2. 論理演算

コンピュータ内部での演算の仕組み
論理演算と論理回路

概要

1章で表した情報を操作しよう

- 2.1 論理演算と論理回路
  - 論理演算
  - 論理演算による2進数の加算の実現
  - ビット演算
  - 組み合わせ回路と順序回路

もっと大きな単位の情報の操作を考えよう

- 2.2 構文と数式の表現
  - BNF(バッカス・ナウア記法)
  - 正規表現
  - 演算子と式の表現

2.1 論理演算と論理回路
(1)命題

命題：その内容が「真」または「偽」のいずれかと判定できる文をいう。

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。

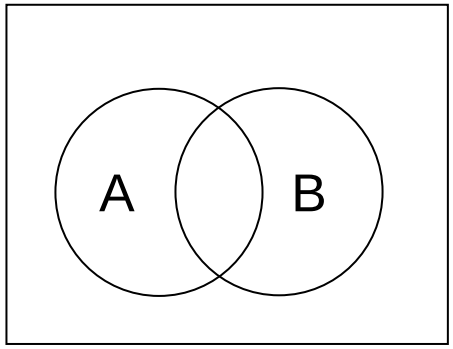
論理演算の種類

- 以下の4種類がある
  - 論理和(OR)
  - 論理積(AND)
  - 排他的論理和(Exclusive OR, EOR, XOR)
  - 否定(NOT)
- 入力変数と演算結果は真理値表を使ってよく表す
- 集合論でも使うベン図を使うと演算を理解しやすい

真理値表　入力　演算結果

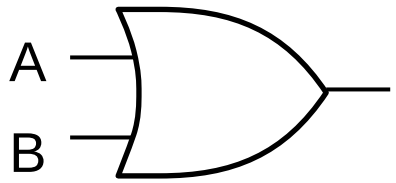
A	$\overline{A}$
0	1
1	0

ベン図

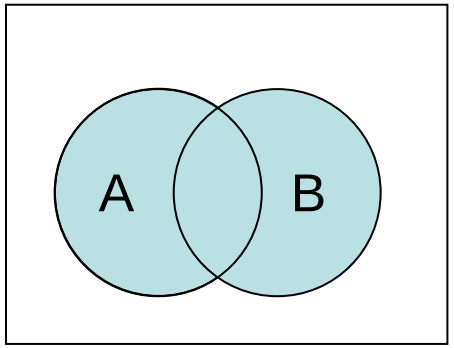


論理和

- ORとも呼ぶ
- 集合論の和集合
- 演算子は“+”



MIL記号



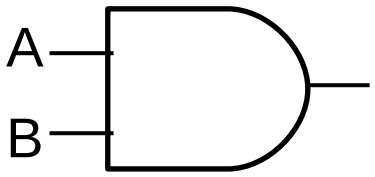
ベン図

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

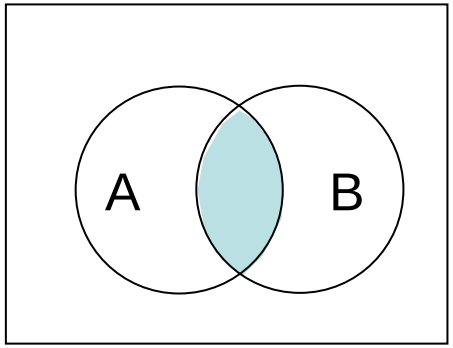
真理値表

論理積

- ANDとも呼ぶ
- 集合論の共通部分
- 演算子は“・”



MIL記号



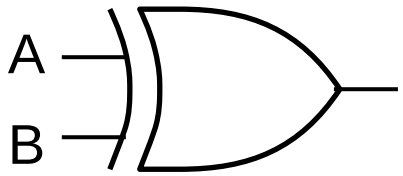
ベン図

A	B	A・B
0	0	0
0	1	0
1	0	0
1	1	1

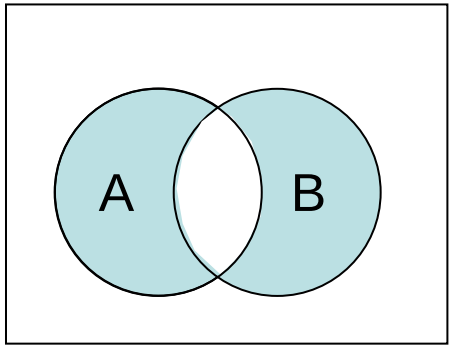
真理値表

排他的論理和

- Exclusive ORとも呼ぶ
  - EOR/XOR/EX-ORと略記が多い
- 決まった演算子はない
  - “^”や”⊕”で表す例もある



MIL記号



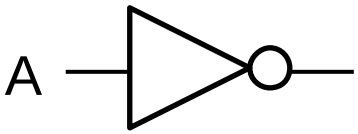
ベン図

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

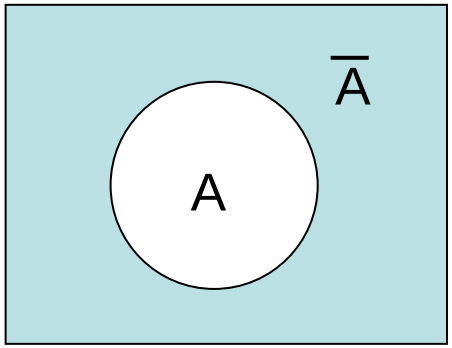
真理値表

否定

- NOTとも呼ぶ
- 集合論の補集合
- 演算子は“$\overline{A}$”のようにバーを追加する
  - “~A”や”!A”で表す例もある



MIL記号



ベン図

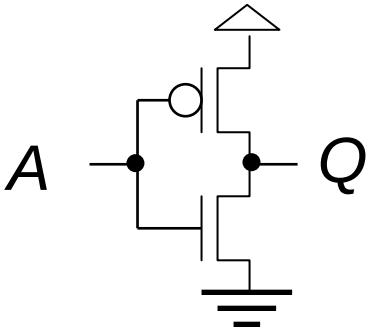
A	$\overline{A}$
0	1
1	0

真理値表

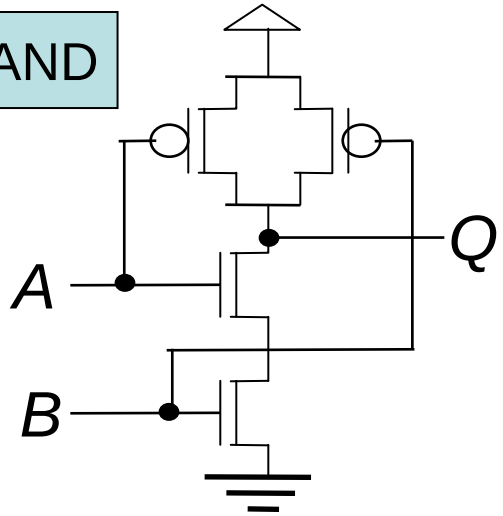
なぜにこのような演算を使うか？

- 電子回路等で実現しやすい
  - 現在は電界効果トランジスタ(FET)
  - 過去にはトランジスタやリレーなど
  - 太古には機械式計算機という時代も
- 実装した回路を論理回路と呼ぶ

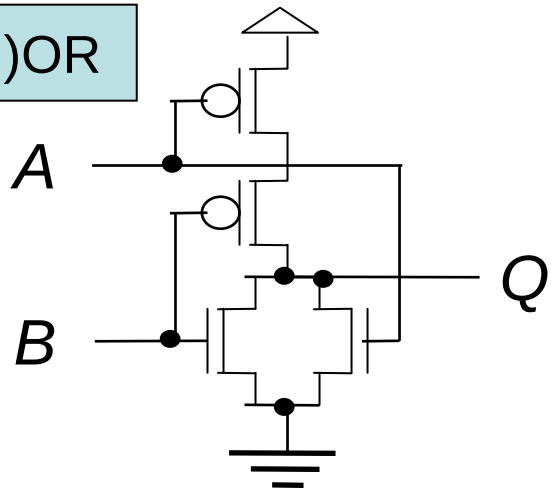
NOT



(N)AND



(N)OR



論理演算の公式

- 一般的な数学と同様、論理演算にも公式があります
  - 分配則：一般的な数学と同じです
  - 吸収則、ド・モルガンの法則：
    論理演算の独特な物。ベン図を描けば分かりやすい。
- 一般的な数学の公式と同様に式の整理に使えます
  - 式が簡単になる
    ->回路にする時に論理素子が少なくなる
    ->回路量が減って回路面積や消費電力が少なくなる
  - 詳しくはハードウェア設計論Iで！

半加算回路

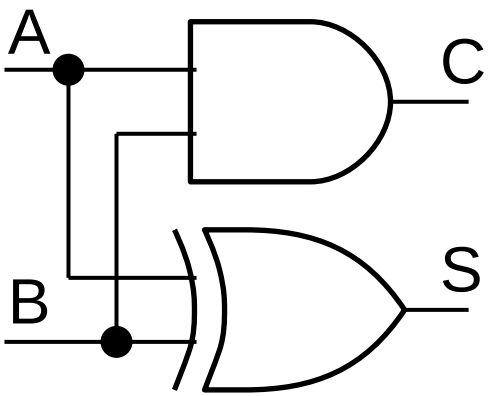
- 論理演算で1ビットの加算を実現してみる
  - 論理回路の状態で示す
  - 入力：2ビット、出力：2ビット(和＋桁上がり)
- 下記の回路となる
  - 半加算回路と呼ぶ

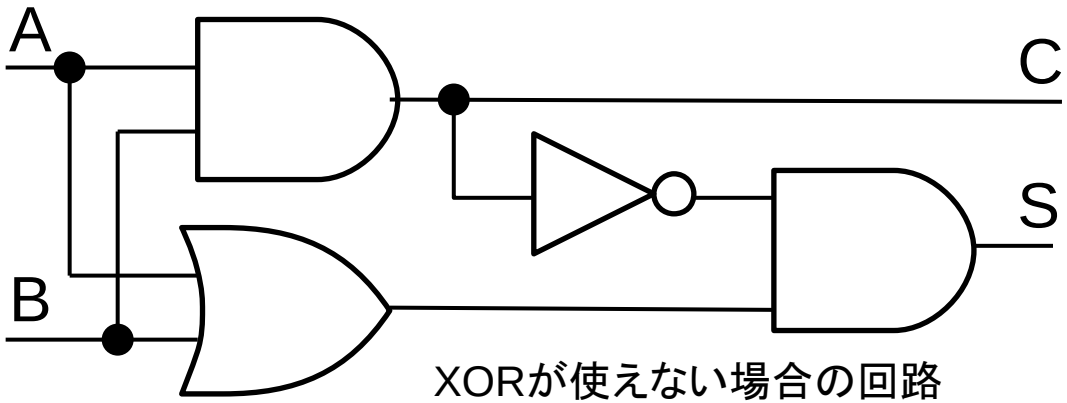
ここ！

```
  0000 1011  (11)
+ 0000 1100  (12)
  0001 0111  (23)
```

半加算回路の真理値表

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



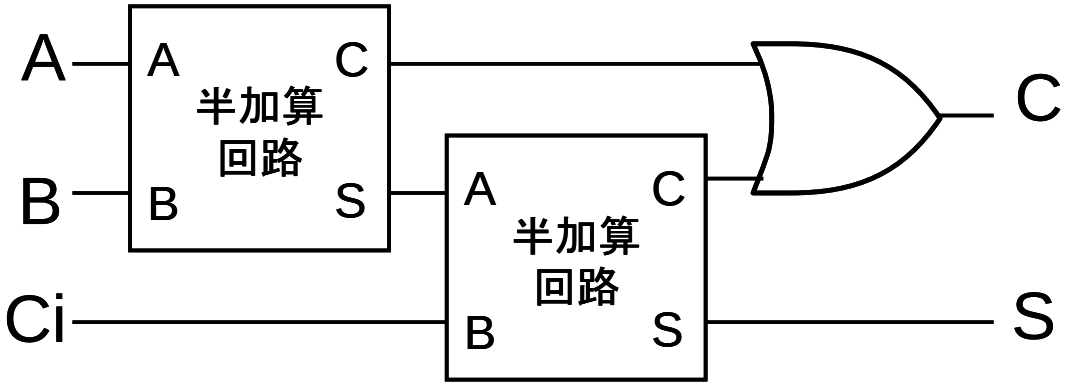


XORが使えない場合の回路

全加算回路

- 中間のビットでは下位からの桁上がりも考慮する必要がある
  - 入力：3ビット、出力：2ビット
- 半加算回路を使うと以下の回路になる
  - 全加算回路と呼ぶ
  - 全加算回路を使ってさらに大きな構造も…

A	B	Ci	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

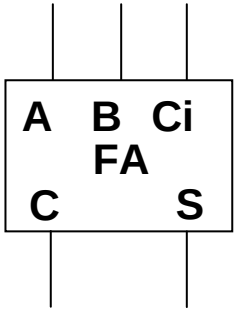


下位からの桁上がりも必要

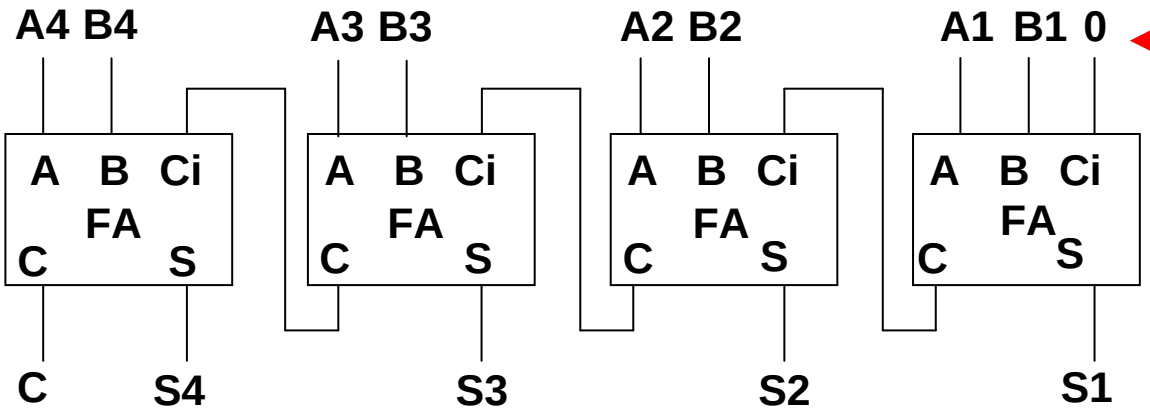
```
      1
  0000 1011  (11)
+ 0000 1100  (12)
  ---------------
  0001 0111  (23)
```

演習

- 全加算回路(FA: Full Adder)を4つ使って4ビット加算回路を設計してみよう



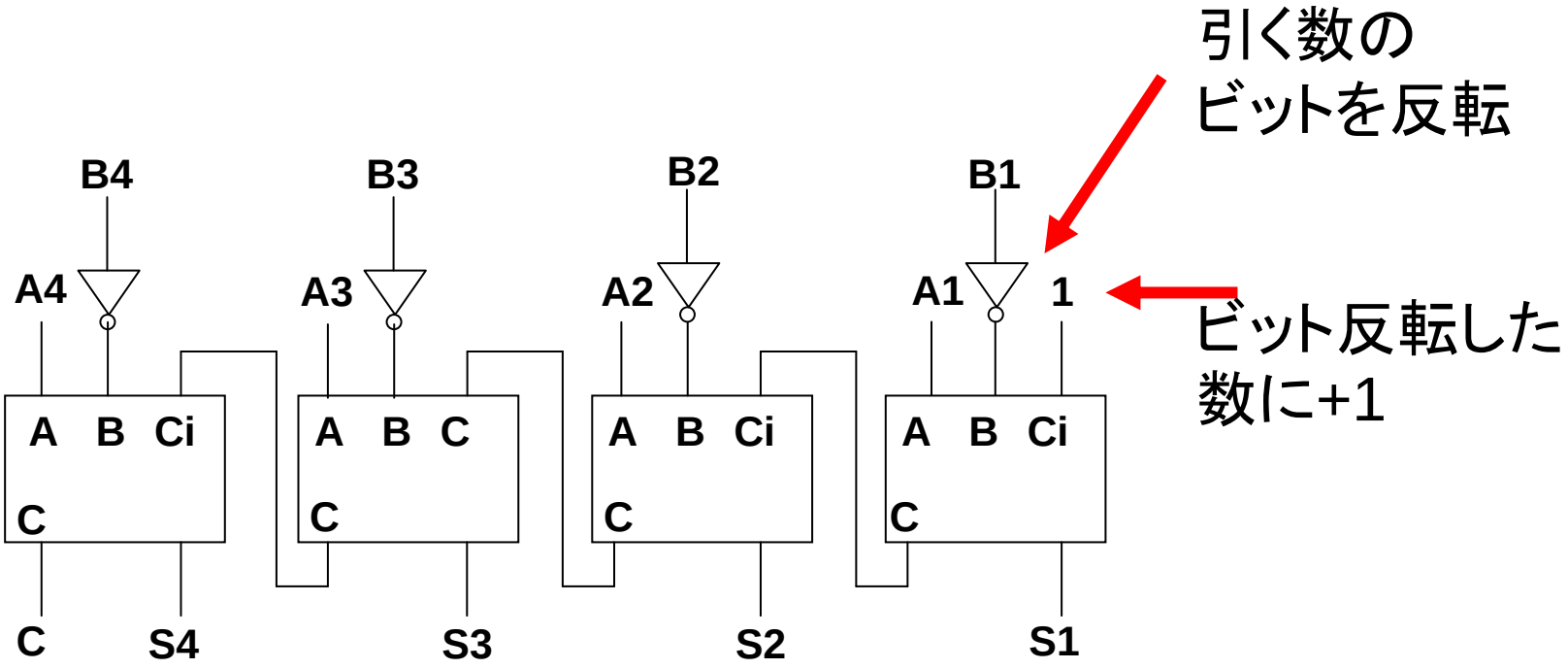
全加算回路(FA)を4つ使って
4ビット加算回路を設計してみよう



最下位のCiには
0を入れておく

4ビットの減算器

2の補数に対する演算の特徴を利用



ビット演算

- ビットの同一の桁に対して演算を行う
- 特徴的な使い方
  - 論理積: 値の一部を切り出すのに使えたりする

A	01010101
B	00001111
A AND B	00000101

  - 否定: 値の反転(2の補数など)

A	00111100
NOT A	11000011

ビットシフト演算(論理シフト)

- ビットの並びをそのままに左右に移動させる
- 空いたところには0を入れる
- nビットシフトした時の値は2^n倍か2^{-n}倍になる
  - 次の算術シフトと区別のため、論理シフトとも呼ばれる
  - オーバーフローには注意

	00011000	$A = 2^4+2^3$ (24)
左に2ビットシフト	01100000	$2^6+2^5 = 2^2A$ (96)
右に2ビットシフト	00000110	$2^2+2^1 = 2^{-2}A$ (6)
左に4ビットシフト	10000000	$2^{-7} \neq 2^4A$?? (-128??)

オーバーフロー発生！

算術シフト

- 2の補数を右シフトすると負の数が正の数になる
  - 空いた所を0で埋めたことが原因

10011000　A = $-2^7+2^4+2^3$ (-104)

右に2ビット論理シフト　00100110　$2^5+2^2+2^1 \neq 2^{-2}$A?? (38??)

- 符号ビットをで埋めれば問題は発生しない
  - 算術シフト演算と呼ぶ
  - 左シフトは論理シフトと変わらない

10011000　A = $-2^7+2^4+2^3$ (-104)

右に2ビット算術シフト　11100110　$-2^7+2^6+2^5+2^2+2^1 = 2^{-2}$A (-26)

シフトと加算による乗除算

- 乗除算は加減算の繰り返しで実現
  ->加減算よりも数倍～数十倍時間がかかる
- 2^nで乗除算するならシフトのみで実現
- 乗数が2進数でシンプルに表されるなら、シフトと加算で乗算を高速化できる
  - 例：00001110 (14)を6倍にする
    - 14を左1ビット(2倍)、左2ビット(4倍)シフトしたものを準備
    - 加算それらを加算する

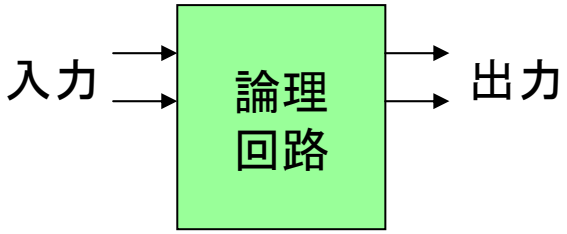
00011100 (28)

+ 00111000 (56)

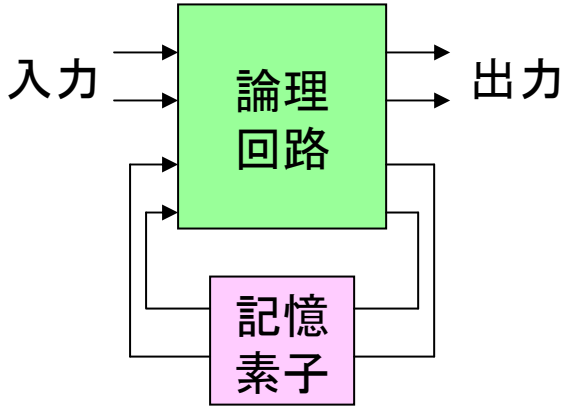
01010100 (84)

組み合わせ回路と順序回路

- 前述のビットシフトや加算の論理回路は組み合わせ(論理)回路に分類される
  - 入力に対して出力が一意に決まる
- 内部状態を持ち、入力と内部状態で出力が決まる回路は順序回路と呼ばれる(例：自動販売機)
  - 出力の一部は内部状態の更新に用いられる
  - 除算回路は順序回路になる



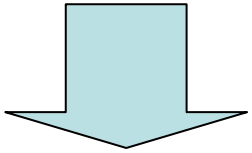
組み合わせ回路



順序回路

2.2 構文と数式の表現

- 1章で話したように、コンピュータ内ではアルファベットも数字も画像も2進数で書かれた羅列
- では、どれが意味のある数字とかを判別するか？



- 構文ルールに則って解析し、一致したらその構文ルールに則った意味を持つと考える
- 本節では、構文やその解析について説明
  - 数式表現も構文の1種

2.2.1 バッカス・ナウア(BNF)記法

- 古典的な構文規則の1つ
  - 分かりやすいので、教科書でよく用いられる
- Backus氏とNaur氏によって作られた
- ALGOL 60 (1960年)という言語を定義に用いられた

BNFのルール

- 終端記号：言語を構成するための基本記号
- 非終端記号：言語を表記する構文規則上の変数
- メタ記号：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
- 再帰的な導出もOK
  - 導出： ルールに従った書き換えによるルールに適合しているかの判定

BNFの構文規則定義の例

- 青い文字が終端記号
- それ以外の<>に囲まれた部分は非終端記号

<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

BNFによる導出の例

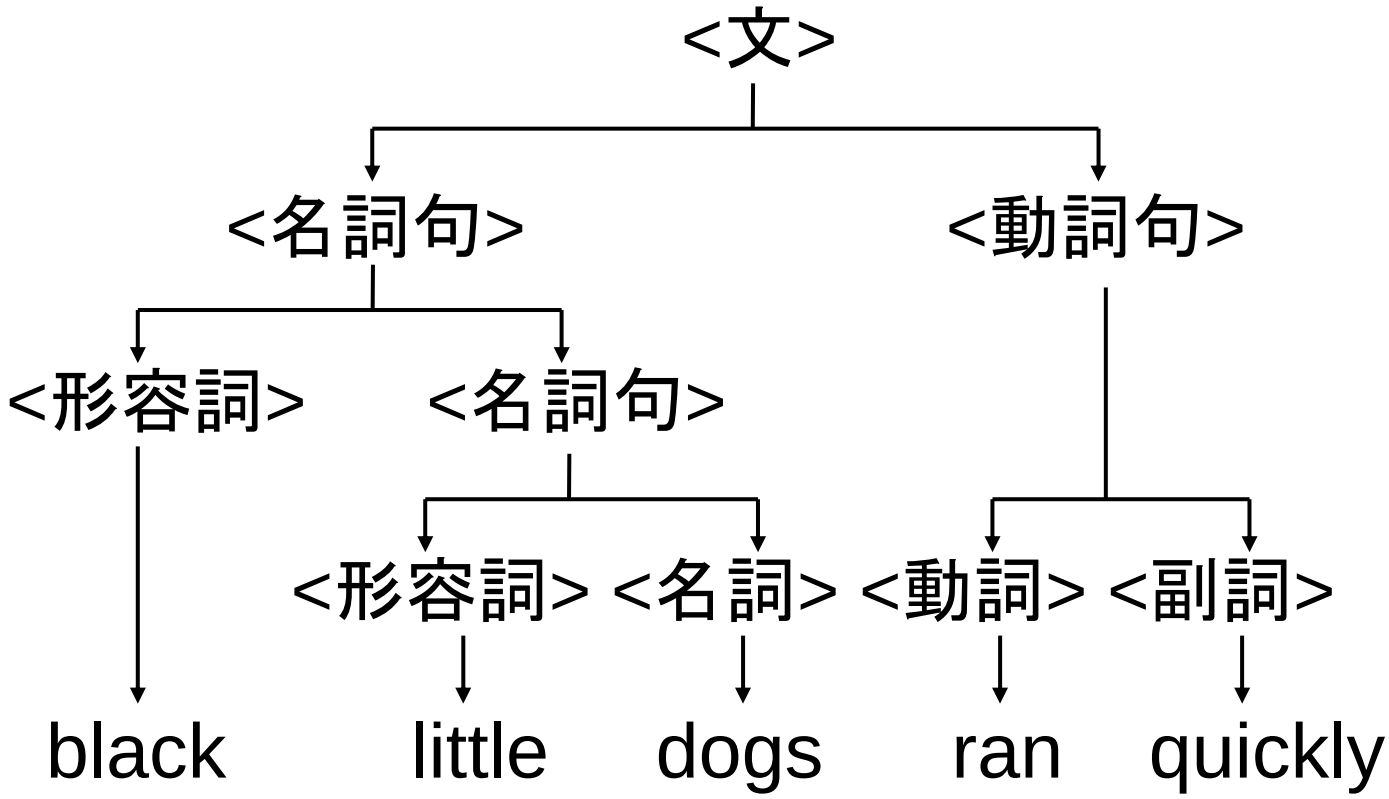
- 下記の導出過程により、”Black little dogs ran quickly”は構文として正しいと受理する

<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

- 上記では、<名詞句>→<形容詞><名詞句>を2回適用していることに注意(再帰的な導出)

構文木による導出過程の表記

- 前スライドのような導出過程はどこにどのルールを適用したか分かりにくい　->構文木による表現



10進数整数のBNFによる定義

- 下記が10進数整数のBNFによる定義
- データ(数値)の羅列の中に、下記の規則に従う部分があれば、10進数整数と判断する
  - 文字列も数字も画像も全部数値で表されている(1章)

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

2.2.2 正規表現

- 文字列の構文を記すのによく使われている表現
- 近年のプログラミング言語でよく用いられる
  - 特にPerl, PHP, Rubyなどのウェブアプリケーション等でよく使われるLight Weight Language
- より便利に使えるように、プログラミング言語で独自に拡張されていることが多い
  - Perlでは大文字、アルファベット、空白を記すルールがある、など

一般的な正規表現の規則

記号	意味
.	任意の1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。

- 「直前のパターン」の繰り返しというものを特によく使います

正規表現の例

- [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
  - “.”は一時的に正規表現のルールから外すものとする
  - マッチする: 0.1, 0.34567
  - マッチしない: 0, 0., 1.1
- [0-9]?.[0-9]+: 以下の両方にマッチする
  - 整数1桁と小数で表記された数字
  - 小数点以下だけで表記された数字

演習

- 正規表現”^[0-9]*[A-Z]*$”に一致するものはどれか？ (注：複数あります、*の定義に注意)
  - ABC123
  - 123ABC
  - 123456
  - ABCDEF

2.3 演算子と式の表現

- 演算子の種類
  - 単項演算子: 1つの数値のみを演算
  - 2項演算子: 2つの数値を演算
  - C言語の3項演算子はとりあえずおいておく
- 2項演算子の表現方法は複数ある
  - 中置記法: 一般的な記法 a+b
  - 前置記法: 演算子を先に置く +ab (ポーランド記法)
  - 後置記法: 演算子を後に置く ab+ (逆ポーランド記法)

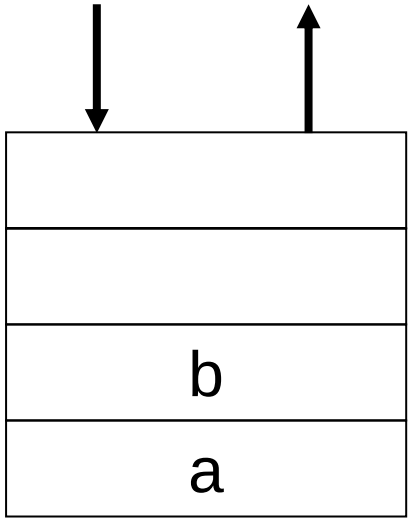
逆ポーランド記法の例

- (a+b)÷c-dを逆ポーランド記法で書こう
- 演算優先度の高い順に
  - (a+b) -> ab+
  - (a+b)÷c -> ab+c÷
  - (a+b)÷c-d -> ab+c÷d-

括弧なしで演算の優先順位を指定できるのは利点

逆ポーランド記法の利点

- 古典的なデータ構造であるスタックで実現容易
  - 下から上に順番にデータを積んで、上から取り出す
  - スタックについての詳細は5章で
- 例：ab+c÷d-
  1. a,bを入れて+を示した時点で演算
  2. スタックの底はab+の結果になる
  3. cを入れて÷を示した時点で演算
  4. スタックの底はab+c÷の結果になる
  5. dを入れて－を示した時点で演算



スタック

2章のまとめ

- 論理演算と論理回路
  - ビットで表現された情報を論理演算で操作できる
  - 論理演算と1対1に対応する論理回路が存在する
  - 論理回路で加算回路等を作成してビットで表現された情報を操作できる
- BNF等の規則で構文や数式を表現できる
  - 逆に、規則に一致すれば求めていた文や数値と考えられる
  - 数式の表現は逆ポーランド記法がコンピュータ業界では有名