双対関数が自分自身と一致する場合に，F は**自己双対関数**（self-dual function）であるという．

自己双対関数がどのような論理関数かを知るために，3 変数論理関数 $F(x, y, z)$ の双対関数の真理値表を考えてみよう．

x	y	z	$F(x,y,z)$	$F(\bar{x},\bar{y},\bar{z})$	$\overline{F(\overline{x},\overline{y},\overline{z})}$
0	0	0	a_{000}	a_{111}	$\overline{a_{111}}$
0	0	1	a_{001}	a_{110}	$\overline{a_{110}}$
0	1	0	a_{010}	a_{101}	$\overline{a_{101}}$
0	1	1	a_{011}	a_{100}	$\overline{a_{100}}$
1	0	0	a_{100}	a_{011}	$\overline{a_{011}}$
1	0	1	a_{101}	a_{010}	$\overline{a_{010}}$
1	1	0	a_{110}	a_{001}	$\overline{a_{001}}$
1	1	1	a_{111}	a_{000}	$\overline{a_{000}}$

ここで，$a_{000}, \ldots, a_{111}$ には，それぞれ 0 または 1 が入る．この真理値表を見て分かるとおり，$F(x, y, z)$ と $F(\bar{x}, \bar{y}, \bar{z})$ は，中央の線を境に上下対称の関係にある．自己双対関数であるためには，$F(x, y, z)$ と $\overline{F(\overline{x}, \overline{y}, \overline{z})}$ が等価でなければいけないので，上半分の $a_{000}, \ldots, a_{011}$ の値は自由に決められるが，それに応じて，下半分の $a_{100}, \ldots, a_{111}$ の値も，ちょうどその否定の値として一意に決まってしまう．そのようなつくり方をすれば，任意の自己双対関数をつくれることが分かる．

1-3-3 線形関数とパリティ関数

環和標準形にしたときに 2 次以上の積項を含まない論理関数，すなわち，

$$F = a_0 \oplus a_1\ x_1 \oplus a_2\ x_2 \oplus \cdots \oplus a_n\ x_n$$

（ただし, $a_i \in \{0,1\}, (i = 0,1,2,\ldots,n)$）というかたちで表現できる論理関数 F を**線形関数**（linear function）と呼ぶ．定数関数も線形関数に含まれる．

更に，線形関数の中でも，$a_0 = 0$ でそれ以外の係数がすべて 1 のとき，すなわち，

$$F = x_1 \oplus x_2 \oplus \cdots \oplus x_n$$

というかたちで表現される n 変数論理関数を，**パリティ関数**（parity function）と呼ぶ．パリティ関数は，1 となっている入力変数が奇数個のときに 1 を出力し，偶数個のときに 0 を出力する．実用的によく使われる論理関数である．

1-3-4 対称関数

上記で述べたパリティ関数は，入力変数の中の 1 の個数が奇数か偶数かによって出力が決まるので，入力変数の順序を入れ替えても出力には全く影響がない．同様に，論理積や論理和を計算する論理関数も，変数の順序による影響を受けない．このような論理関数のことを**対称関数**（symmetric function）と呼ぶ．

対称関数は，入力変数の 1 の個数によってのみ出力が定まる．入力変数の総数が n 個のとき，1 となっている入力変数の個数は，0 個以上 n 個以下であるから，その個数について $(n+1)$ 通りの場合分けを行い，それぞれについて出力が 0 か 1 かを決めれば，任意の n 変数対称関数を定義できる．例えば，$F(x,y,z) = \bar{x}\ \bar{y}\ \bar{z} + x\ y + y\ z + z\ x$ は対称関数であるが，これは入力変数の 1 の個数が {0,2,3} のいずれかのときに出力が 1 となり，それ以外のときには 0 となる論理関数である．これを $S_{0,2,3}(x,y,z)$ と書くこともある．

共通の n 個の入力変数をもつ二つの対称関数 F, G の論理和 $F + G$ は，やはり対称関数となる．これは，入力変数の 1 の個数が決まれば，F, G の値がそれぞれ一意に決まり，その結果 $F + G$ の結果も一意に決まることから明らかである．同様に，対称関数の論理積 $F \cdot G$ や，否定 $\overline{F}$ もまた対称関数となることが分かる．

ここまでは n 個の変数がすべて対称な場合について述べたが，n 個の変数の一部のみが対称である論理関数も考えられる．これを**部分対称関数**（partially symmetric function）と呼ぶ．例えば，n 変数論理関数 F において，F の論理式の x_1 と x_2 を互いに交換しても，交換前と等価であるとき，F は変数 x_1, x_2 に関して対称であるという．この関係は推移律が成り立つ．例えば，x_1, x_2 に関して対称で，x_2, x_3 に関しても対称であれば，x_1, x_3 に関しても必ず対称である．現実の論理回路では，このような部分的に対称な変数グループをもつ論理関数もしばしば見られる．

部分対称関数の性質を知るために，論理関数 $F(x_1, x_2, x_3, \ldots, x_n)$ に対して，x_1, x_2 に関する 2 段階のシャノン展開を行ってみよう．ここで $F_{00} = F(0,0,x_3,\ldots,x_n)$, $F_{01} = F(0,1,x_3,\ldots,x_n)$, $F_{10} = F(1,0,x_3,\ldots,x_n)$, $F_{11} = F(1,1,x_3,\ldots,x_n)$ とすると，

$$F(x_1,x_2,x_3,\ldots,x_n) = \overline{x_1}\ \overline{x_2}\ F_{00} + \overline{x_1}\ x_2\ F_{01} + x_1\ \overline{x_2}\ F_{10} + x_1\ x_2\ F_{11}$$

と展開できる．一方，x_1 と x_2 を交換した論理関数は，

$$F(x_2, x_1, x_3, \ldots, x_n) = \overline{x_1}\ \overline{x_2}\ F_{00} + \overline{x_1}\ x_2\ F_{10} + x_1\ \overline{x_2}\ F_{01} + x_1\ x_2\ F_{11}$$

と展開できる．結局，F_{00} と F_{11} は，変数を交換しても影響がないので，F_{01} と F_{10} が等価であるかどうかを調べれば，x_1 と x_2 に関して対称かどうかが判定できる．

もしも F が完全な対称関数であれば，この関係がすべての変数の組合せについて成り立つことになる．

1-3-5 多数決関数としきい関数

対称関数の中でも重要な性質をもつ関数として多数決関数がある．**多数決関数**（majority function）とは，奇数個の入力変数をもつ対称関数であって，1 の個数が過半数のときに 1 を出力し，0 が過半数のときに 0 を出力する関数である．例えば，3 変数多数決関数は $x\ y + y\ z + z\ x$，または $S_{2,3}(x, y, z)$ と表せる．一般に，$(2m+1)$ 変数の多数決関数（m は任意の自然数）は，

$$S_{m+1,m+2,\ldots,2m+1}(x_1, x_2, \ldots, x_{2m+1})$$

と表せる．更に多数決関数は，単調増加関数であり，自己双対関数であることが知られている．

多数決関数を一般化したものとして，しきい関数がある．**しきい関数**（threshold function）とは，任意の実数値の**しきい値**（threshold）T と，任意の実数値の**重み**（weight）ベクトル $(w_1, w_2, \ldots, w_n)$ が与えられたときに，

$$F(x_1, x_2, \ldots, x_n) = \begin{cases} 1 & \sum_{i=1}^{n} w_i\ x_i\ \geq\ T \\ 0 & \text{上記以外} \end{cases}$$

で定義される論理関数である．しきい値や重みは負の値でも構わない．

例えば，$n = 3, T = 0.5, w_1 = -1, w_2 = 2, w_3 = 1$ のときは，

$$F(x_1, x_2, x_3) = \begin{cases} 1 & -x_1 + 2x_2 + x_3 \geq 0.5 \\ 0 & \text{上記以外} \end{cases}$$

となり，この論理関数は，論理式 $\overline{x_1}\ x_3 + x_2$ と等価である．なお上記の重み付き加算の演算子 $+$ は，論理和ではなく実数値の算術和を表すことに注意．

しきい関数において，$n = 2m+1, T = m+1, w_i = 1 (i = 1, 2, \ldots, n)$ とすれば多数決関数をつくることができる．しきい関数は，それぞれの入力変数について，重みが正ならば単調増加，負なら単調減少であり，全体としては常にユネイト関数となる．

しきい関数は，入力と出力はそれぞれ 0,1 の論理関数であるが，関数を定義する式は連続的な実数値の線形和の大小比較であり，シンプルで応用範囲が広い．

1-3-6 論理関数の個数と NPN-同値類

これまで述べたように，様々な性質をもつ論理関数が考えられるが，論理関数の個数がど

のくらいあるかを考えてみよう．n 変数の論理関数を一意に表す真理値表は，2^n の行数をもち，その各行に 0 または 1 が入る．この 2^n 個の 0,1 の組合せが 1 か所でも違えば，異なる論理関数となるから，n 変数論理関数の総数は，2^{2^n} 個存在することが分かる．以下に示すとおり，2^{2^n} は n に対して猛烈な勢いで増大するので，n が少し大きくなると，n 変数論理関数をすべて列挙して処理するということは，すぐに困難になる．

n	0	1	2	3	4	5	6
2^{2^n}	2	4	16	256	65536	4294967296	18446744073709551616

一般に，与えられた論理関数に対して，入力変数の順序の交換や，入力や出力の否定演算を行うことにより，別の論理関数を生成できる場合がある．このような操作により，論理関数どうしの同値関係を構成することができ，論理関数の同値類が得られる．以下の三つの操作を組み合わせて得られる同値類のことを **NPN-同値類**（NPN-equivalent class）または **NPN-同値関数**（NPN-equivalent function）という．

(1) 一部またはすべての入力変数の否定（Negation）

(2) 一部またはすべての入力変数の順序の変更（Permutation）

(3) 出力結果の否定（Negation）

このうち，(1)(2) だけを用いて得られる同値類を NP-同値類，(2) だけを用いた場合を P-同値類，(1) だけを用いた場合を N-同値類と呼ぶことがある．例えば，論理関数 $f(x,y) = x+y$ が属する NPN-同値類には，$\bar{x}+y$，$x+\bar{y}$，$\bar{x}+\bar{y}$，$x\cdot y$，$\bar{x}\cdot y$，$x\cdot\bar{y}$，$\bar{x}\cdot\bar{y}$ が含まれる．これら八つの論理関数は，上記の (1)(2)(3) の操作の組合せにより，同一の論理関数に互いに変換可能である．現実の論理回路では，論理の否定や変数の交換は容易に実装できることが多いため，同じ同値類に属する論理関数の中から一つ代表を選んで論理回路を設計しておけば，ほかの論理関数にも再利用が可能となる．

n 変数論理関数の NPN-同値類の個数を求めることは，それほど簡単ではない．論理関数の対称性や自己双対性などによって，同値類の個数が影響を受けるからである．文献[3]にはこれらの同値類の個数を $n = 6$ まで調べた結果が記されている．4 までの n については以下の結果が知られている．

n	0	1	2	3	4
関数の総数	2	4	16	256	65536
P-同値類	2	4	12	80	3984
NP-同値類	2	3	6	22	402
NPN-同値類	1	2	4	14	222

1-3-7 不完全指定論理関数

n 変数論理関数の入力値の 0,1 の組合せは 2^n 通りある．通常の論理関数は，すべての入

力組合せに対する出力結果が定義されているが，現実の論理回路においては，必ずしもすべての組合せが入力されるとは限らず，一部の入力値に対しては，出力結果は未定義でも構わないことがしばしばある．例えば，$F(x,y,z) = \bar{x}\ y + x\ \bar{y}\ \bar{z}$ という 3 変数論理関数を考えてみよう．この関数の入力の組合せは全部で 8 通りあるが，もしも，ある設計条件から x と y は同時に 1 になることがないと分かっていたとすると，そのようなあり得ない入力に対して論理関数の出力を定義する必要はない．その場合，以下の真理値表に示すとおり，6 通りの入力に対する出力だけを定義し，2 通りの入力については未定義（d と記入）と書いても構わない．

x	y	z	$\bar{x}\ y + x\ \bar{y}\ \bar{z}$	(不完全指定)	$y + x\ \bar{z}$
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	1	1	1
0	1	1	1	1	1
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	0	d	1
1	1	1	0	d	1

このように一部だけ定義された論理関数のことを**不完全指定論理関数**（incompletely specified switching function）と呼ぶ．不完全指定でない通常の論理関数のことを特に**完全指定**（completely specified）論理関数と呼ぶことがある．また，真理値表に記入された d という値のことをドントケア（don't care）と呼ぶ．ドントケアは 0 でも 1 でもどちらでも構わないという意味である．

不完全指定論理関数では，ドントケアを都合の良い値に解釈することにより，論理式を簡単化できる場合がある．上記の例では，ドントケアを 1 とおくことにより，$F' = y + x\ \bar{z}$ という，より簡単な論理式を用いても，仕様を満たすことが分かる．つまり，一つの不完全指定論理関数は，それが許容する複数の完全指定論理関数の集合を表現している．

1-3-8 多出力論理関数

共通の入力変数をもつ複数の論理関数をまとめて一つの関数と考えたものを**多出力論理関数**（multi-output switching function）と呼ぶ．例えば以下の真理値表は，F_1, F_2 の二つの 2 変数論理関数を表現しているが，これをまとめて見れば 2 入力 2 出力の多出力論理関数となる．

x	y	F_1	F_2
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

実は，この関数は，x, y に含まれる 1 の個数を 2 桁の 2 進数で表したものになっている．一般に，m 個の論理値の列により，m ビット幅の 2 進数を表すことができることから，m 出力の多出力論理関数は，2^m 通りの値を取りうる多値の 1 出力論理関数とみなすこともできる．

1-3-9 多値論理関数

ここまで 0,1 の 2 値の論理関数について述べてきたが，現実の世界では 3 値以上の多値の事柄を扱うことも少なくない．このような多値を扱う論理関数を**多値論理関数**（multi-valued logic function）と呼ぶ．

多値論理を扱う場合，複数の 2 値変数を組み合わせて多値を表現することが可能である．大きく分けて，2 進符号化（binary coding）とワン・ホット符号化（one-hot coding）の二つの方法がある．2 進符号化は k 個の 2 値変数を並べて，その 2 進数により最大 2^k 値の多値を表す方法である．一方，ワン・ホット符号化は k 個の 2 値変数のどれか一つだけが 1 になるという符号により，最大 k 値の多値論理を表現する．以下に 4 値論理を表現した例を示す．

多値論理	2 進符号化		ワン・ホット符号化			
	x_1	x_0	s_0	s_1	s_2	s_3
0	0	0	1	0	0	0
1	0	1	0	1	0	0
2	1	0	0	0	1	0
3	1	1	0	0	0	1

2 進符号化の方が変数の個数が少なくて済むが，符号化により論理式が複雑になる欠点がある．ワン・ホット符号化は，2 進符号化よりも論理式が単純で分かりやすいが，値の数が大きくなると効率が悪い．

多値論理を符号化して 2 値で扱うと，2 値の論理関数に関する様々な技術が適用できるというメリットがあるが，符号化の分だけ処理が複雑になったり効率が低下したりする．多値を多値のまま扱う手法も有効な場合があり，研究が行われている．

多値論理関数のより詳しい性質については，文献 4) に書かれている．

■参考文献

1) Donald E. Knuth, “Section 7.1: Zeros and Ones,” In “The Art of Computer Programming,” vol.4, Fascicle 0, Addison-Wesley, pp.47-133 (2008)
2) Zvi Kohavi, “Switching and Finite Automata Theory,” second edition, McGraw-Hill, 1978.
3) 室賀三郎／笹尾勤訳, “論理設計とスイッチング理論 - LSI,VLSI の設計基礎,” 共立出版, 1979.
4) 笹尾　勤, “論理設計 スイッチング回路理論,” 第 4 版, 近代科学社, 2005.

■1 群 - 8 編 - 1 章

1-4 論理関数の計算機上での処理

（執筆者：湊　真一）[2009 年 2 月 受領]

前節では，種々の論理関数の数学的性質について述べた．この節では，論理関数を実際に計算機上で表現する方法（データ構造）について述べる．まず，計算機上での論理関数表現に求められる性質について説明し，いくつかの代表的な論理関数表現法とその特徴について解説する．中でも近年盛んに用いられている二分決定グラフ（BDD）については，やや詳しく述べる．

1-4-1 論理関数の表現とは

論理回路を設計する際には，実現しようとする機能（仕様）を何らかの方法で表現し，それに基づいて，効率の良い論理回路を間違いなくつくる必要がある．論理関数の演算処理を効率良く行うことは，限られた設計期間で論理設計を行うための重要な技術である．論理関数の入力変数が 4〜5 個までであれば，紙と鉛筆を使って人手で設計することも可能だが，それより少し大きな規模の論理関数になると，手間がかかりすぎる上に，人間はしばしば誤りを犯す．そこで，計算機上で論理関数を表現し，高速に誤りなく演算処理を行う方法が種々提案され，利用されている．このような論理関数表現は，論理設計だけに限らず，種々の制約充足問題，機械学習，データマイニングなど，計算機分野の様々な問題で用いられる基盤技術である．

まず，計算機上での論理関数表現に求められる要求条件を考えよう．

(1) 論理関数の表現がコンパクトである．

(2) 任意の論理式から，その論理関数表現を高速に生成することができる．

(3) 論理関数の表現に一意性がある．または恒真性判定が高速に行える．

(4) 論理関数を充足する（出力が 1 になる）ような入力組合せを高速に求めることができる．

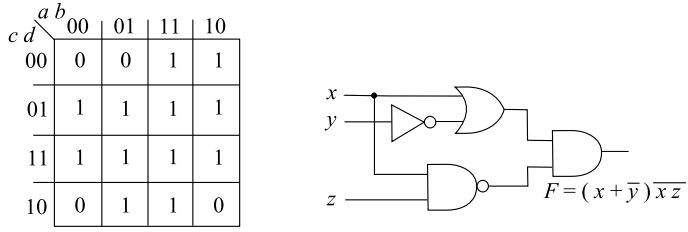
(1) の条件は，当然のことである．同じ論理関数を表現するのにデータ量が少ない方が良い．計算機の主記憶サイズは限られているし，ほとんどの演算処理の計算時間はデータ量に依存するからである．ただし，前節で述べたとおり，n 変数論理関数の総数は 2^{2^n} 個あるため，それらを固定長データで識別するためには，少なくとも 2^n ビットは必要である．これは情報理論的に明らかである．しかしながら，2^{2^n} 個のすべての論理関数が均等に現れるわけではないので，実用的によく使われる論理関数がコンパクトに表現できるような，可変長データが好ましい．

(2) は，別の言い方をすると，論理関数の否定や，二つの論理関数どうしの論理積や論理和の結果に対応する論理関数表現を高速に生成できる，ということである．これができれば，任意の論理式や論理回路から，それらが計算しようとしている論理関数を効率良く生成することができる．

(3)(4) は，論理関数表現を生成した後に，何を調べたいかということである．二つの論理式の等価性判定や包含性判定は，現実の問題でしばしば必要となる．F と G の等価性判定は

$F\ G + \overline{F}\ \overline{G}$ が恒真であるかどうかを調べればよい．包含性判定（F ならば G）は $\overline{F} + G$ の恒真性判定に帰着できる．したがって，論理関数の恒真性判定が高速に行えることが多くの検証問題の基本処理となる．このとき，論理関数に対して表現の一意性があれば等価性判定も恒真性判定も極めて簡単になるので，表現の一意性は非常に好ましい性質である．そして (4) の充足解探索は，二つの論理式が等価でないときに反例をみつけたい，というときに必要な処理である．

計算量理論においては，(3) は co-NP 完全問題，(4) は NP 完全問題であることが知られており，どちらも，普通の計算機では入力データ長に対して指数関数的な計算時間がかかる方法以外は見つかっておらず，今後も見つからないと予想されている．しかしながら，最悪の場合に指数時間かかることは避けられないとしても，現実によく現れる比較的単純な論理関数に対しては高速に処理できる，という方法は実現可能であり，そのようなデータ構造が好ましい．



(a) **カルノー図**　　(b) **論理回路図**

図 1·1　論理関数の図的表現

なお，人手で扱えるような小規模な論理関数を表現する方法としては，図 1·1 のような**カルノー図**（Karnaugh map）*や**論理回路図**といった，人間の目で見たときに見通しが良い図的な表現がよく使われていた．カルノー図は，真理値表を二次元のマス目に並べた図であるが，入力の組合せを反射 2 進の順序（00, 01, 11, 10）で配置することによって，互いに類似する入力組合せが隣接するように工夫したものである．

しかし，大規模な論理関数を計算機上で扱う場合には，人間が見て理解しやすいかどうかはあまり重要ではなく，記憶量と処理速度がより重要である．計算機上での論理関数の表現方法としては，現在までに主なものとして

- **真理値表**（truth table）
- **積和形論理式**（sum-of-products form）
- **二分決定グラフ**（BDD, Binary Decision Diagram）

が知られている．以下，それぞれの方法について述べる．

* ここに示したカルノー図は典型的な形式であるが，ほかにもいくつかの流儀がある．

1-4-2 真理値表

真理値表は，以前の節で述べたように，2^n 通りのすべての入力組合せに対する論理関数の出力を列挙したものである．図 1･2 に例を示す．真理値表とカルノー図は，本質的に同じものであるが，計算機上で扱う場合は見やすく並べる必要はなく，単純な一次元配列として実装すればよい．この図では，便宜上入力変数の組合せも書いてあるが，変数の個数と順序が固定されていて，2 進数の順序で並べることにすれば，出力を表すビット列だけで表現できる．つまり一つの n 変数論理関数を表すのに，ちょうど 2^n ビットを要する．

$x_n \cdots x_3 x_2 x_1$	F_1	F_2	F_3
0 ⋯0 0 0	1	0	1
0 ⋯0 0 1	0	1	0
0 ⋯0 0 0	1	1	0
0 ⋯0 1 1	0	0	0
0 ⋯1 0 0	1	1	0
0 ⋯1 0 1	0	1	0
0 ⋯1 1 0	1	1	0
⋮	⋮	⋮	⋮
1 ⋯1 1 1	1	0	0

図 1･2　真理値表

真理値表による表現は，どんな簡単な論理関数に対しても，n に関して指数関数的な記憶量と処理時間を必要とするという欠点がある．逆に，どんな複雑な論理関数でも一定の処理時間で扱えるという特長もある．真理値表は，n が 20〜30 程度までならば，現実的に使用可能である．また，データ構造が極めて単純なので，ベクトル計算機や並列計算機に向いている．

1-4-3 積和形論理式

積和形論理式は，以前の節で述べたように，正または負のリテラルの組合せからなる積項の論理和により，論理関数を表現する方法である．これは，論理関数の出力を 1 にするための入力変数の条件を列挙する形式とみなすこともでき，人間の直感に合い，比較的コンパクトな表現である．積和形以外の一般の論理式も，論理関数の表現であることには違いないが，自由度が大きすぎて，恒真判定や論理演算などの処理がしにくい．積和形は論理式のかたちに一定の制約を加えて，計算機上で処理しやすくしたデータ構造といえる．

積和形論理式は，可変長データであるという点で，真理値表とは大きく異なる．例えば，$a, b, c, \ldots, z$ の 26 変数の場合を考えると，真理値表では恒真関数を表現するのにも $2^{26}(=67108864)$ ビットの記憶量を必要とするが，積和形論理式では，「1」という 1 文字で済む．また，26 変数の論理積や論理和を表す論理関数も，真理値表では同じく 2^{26} ビット使用するのに対し，積和形論理式では 26 文字（と演算子）だけで表現できる．このように，単純な論理関数はコンパクトに表現できるという特長がある．

積和形論理式を計算機上で表現する方法はいくつか考えられる．論理式をそのまま文字列として格納することも可能である．文字列形式は人間が読み書きしやすいが，計算機での処理は面倒である．

大規模な積和形論理式を効率よく扱う方法として，図 1･3 のように，各積項をビット列で表し，積項データの直列リストにより積和形を表現するデータ構造がしばしば用いられる．この表現方法では，使用する変数（リテラル）の個数はあらかじめ固定する．積項数は論理計算の途中で増減するので，使用記憶量は動的に変化する．

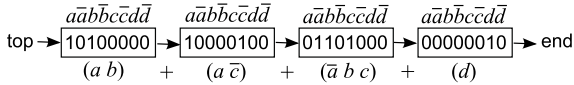


図 1･3　積和形論理式のリスト表現の例

積和形論理式は，変数順序と積項の並べ方を決めておけば，積項の集合を一意に表現できる．したがって二つの積和形論理式が同一かどうかは，先頭から順に比較していけば容易に判定できる．ただし，異なる積和形論理式が同じ論理関数を表現する場合があるので，注意が必要である，二つの積和形論理式の等価性判定や恒真判定は，最悪の場合，式の長さに対して指数時間かかる可能性がある．

二つの積和形どうしの論理和を表す積和形を求める処理は，式の長さの和に比例する計算時間で容易に実行できる．しかし二つの積和形どうしの論理積や排他的論理和は，最悪の場合，式の長さの積に比例する計算時間を要する．更に積和形論理式は否定演算に弱いという欠点があり，最悪の場合，否定演算の結果が，元の式の長さに対して指数的に増大する．

更に，これらの論理演算処理の過程で，冗長な積項が発生することがしばしばある．いったん，冗長項が発生すると，冗長項どうしの演算により更に冗長項が増えるため，演算中に冗長項を除去する必要がある．完全に除去するには，積和形論理式の最小化・簡単化アルゴリズムを適用するが，これは式の長さの指数またはそれ以上の計算時間がかかる．そこで，簡単に見つかる冗長項だけを，比較的高速に（式の長さの 2 乗程度で）除去する方法を用いることもある．

積和形論理式の処理時間は，総リテラル数（各積項のリテラル数の総和）にほぼ比例する．総リテラル数は論理関数の複雑さに依存し，最悪の場合は入力変数の個数 n に対して指数関数的に増大する．また，総リテラル数は処理中に動的に増減するため，たとえ最終結果は小さくても，計算途中で爆発的に増大する場合もある．経験的には，総リテラル数が数十から数百であれば，たいていの処理はほとんど待たずに終了する．総リテラル数が 1000 を超えると，処理内容によっては数時間以上かかる場合がある．総リテラル数が 100000 を超えると，現実的な時間で処理が完了する望みは薄いので，問題をもっと単純化するなど，解き方を根本的に考え直した方がよい．

1-4-4 BDD（二分決定グラフ）

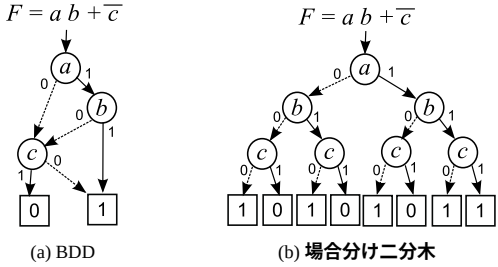
論理関数のデータ表現に **BDD（二分決定グラフ）**（Binary Decision Diagram）を用いる方法は，記憶効率や処理速度の面で優れており，1990 年頃から，論理回路設計をはじめとする様々な分野で広く使われるようになっている．

BDD は，図 1･4(a) に示すような論理関数のグラフによる表現である．これは，それぞれの入力変数に 0,1 の値を入れて場合分けしたときの論理関数の出力（すなわち，シャノン展

開をすべての変数に対して繰り返し適用して得られる入れ子構造）を，図 1･4(b) のように場合分け二分木で表現し，更にこれを縮約することにより得られる．このとき，場合分けする変数の順序を固定し，

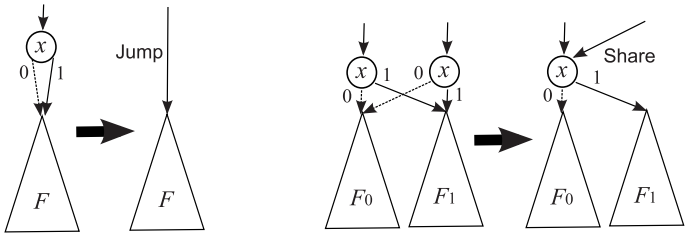
- 冗長な節点を削除する（図 1･5(a)）．
- 等価な部分グラフを共有する（図 1･5(b)）．

という処理を可能な限り行うことにより「既約」なかたちが得られ，論理関数をコンパクトかつ一意に表現できることが知られている．このような既約な順序つき BDD のことを Reduced Ordered BDD（ROBDD）と呼ぶ．多くの場合，単に BDD というときはこの ROBDD のことを指す．BDD は，真理値表や積和形論理式と比較して，多くの優れた性質をもっている．BDD の記憶量は，論理関数の性質に依存し，最悪の場合は入力変数の個数に対して指数関数的なサイズとなるが，人間が設計するような論理式や論理回路で表現された論理関数に対しては，多くの場合，非常にコンパクトに圧縮されたグラフになる．それに加えて BDD は論理関数を一意に表せるという大きな特長がある．



(a) BDD　　(b) **場合分け二分木**

図 1･4　BDD **と場合分け二分木**



(a) **冗長節点の削除**　　(b) **等価節点の共有**

図 1･5　BDD **の簡約化規則**

図 1･6 に代表的な論理関数の BDD を示す．このように，n 入力 AND, OR, EXOR の論理関数（更に一部の入力や出力に否定がついたもの）は，n に比例する節点数で表現可能である．

BDD は，積和形が苦手としていたパリティ関数や対称関数もコンパクトに表現でき，データが計算機の主記憶に納まる限りは，各種の論理演算をその記憶量にほぼ比例する時間で効率よく実行できるという特長をもつ．論理関数の性質にもよるが，数十から数百個もの入力変数をもつ論理関数を，汎用 PC の主記憶容量（1GB 程度）の範囲で現実的に表現し，種々の論理演算を実行することができる．

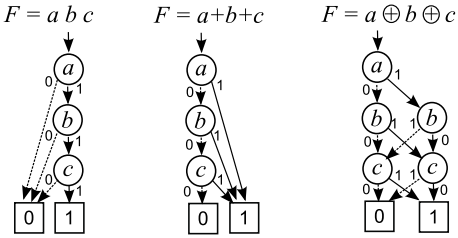


図 1･6　代表的な論理関数の BDD

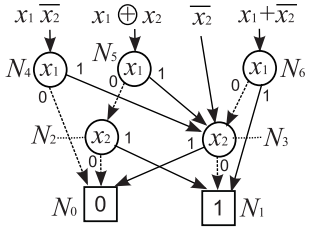
1-4-5 BDD の演算処理アルゴリズム

ある論理関数を表す BDD を構築する際に，上記で述べたように二分木を縮約してつくる方法では，二分木のサイズが真理値表と同様に，入力変数の個数 n に対して常に指数サイズになってしまうので，BDD をつくる利点が失われてしまう．そこで，二つの既約な BDD を入力とし，それらの二項論理演算の結果を表す既約な BDD を直接生成するアルゴリズムが Bryant[2)] により提案された．この方法が提案されてから，任意の論理式の論理関数を表す BDD を効率よく生成できるようになり，BDD が広く実用的に使われるようになった．

論理式に対応する BDD をつくるには，まず 1 変数の論理関数に対応する 1 節点の BDD をつくっておき，それらの間で，論理式の構造にしたがって，二項論理演算を適用してその結果の BDD を構築していけばよい．例えば $F = a\ b + c$ という論理式に対しては，まず a, b, c それぞれの BDD をつくっておき，次に a と b の BDD どうしの論理積演算により $G = a\ b$ を表す BDD をつくり，続いて G と c の BDD どうしの論理和演算により F の BDD が得られる．

このように，ある論理式に対応する BDD を生成する過程で，いくつかの途中結果の BDD が生成される．これらの BDD の間でも互いに部分グラフを共有することができる．実際の BDD 処理系では，一つの大きなメモリ空間を用意して，その中で，処理系に現れるすべての BDD を共有して表現する方法が用いられている．図 1･7 に，共有化された複数の BDD のかたちと，それらを主記憶上の一つのテーブルにまとめて表現した例を示す．

BDD どうしの二項論理演算アルゴリズムは，BDD 処理技法の中で最も重要な部分である．詳細については，文献 2), 7), 9) などに書かれているが基本的な考え方は以下のとおりである．



(節点 ID)	変数	0-枝	1-枝	(論理関数)
N_0	−	−	−	$\leftarrow \boxed{0}$
N_1	−	−	−	$\leftarrow \boxed{1}$
N_2	x_2	N_0	N_1	
N_3	x_2	N_1	N_0	$\leftarrow \overline{x_2}$
N_4	x_1	N_0	N_3	$\leftarrow x_1\ \overline{x_2}$
N_5	x_1	N_2	N_3	$\leftarrow x_1 \oplus x_2$
N_6	x_1	N_3	N_1	$\leftarrow x_1 + \overline{x_2}$

図 1·7　複数の BDD の共有化

(1) 論理演算 $f \circ g$ を，最上位の変数 x について場合分けして，二つの部分グラフの組 $f_{(x=0)} \circ g_{(x=0)}$ と $f_{(x=1)} \circ g_{(x=1)}$ に分解する．

(2) 上記の分解を再帰的に繰り返して，最終的に自明な演算になったところで，部分問題の演算結果を求める．

(3) 得られた部分問題の結果を再度組み上げることにより，新しい BDD を構築する．

この方法を単純に実行すると，二分木状の再帰的な手続き呼出しが必要になり，入力変数の個数に対して指数回の計算ステップが必要になる．しかし，

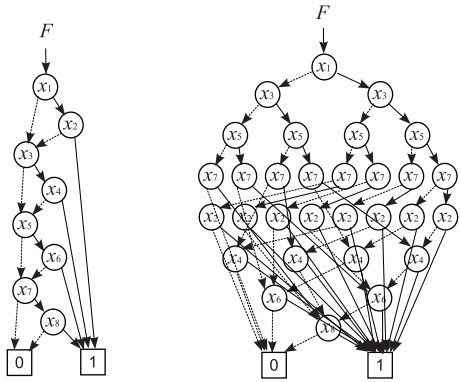
- 途中で生成したすべての BDD 節点を共通のハッシュテーブルに登録しておき，これをチェックすることにより，等価な節点は重複して生成しないようにする．
- 途中の演算結果を記録しておく「演算キャッシュ」と呼ぶテーブルを用意し，同じ演算が過去に行われていれば，再帰呼出しをせずに即座に結果を返す．

という二つの高速化技法を用いることにより，演算対象の BDD の節点数にほぼ比例する時間で論理演算を実行できる．つまり，BDD がコンパクトになるような論理関数に対しては，非常に高速に BDD を生成できることになる．

1-4-6 BDD の変数順序付け

一般に BDD は，入力変数を展開する順序によって同じ論理関数でも異なるかたちとなり，その節点数も変化する．順序付けによる影響の大きさは論理関数の性質に依存し，対称関数では全く変化しないが，大きく変動する場合もあり，ときには数百倍もの差が生ずることがある．例えば図 1·8 のように，変数順序の変更による節点数の増減が，変数の個数に関して指数関数的になる例も存在する．

論理式や論理回路から BDD を生成する際に，変数順が不適当だと節点数が爆発的に増大し，途中で記憶あふれを起こして BDD を生成できない場合もあるため，変数順序付けは重要な問題である．任意の論理関数に対して節点数最小の変数順を求める問題は，NP 完全であることが知られている[16]．論理関数にもよるが，変数の個数が 20〜30 程度までが，最適な順序を求められる限界である[4]．



$$F = x_1\,x_2 + x_3\,x_4 + x_5\,x_6 + x_7\,x_8$$

図 1·8　変数順序付けの影響

実用的な観点からは，厳密に最適でなくても，比較的良い順序が得られれば十分であることも多い．例えば，論理式や論理回路から BDD を生成する場合に，論理式や回路の構造をたどりながら，発見的手法により変数順序付けを行う方法がいくつか提案されている[5, 11]．この方法は論理式や回路の構造に依存し万能ではないが，実験的には，多くの場合，無作為な順序に比べ相当に良い結果が得られることが知られている．

一方，ある変数順序で生成された BDD について，その変数順序を並べ替えることにより節点数を削減する方法も研究されている[6, 8]．隣接する 2 変数の順序交換は，その 2 変数に関する節点以外には影響を与えないため，比較的高速に実行できる．これを繰り返すことにより，任意の順序交換を実現できる．一般論として，BDD 構築後の変数順序改善法は，BDD 構築前の順序付けよりも最適化能力が高く，最終的には，よりコンパクトな BDD を得られることが多い．しかし，最初に適当な初期順序で BDD を構築できなければ適用できないという欠点がある．また，BDD が大きくなると，変数交換にかかる時間も無視できないため，最初から比較的良い順序で BDD を構築することが望ましい．

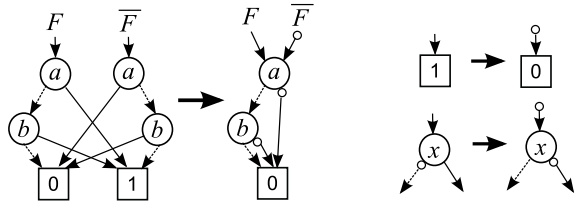
複雑な構造をもつ論理式から BDD をつくる場合，変数順序の良し悪しは BDD をつくってみないと分からないことが多い．そこで，BDD を構築する途中に，節点数があるしきい値を超えると，自動的に変数順序改善の機能が起動される「動的順序付け法」[15]という技法も開発されている．

1-4-7 BDD の改良技術

BDD のデータ構造に工夫を加えることで，更に処理効率を向上させようとする研究も多く報告されている．以下にいくつかの例を紹介する．

否定枝（negative edge）[1]は，BDD の枝に否定演算を表す属性を与えるもので，グラフをたどる際には否定枝を通った回数だけ論理を反転させるという意味をもつ．これにより，互いに否定の関係にある BDD を完全に共有することができる（図 1·9(a)）．記憶量が最大 50 %削減できる上に，否定演算を最上位の節点を指す枝の属性を反転するだけで，定数時間

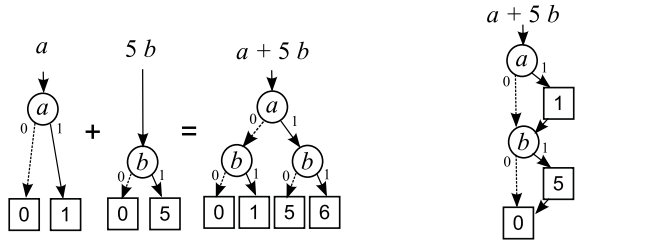
で実行できるようになる．ただし，無制限に使用すると同じ論理関数を何通りにも表せるようになって表現の一意性が失われるため，図 1･9(b) に示すとおり，0-枝には否定枝を使わないこと，及び終端節点は 0 だけにすること，という制限を加える必要がある．否定枝は実装が容易で効果が大きいので，ほとんどの BDD 処理系で採用されている．



(a) **否定枝の効果**　　(b) **否定枝の使用ルール**

図 1･9　否定枝

これまで述べた通常の BDD は終端節点が 0 または 1 の 2 種類だが，これを任意の整数値をもつようにしたのが，**MTBDD**（Multi-Terminal BDD）[3] と呼ばれる BDD である．図 1･10(a) に簡単な例を示す．入力変数は 0,1 の論理値であるが，出力を整数値に拡張した論理関数を表現している．更に図 1･10(b) のように，MTBDD の枝に算術加算を表す属性を付加して，節点数の削減を図ったものを **EVBDD**（Edge Valued BDD）[10] と呼ぶ．MTBDD や EVBDD は，大規模ブール行列の演算処理[3] や 0-1 整数計画法の処理[10] などに利用されている．

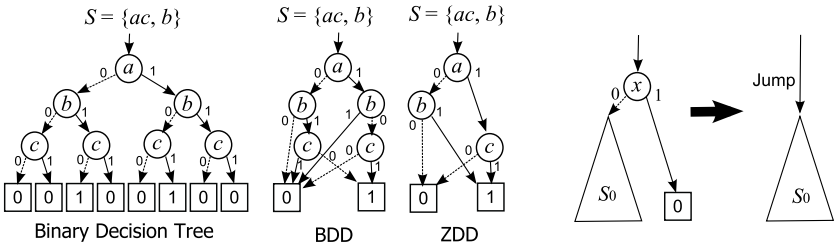


(a) $a + 5\,b$ **を表す** MTBDD　　(b) $a + 5\,b$ **を表す** EVBDD

図 1･10　MTBDD **と** EVBDD

最後に**ゼロサプレス型 BDD**（ZDD, または ZBDD, Zero-suppressed BDD）[12] について述べる．BDD は元々は論理関数を表現するために考案されたものだが，BDD を用いてアイテムの組合せを要素とする集合（組合せ集合，または集合族とも呼ばれる）を表現し，各種の組合せ問題や制約充足問題を解くために利用することもできる．ZDD はそのような用途に適した BDD の変化形である．図 1･11(a) に通常の BDD と ZDD の比較を示す．通常の BDD では，ある節点の 0-枝と 1-枝が同じ行き先であるときに，これを冗長節点として削除するが，ZDD では，図 1･11(b) に示すように 1-枝が 0-終端節点を直接指して

いる場合に，この節点を冗長節点として削除する（その代わり，通常の BDD で削除していた節点は残す）．等価な節点の共有は従来どおり行う．このような非対称な簡約化規則でも，表現の一意性は保つことができる．



(a) **場合分け二分木と** BDD, ZDD **の比較**　　(b) ZDD **の簡約化規則**

図 1·11　ゼロサプレス型 BDD

ZDD は BDD と似たデータ構造をもつが，その解釈が異なる．通常の BDD で論理関数を表現する場合，論理式に出現しない変数は，0 でも 1 でも同じ出力になると考える．一方，組合せ集合や集合族のデータを扱う場合，関係ない変数は，常に 0 に固定（つまり要素として一度も出現しない）と考える方が自然である．ZDD はそのようなモデルを表現するのに適しており，特に疎な組合せを多く含む集合を扱う場合に非常に効率が良く，近年，様々な用途に応用されている[14)]．

そのほか，BDD の変化形については文献 13) にまとめられている．更に，BDD 及び ZDD に関するアルゴリズムの技法については，Knuth の名著「The Art of Computer Programming」の最新巻[9)]でもかなり詳しく取り上げられている．

■参考文献

1) S.B. Akers, “Binary decision diagrams,” *IEEE Transactions on Computers*, vol.C-27, no.6, pp.509-516, 1978.
2) R.E. Bryant, “Graph-based algorithms for boolean function manipulation,” *IEEE Transactions on Computers*, vol.C-35, no.8, pp.677-691, 1986.
3) E.M. Clarke, K.L. McMillan, X. Zhao, M. Fujita, J. Yang, “Spectral transform for large Boolean functions with applications to technology mapping,” In Proc. 30th IEEE/ACM Design Automation Conference (DAC-93), pp.54-60, 1993.
4) R. Drechsler, N. Drechsler, and W. G′unter, “Fast exact minimization of BDDs,” In Proc. 35th IEEE/ACM Design Automation Conference (DAC-98), pp.200-205, 1998.
5) M. Fujita, H. Fujisawa, and N. Kawato, “Evaluation and implementation of boolean comparison method based on binary decision diagrams,” In Proc. of ACM/IEEE International Conf. on Computer-Aided Design (ICCAD-88), pp.2-5, 1988.
6) M. Fujita, Y. Matsunaga, and T. Kakuda, “On variable ordering of binary decision diagrams for the application of multi-level logic synthesis,” In Proc. of IEEE European Design Automation Conference (EDAC-91), pp. 50-54, 1991.

7) 藤田昌宏, 佐藤政生, “特集: BDD（二分決定グラフ）, ”情報処理学会誌, vol.34, no.5, pp.584-630, 1993.
8) N. Ishiura, H. Sawada and S. Yajima, “Minimization of binary decision diagrams based on exchanges of variables,” In Proc. of ACM/IEEE International Conf. on Computer-Aided Design (ICCAD-91), pp.472-475, 1991.
9) Donald E. Knuth, “Section 7.1.4: Binary Decision Diagrams,” In “The Art of Computer Programming,” vol.4, Fascicle 1, Addison-Wesley, to appear, 2009.
10) Y.-T. Lai, M. Pedram, S.B. Vrudhula, “FGLIP: An integer linear program solver based on function graphs,” In Proc. ACM/IEEE International Conference on Computer-Aided Design(ICCAD-93), pp.685-689, 1993.
11) S. Minato, N. Ishiura and S. Yajima, “Shared binary decision diagrm with attributed edges for efficient Boolean function manipulation,” In Proc. 27th ACM/IEEE Design Automation Conference (DAC-90), pp.52-57, 1990.
12) S. Minato, “Zero-suppressed BDDs for set manipulation in combinatorial problems,” In Proc. of 30th ACM/IEEE Design Automation Conference, pp.272-277, 1993.
13) S. Minato, “Graph-based representations of discrete functions,” In T. Sasao, editor, “Representation of discrete functions,” Kluwer Academnic Publishers, chapter 1, pp.1-27, 1996.
14) 湊　真一, “［招待講演］二分決定グラフ（BDD）を活用したデータマイニング・知識発見技術の最近の話題, ”電子情報通信学会人工知能と知識処理研究会, 信学技報, vol.107, no.78, AI2007-6, pp. 27-32, 2007.
15) R. Rudell, “Dynamic variable ordering for ordered binary decision diagrams,” In Proc. ACM/IEEE International Conference on Computer-Aided Design(ICCAD-93), pp.42-47, 1993.
16) S. Tani, K. Hamaguchi, and S. Yajima, “The complexity of the optimal variable ordering problems of shared binary decision diagrams,” Springer, In 4th International Symposium on Algorithms and Computation, vol.LNCS-762, pp.389-398, 1993.