2.コンピュータ科学基礎 II: 論理演算と論理回路

この章では、1 章で説明した 0/1 で表された情報をどのように操作(演算)するかについて説明する。2.1 節では 0/1 に対する操作である論理演算について説明し、2.2 節では、もう少し大きな単位の情報の操作について説明する。

2.1 論理演算と論理回路

コンピュータ内部では、0/1 で表された情報を**論理演算**という演算で操作する。この論理演算は論理回路という形で電気的にコンピュータ内部に実装されている。この節では、コンピュータの内部の論理回路を理解するために必要な論理演算と、論理回路の実装の初歩を説明する。

(1)命題

論理演算とは、一つまたは複数の命題から新しい**命題**を作ることをいう。命題とは、その内容が「真」または「偽」のいずれかと判定できることをいう。命題の真を、”True”または”1”、偽を”False”または”0”で表す。コンピュータに関するビット演算は、命題の真偽を”1”、”0”とした論理演算をいう。

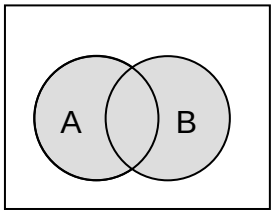
(2)論理演算

論理演算には、論理積、論理和、排他的論理和、否定がある。論理演算の結果を表にまとめた真理値表で、それぞれの論理演算を説明する。同様の演算は集合論等にも使われており、集合論で使われるベン図を使うと理解しやすい。論理変数に対する処理を論理演算を用いて表した物を**論理式**と呼ぶ。

論理和(OR), “+”とも表す。A + B

いずれか一方が真(“1”)であれば、結果が真となる演算。以下は、論理演算における変数の値と演算結果を示した表で、**真理値表**と呼ぶ。

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

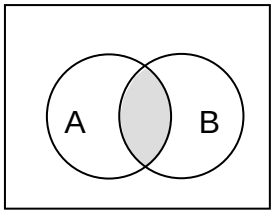


左記がベン図。灰色の部分が論理和。

論理積(AND) ”･”とも表す。A･B

双方がが真(“1”)の場合のみ、結果が真となる演算。

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

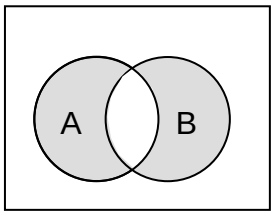


灰色の部分が論理積

排他的論理和(Exclusive OR; EOR, XOR) （言語によっては″A ^ B″とも表す）

いずれか一方のみが真(“1”)の場合、結果が真となり、双方の値が一致した場合には結果が偽となる演算。

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

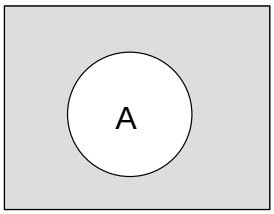


灰色の部分が排他的論理和

否定(NOT), $\overline{A}$ であらわす。（言語によっては″!A″や″~A″とも表す）

真偽を反転させる演算

A	NOT A($\overline{A}$)
0	1
1	0



灰色の部分が否定

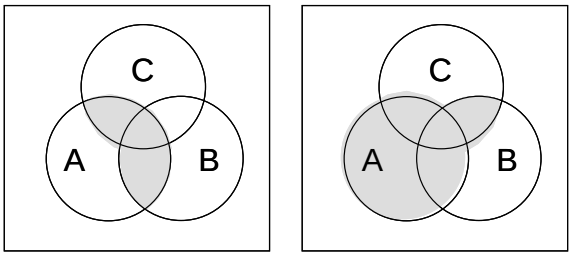
(3)論理演算の公式

一般的な数学における公式と同様、論理演算においても公式が存在する。うまく使えば、論理式を単純化にして、後述の論理回路として実装する時の回路量を削減することができる。

分配則

$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$

$A + (B \cdot C) = (A + B) \cdot (A + C)$

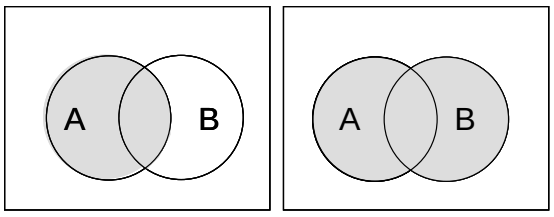


吸収則

$A + (A \cdot B) = A$

$A \cdot (A + B) = A$

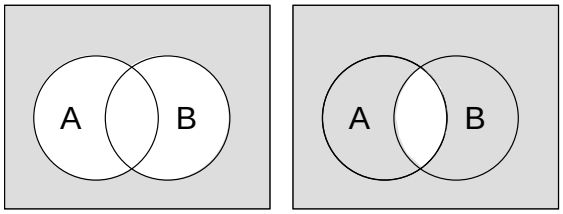
$A + (\overline{A} \cdot B) = A + B$



ド・モルガン

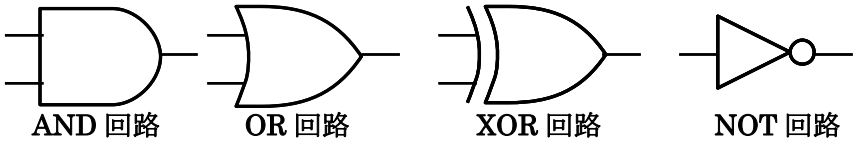
$\overline{(A \cdot B)} = \overline{A} + \overline{B}$

$\overline{(A + B)} = \overline{A} \cdot \overline{B}$



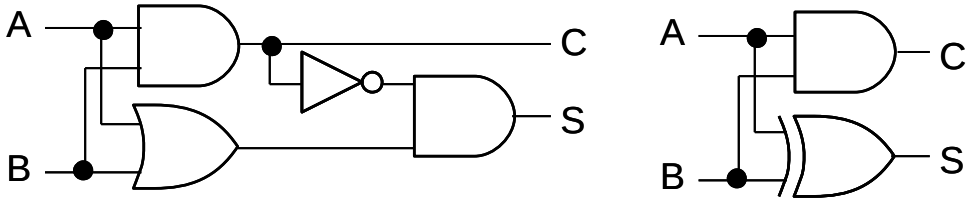
(4) 論理回路(加算回路)

論理式は論理変数が流れる信号線と、入力された信号線に対して論理演算を行って結果を出力する**論理素子**を利用して、**論理回路**として実装することができる。
論理回路を図に示す場合には以下の論理素子の **MIL 記号**を用いる。



半加算回路

A と B を加算の結果を S(Sum)に、桁上がりがあれば C(Carry)に出力する回路。2 進数の 1 桁の加算を実現できる。



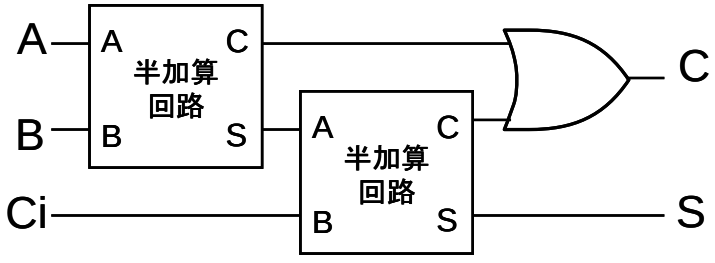
半加算回路の真理値表

入力 A	入力 B	出力 C	出力 S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

「入力の"1"の数が 1 個の時に S が"1"に、2 個の時に C が"1"になる。」と考えても良い。

全加算回路

複数桁のビットの加算を行うためには、足す数と足される数に加え、下の桁からの桁上がりを考えなくてはならない。二つの入力 A と B に加えて、下の桁からの繰り上がり(Ci: Carry in)も考慮に入れた加算回路を**全加算回路**とよぶ。



最下位桁は半加算回路、それ以外を全加算回路として必要な桁数分回路を並べれば、任意のビット数の加算を行う論理回路ができる。このような再帰的な設計は、プログラムの分野のみならず、論理回路の分野でもよく用いられる。

加算回路があれば負の数の補数表示により減算も可能になる。この減算の実現は、引く数のビットの反転(NOT)と最下位桁も全加算回路を用いて Ci で+1 を行うことで実現できる。また、加算を繰り返すことにより乗算を、減算を繰り返すことにより除算を行うことも可能となる。

全加算回路の真理値表

入力 A	入力 B	入力 Ci	出力 S	出力 C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

「S は入力のうち"1"が 1 個か 3 個なら"1"となり、C は入力のうち"1"が 2 個以上なら"1"となる。」と考えても良い。

(5)ビット演算

ビット演算は、ビットの同一の桁のみを考慮して演算をすることにより達成される。加算回路と同様に、論理素子を用いて論理回路として実現できる。

```
A         01010101
B         00001111   値の一部を切り出す使い方が便利
A AND B   00000101   (例：下位 4 ビット切り出し)
```

```
A         01010101
B         00111100
A OR B    01111101

A         01010101
B         00111100
A XOR B   01101001

B         00111100
NOT B     11000011
```

ビットシフト演算

ビット列の並びをそのまま左右にシフトする、ビットシフト演算という演算も存在する。ビットシフト演算には、論理シフトと算術シフトの 2 種類がある。

論理シフト演算

指定されたビット数だけ全体をシフトし、空いたビットには 0 を挿入する。右に n ビットシフトすると 2^{-n} 倍、左にシフトすると数値を 2^{n} 倍することになる。これは、加減算のくりかえしで実現しなくてはならない通常の乗除算による 2^{n} 倍／2^{-n} 倍と比較して、高速に処理できる。

```
C         10111100
C >> 2    00101111 (">> 2"は右に 2 ビットシフトの意味)

C         10111100
C << 2    11110000 ("<< 2"は左に 2 ビットシフトの意味)
```

算術シフト演算

2 の補数で表記された負の数を右論理シフトすると、符号ビットがあった場所に 0 が挿入されるため、正の数に変わってしまう。値の符号を保ったままシフト演算を行うため、算術シフトがある。

算術シフトでは、符号ビット（一番左のビット）を固定することで、負の数のシフト演算を可能にする。右シフトの場合は、符号ビットと同じ値を空いたビットに挿入する。左シフトの場合は、空いたビットに 0 を挿入する。

```
C         10111100
C >> 2    11101111 (空いた上位 2 ビットは符号ビットを挿入)
```

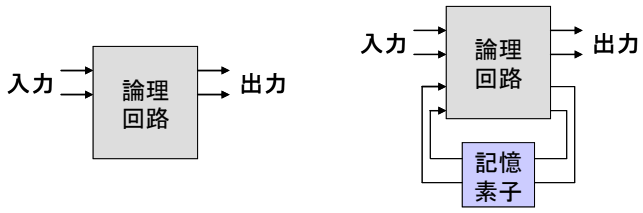
C　　10111100 左シフトは論理シフトと変わらないが、オーバーフロー
C << 1　01111000 には注意(左記では-68×2=-136 で発生している)

シフト演算の組み合わせによる計算

元の値を５倍したい場合、２ビット左にシフト(2^2)し、元の値を加算すれば５倍となる。これは、乗算や除算を高速に実行する手法として、プログラム中でしばしば使われる。(乗算／除算は加算／減算の繰り返しで実装されるため、一般的な CPU では、加減算に比べて乗算は数倍ほど、除算は 10 数倍ほど時間がかかる。)

(6) 順序回路

前述の加算回路やビット演算の回路は、入力に対して出力は一意に決まる。このような論理回路を、**組み合わせ論理回路**と呼ぶ。一方、記憶素子(フリップフロップなど)に内部状態を保持し、出力は入力と内部状態の両方から決まる回路も存在し、このような回路は**順序回路**と呼ばれる。順序回路と呼ばれる由縁は、出力の一部を内部状態の更新に用いることで、入力によって内部状態を順番に更新して動作する形を取る物がおおいからである。



組み合わせ論理回路　　**順序回路**

順序回路の例：　自動販売機
・内部状態(投入金額)が 50 円の時に 100 円を入れる。　->投入金額の出力は 150 円。
・内部状態(投入金額)が 0 円の時に 100 円を入れる。　->投入金額の出力は 100 円。

2.2 構文と数式の論理表現と操作

2.1 節ではビット単位での情報の操作について説明したが、この節では、もう少し大きな単位での情報の論理表現や操作について概要を説明する。以下、大きな単位の情報として、構文と数式の論理表現と操作の例を示す。

2.2.1 BNF(Backus Naur Form)記法

BNF 記法は J.Backus によって提案され、P.Naur がプログラミング言語 ALGOL60 の定義に用いた構文表記法である。BNF 記法は、次の３つの要素から構成される。

(1)終端記号(terminal symbol)：言語を構成するための基本記号。

(2)非終端記号（nonterminal symbol;メタ変数ともいう)：言語を表記する構文規則上の変数。

(3)メタ記号(metasymbol)：構文の定義のために用いる記号

::=	左辺が右辺に定義される。
|	左辺と右辺の記号列の論理和をとる。
< >	< >で囲まれた系列を一つの非終端記号として扱う。

構文の定義に必要な規則数を減らすため、しばしば、再帰的な定義が使われる。

例 1：構文規則 1

```
<文>          ::= <名詞句> <動詞句>
<名詞句>      ::= <形容詞> <名詞句>
<名詞句>      ::= <形容詞> <名詞>
<動詞句>      ::= <動詞> <副詞>
<形容詞>      ::= black | little
<名詞>        ::= dogs
<動詞>        ::= ran
<副詞>        ::= quickly
```

上記の構文規則では、black, little, dog, ran, quickly が終端記号であり、<文>, <名詞句>, <動詞句>, <形容詞>, <名詞>, <動詞>, <副詞>が非終端記号となる。英文"Black little dogs ran quickly."は上記の構文規則により<文>として表現されている。これは、以下の形の書き換え操作により<文>から英文を求めることができるからである。下記の過程を**導出**(derivation)と呼ぶ。

```
<文>          ⇒ <名詞句> <動詞句>
              ⇒ <形容詞> <名詞句> <動詞句>
              ⇒ <形容詞> <形容詞> <名詞> <動詞句>
              ⇒ <形容詞> <形容詞> <名詞> <動詞> <副詞>
              ⇒ black little dogs ran quickly
```

構文の定義は厳密に行う必要がある。さもなければ、構文規則違反にしたかった表記が、構文的に正しいと受理されてしまう。以下は、その例である。

例 2：10 進整数の定義 1(不完全)

```
<digit>          ::= 0|1|2|3|4|5|6|7|8|9
<unsigned integer>::= <digit> | <digit><unsigned integer>
<integer> ::= <unsigned integer> | +<unsigned integer> | -<unsigned integer>
```

上記の規則では、

<unsigned integer>⇒<digit><unsigned integer>

⇒0<unsigned integer>

⇒0<digit><unsigned integer>

⇒01<unsigned integer>

⇒015

と導出できるので、頭が 0 の文字列も 10 進数となる。

例 3：10 進整数の定義 2(不完全)

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<non-zero digit> ::= 1|2|3|4|5|6|7|8|9

<unsigned integer>::= <digit> | <non-zero digit><unsigned integer>

<integer> ::= <unsigned integer> | +<unsigned integer> | -<unsigned integer>

上記の規則では、

<unsigned integer>⇒<non-zero digit><unsigned integer>

となるので、頭に 0 を置くことはできなくなる。しかしながら、

<integer>⇒-<unsigned integer>

⇒-<digit>

⇒-0

<integer>⇒+<unsigned integer>

⇒+<digit>

⇒+0

<integer>⇒<unsigned integer>

⇒<digit>

⇒0

となり、３種の０の記法を許すことになる。

例 4：10 進整数の定義 3

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<non-zero digit> ::= 1|2|3|4|5|6|7|8|9

<unsigned integer>::= <non-zero digit> | < unsigned integer ><digit>

<integer> ::= 0| <unsigned integer> | +<unsigned integer> | -<unsigned integer>

上記の規則では、

<unsigned integer>::=<non-zero digit>|< unsigned integer ><digit>

により

<unsigned integer>⇒<non-zero digit>

が導かれ、

<unsigned integer>として、1,2,…,9 は導出できる。

<unsigned integer>⇒<unsigned integer>< digit>

⇒[1-9 のどれか]<digit>

⇒[1-9 のどれか] [0-9 のどれか]

となるので、頭が 0 の文字列は<unsigned integer>において定義されない。

<integer> ::=0|<unsigned integer>|+<unsigned integer>|-<unsigned integer>

より、

<integer>⇒0

<integer>⇒<unsigned integer>

<integer>⇒+<unsigned integer>

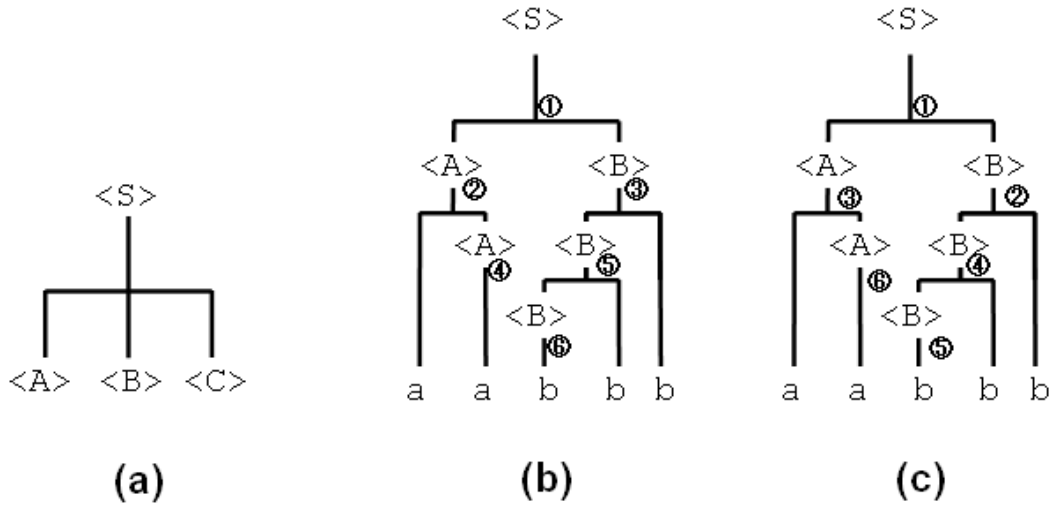
<integer>⇒-<unsigned integer>

ここで、<unsigned integer>では 0 は導出されないので、<integer>として表現できるのは 0 であり、+0、-0 は導出できない。

構文木(syntax tree)とは、構文の導出過程を木構造で表したものである。

<S>⇒<A><B><C>

について図(a)のように記述する。式の<A><B><C>の順番は構文木の葉<A>,<B>,<C>の順番と対応する。



図(b)(c)は構文規則で文"aabbb"を導出する過程である。

```
<S>          ::= <A><B>                    [1]
<A>          ::= a | a<A>                  [2]
<B>          ::= b | <B>b                  [3]
```

により aabbb を導出する過程を構文木で書くと図(b)になる。図中の番号は以下の導出順

と対応する。

<S> ⇒<A><B> 規則[1]
⇒a<A><B> 規則[2]
⇒a<A><B>b 規則[3]
⇒aa<B>b 規則[2]
⇒aa<B>bb 規則[3]
⇒aabbb 規則[2]

同様に aabbbb は以下により導出できる。

<S> ⇒<A><B> 規則[1]
⇒<A><B>b 規則[3]
⇒a<A><B>b 規則[2]
⇒a<A><B>bb 規則[3]
⇒a<A>bbb 規則[3]
⇒aabbb 規則[2]

この過程を図(c)により構文木で表すことができる。ここで重要なことは、図(b)と図(c)の構文木は、適応する規則の順は異なっても構造が同じであるという点である。

１つの文について構文規則から生成される構文木が複数存在する場合、このような文を**あいまい文**といい、あいまいな文を生成する規則を**あいまいな構文規則**という。

例 5：<S>::=0|<S>1<S>

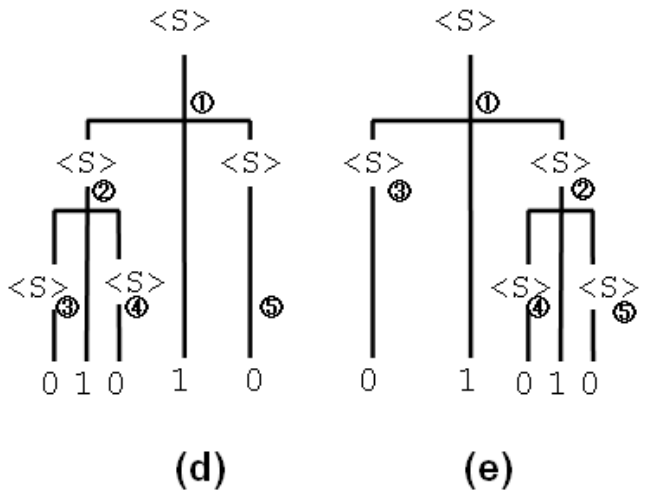
この構文規則を使うと、

<S> ⇒<S>1<S> ①
⇒<S>1<S>1<S> ②
⇒ 0 1<S>1<S> ③
⇒ 0 1 0 1<S> ④
⇒ 0 1 0 1 0 ⑤

と導出され、構文木は図(d)となる。一方、

<S> ⇒<S>1<S> ①
⇒<S>1<S>1<S> ②
⇒ 0 1<S>1<S> ③
⇒ 0 1 0 1<S> ④
⇒ 0 1 0 1 0 ⑤

とも導出され、構文木は図(e)となる。



これら二つの構文木の構造は異なっている。一つの文に対して構文木が 2 通り以上あることは「文が複数とおり解釈できる」ことを意味する。プログラム言語で用いられる文はあいまいなく記述することが必要である。例 9 のように構文規則を表現すればこのことは回避される。

例 9：	<S>::=0|<S><t> <t>::=1<S>

2.2.2 正規表現

いくつかの文字列を一つの形式で表現するための表現方法をいう。主に文法規則を表現するために用いられる。現在のプログラミング言語やアプリケーションの文字列処理でよく使われる。

記号	意味
.	任意の一文字を示す。
+	直前の文字またはパターンの 1 回以上の繰り返しを表す。
*	直前の文字またはパターンの 0 回以上の繰り返しを表す。
?	直前の文字またはパターンの 0 回または 1 回現れることを表す。
[abc]	カッコ内に含まれる文字の群からの任意の 1 文字の選択を表す。
[m-n]	m から n までの連続した文字の群からの任意の 1 文字の選択を表す。
[^m-n]	m から n までの連続した文字の群の含まれない 1 文字の選択を表す。
^	行頭の選択を表す。
$	行末の選択を表す。

例：

・[Nn][Aa][Ii][Ss][Tt]： NAIST, naist, Naist などにマッチする
・[0-9]+： 任意の 1 文字以上の数字にマッチする
・0.[0-9]+： 1 未満の小数で表記された数字にマッチする(."."は正規表現の規則から一時的に外れているものとする)

正規表現はプログラミング言語によって独自の拡張がされていることが多い。特に、Light Weight Language(Perl, PHP, Ruby など)では、より文字列処理に便利なように、大文字、アルファベット、空白等を示すルールが追加されている。

2.3 演算子と式の表現

演算子における引数の数により、単項演算子、二項演算子などと呼ぶ。

単項演算子

~x： 否定演算

二項演算子

2 個の引数に対してなされる演算をいう。

算術演算	+, －, ×, ÷など
関係演算	≠, ≦, ≧, >, <など
論理演算	and, or など

２項演算の例のように、演算子が引数の間に出現する表記法を**中置記法(infix notation)**という。演算の意味を変えずに演算子を前置した記法を**前置記法(prefix notation)**、演算子を後置した記法を**後置記法(postfix notation)**という。以下、計算機内部での式の表現としてよく用いられる、逆ポーランド記法について詳細を説明する。

x + y	中置記法
+ (x, y)	前置記法, ポーランド記法
(x, y) +	後置記法, 逆ポーランド記法

逆ポーランド記法

A+B を逆ポーランド記法で書くと

AB+

となる。

例 (a+b)÷c-d を逆ポーランド記法で書こう。

まずはじめに演算するところは

(a+b)

でありこの部分を逆ポーランド記法で書く

ab+

となるので

ab+÷c-d

続いて計算するところは、

÷c

であるので

ab+c÷-d

最後に

-d

を計算するので

ab+c÷d-

となる。

上記の例で分かるように、逆ポーランド記法は演算子を置く位置で演算の順序を明示的に示すことができるため、括弧による演算順序の指示が不要となる。この利点と、後の章で説明するスタックというデータ構造で実現できる利点のため、逆ポーランド記法は計算機用の式の表現として広く使われている。

参考書

白鳥則郎、高橋薫、神長裕明著、情報系教科書シリーズ第８巻ソフトウエア工学の基礎知識、昭晃堂, (1997)

練習問題

第２章

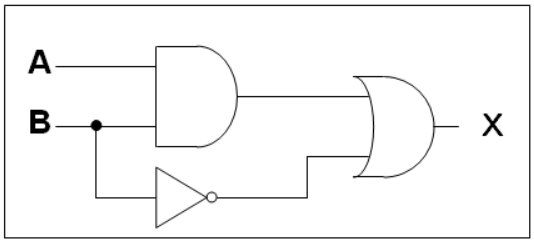
問題 2.1 真理値表と等価な論理式はどれか。ここで、・は論理積、＋は論理和、$\overline{A}$ は A の否定を表す。

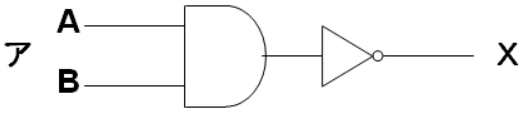
x	y	演算結果
0	0	0
0	1	0
1	0	1
1	1	0

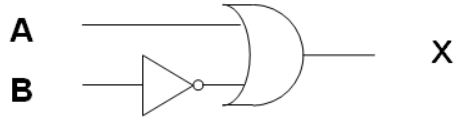
ア　$x+\overline{y}$　　イ　$\overline{x}+y$　　ウ　$x\cdot\overline{y}$　　エ　$\overline{x}\cdot y$

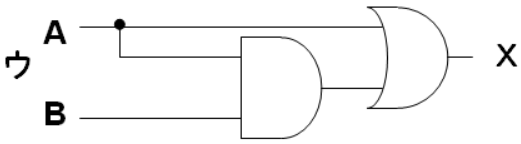
（基 14 秋午前問 7）

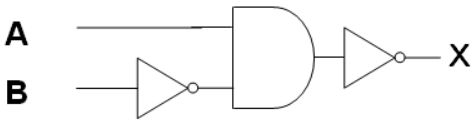
問題 2.2 枠で囲まれた論理回路と同じ出力が得られる論理回路はどれか。



ア


イ


ウ


エ


（基 14 秋午前問 7）

問題 2.3 32 ビットのレジスタに 16 進数 ABCD が入っている。これを 2 ビットだけ右に論理シフトしたときの値はどれか。

ア　2AF3　　イ　6AF3　ウ AF34　エ EAF3　　（基 16 春午前問 4）

問題 2.4 数値に関する構文が次の通り定義されているとき<数値>として扱われるのはどれか。

<数値> ::= <数字列>| <数字列>E <数字列>| <数値列>E<符号><数字列>

<数字列> ::= <数字>| <数字列> <数字>

<数字> ::= 0|1|2|3|4|5|6|7|8|9|

<符号> ::= +|－

ア　-12　　イ　12E-10　ウ　+12E-10　エ +12E10　　（基15春午前問11）

問題 2.5 正規表現[A-Z]+[0-9]＊が表現する文字列の集合要素となるものはどれか。

ここで正規表現は次の規則に従う。

[A-Z]は、英字 1 文字を表す。

[0-9]は、数字 1 文字を表す。

＊は、直前の正規表現の０回以上の繰り返しを表す。
+は、直前の正規表現の１回以上の繰り返しを表す。
ア　4567879　イ　ABC99＊　ウ　ABC+99　エ ABCDEF　（基15春午前問11）

問題 2.6 数値を２進数で格納するレジスタがある。このレジスタに正の整数 x を入れた後、「レジスタの値を２ビット左にして、これに x を加える」操作を行うと、レジスタの値は x の何倍になるか。ここでシフトによるあふれ（オーバーフロー）は発生しないものとする。
ア　3　イ　4　ウ　5　エ 6　（基 15 春午前問 3）

問題 2.7 次の論理式 $A \vee (\overline{A} \wedge B)$ と等価なものはどれか。ここで∧は論理積、∨は論理和、$\overline{X}$ はXの否定を表す。
ア　$A \wedge B$　イ $A \vee B$ ウ　$A \wedge \overline{B}$　エ　$A \vee \overline{B}$
（基 15 春午前問 7）

問題 2.8 最上位パリティビットとする８ビット符号において、パリティビット以外の下位７ビットを得るためのビット演算はどれか。
ア　16 進数 0F との AND をとる。
イ　16 進数 0F との OR をとる。
ウ　16 進数 7F との AND をとる。
エ　16 進数 FF との XOR（排他的論理和）をとる。
（基 18 春午前問 6）

問題 2.9 次の中置記法を前置記法で記述せよ。ただし、×は+より優先順位は高い物とする。
(A) a + b + c　(B) a + b × c　(C) a × b + c × d

問題 2.10 次の前置記法を中置記法で記述せよ。ただし、×は+より優先順位は高い物とする。
(A) ××abc　(B) +a×bc　(C) ×+abc