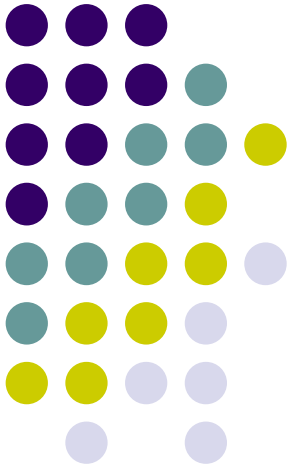
情報処理概論
後半第2回

岩橋政宏



今日の内容

- 復習
- アルゴリズムとプログラムの設計
  - テキスト処理
    - 正規表現

ロボットを制御するプログラムを考える（復習）

ロボットのプログラム (1)

- 動作 = モータの制御
  - モータを○○%の出力でまわす
  - モータを△△回転まわす
- 右，左の車輪にそれぞれ1つずつ
  - 左右を同じ速度でまわす：直進
  - 右か左のどちらかだけをまわす：旋回
  - 一方をゆっくり目にまわす：ゆるい旋回
  - 右と左を逆にまわす：超信地旋回

ロボットのプログラム (2)

- 条件を調べるには = センサ
  - タッチセンサ: ボタンが押されるとONになる
  - 音響センサ: 音が入力されるとONになる
  - 光センサ: 光を当てて，反射光の色や強さに応じた値が出力される
  - 超音波センサ: 超音波により距離を計測し，その距離に応じた値が出力される

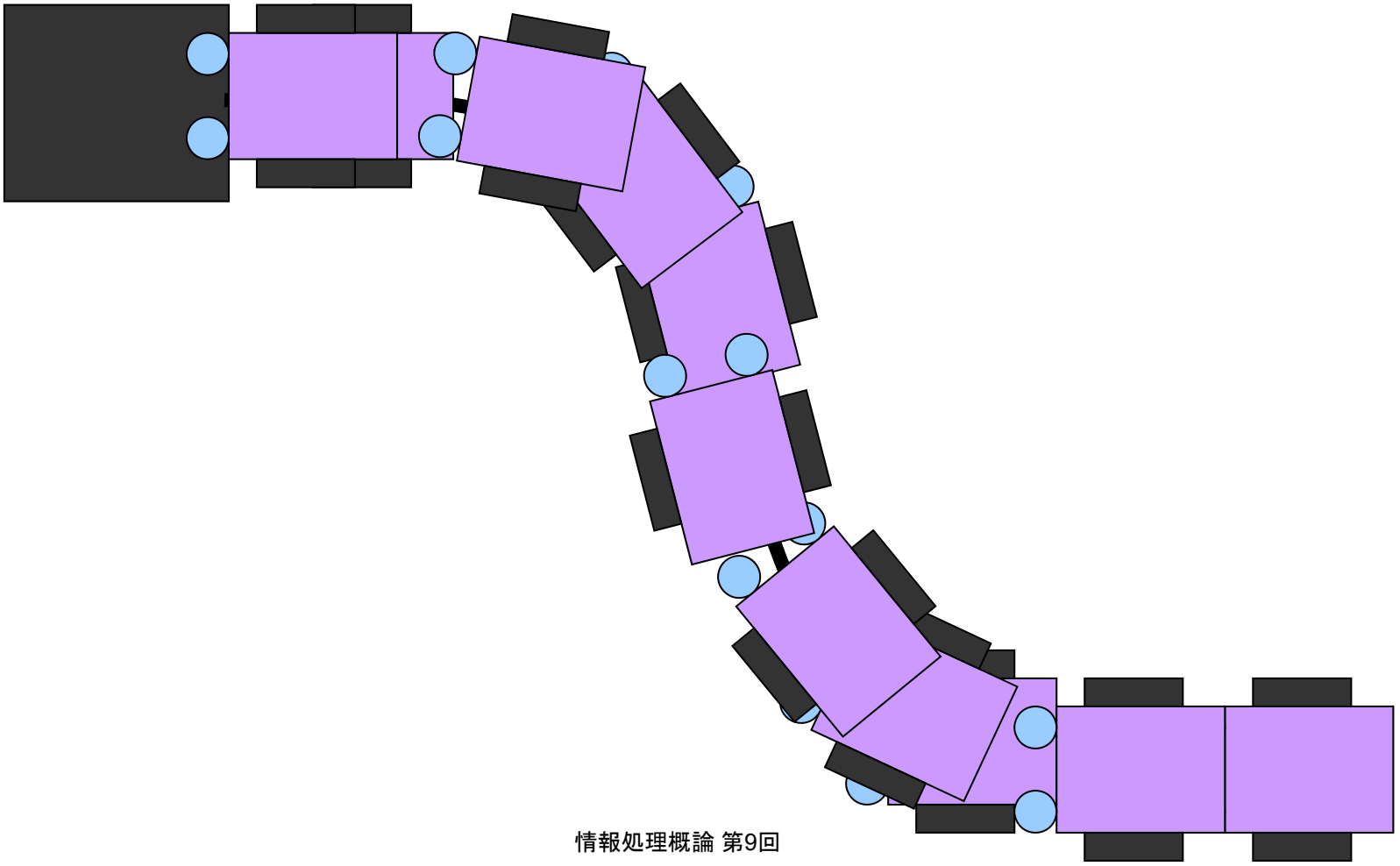
ロボットのプログラム (3)

- センサの出力 (ON/OFFや値) を条件として
  - もしも，〇〇センサが△△ならば，□□をする
  - ××センサが■■になるまで，◎◎を繰返す

といった動作を組合せて

- 思った通りに動作をさせる

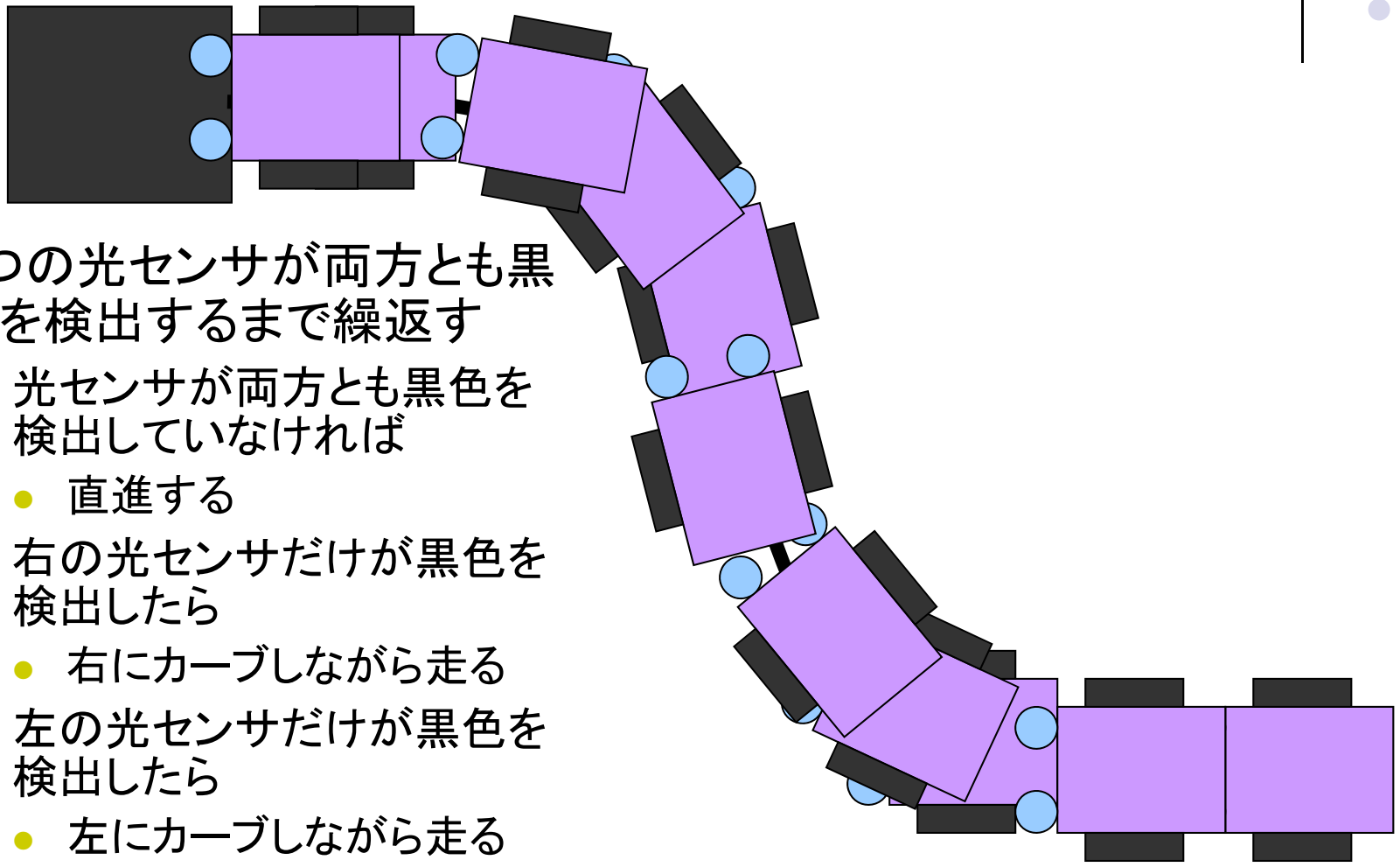
線に沿って走るロボット (ライントレーサ)



ライントレーサのアルゴリズム

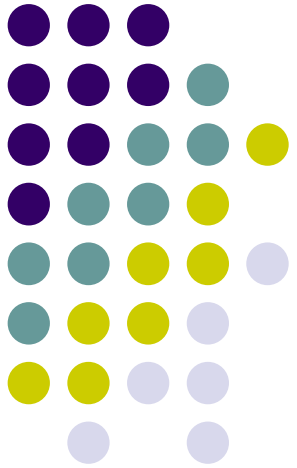
- 2つの光センサが両方とも黒色を検出するまで繰返す
  - 光センサが両方とも黒色を検出していなければ
    - 直進する
  - 右の光センサだけが黒色を検出したら
    - 右にカーブしながら走る
  - 左の光センサだけが黒色を検出したら
    - 左にカーブしながら走る

ライントレーサの動作



- 2つの光センサが両方とも黒色を検出するまで繰返す
  - 光センサが両方とも黒色を検出していなければ
    - 直進する
  - 右の光センサだけが黒色を検出したら
    - 右にカーブしながら走る
  - 左の光センサだけが黒色を検出したら
    - 左にカーブしながら走る

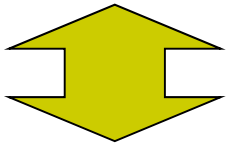
テキスト処理（今日の内容）



コンピュータが活躍する場所

- 数値計算
  - 微分方程式を数値的に解く → シミュレーション
  - 統計処理
  - 会計処理

昔はほとんどこっち



今はこの用途の方が多い

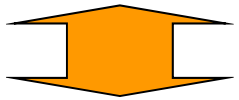
  - 電子メイルを送る/受ける
  - Webページを表示する
  - 知りたいことが書いてあるwebページを探す
- テキスト処理

テキスト処理とは

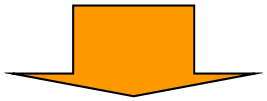
- コンピュータのデータを文字の並び(文字列)として処理する操作
- 文字列は言葉を表すものとして扱うことが多い
  - 英数文字の文字列と空白で区切られている→単語
  - 単語が並んでおり，末尾に“.”がある→文
- 基本的なテキスト操作
  - 切り貼り: 一部を切り出す，二つの文字列をつなぐ
  - **検索: 条件に合う文字列のパターンをさがす**

文字列のパターンをさがす？！

- 文字列そのものをさがす
  ←これでも十分に役に立ちそう
  - 例：教科書の中から，「レジスタ」という文字列が含まれた箇所を探し出す



  - 例：文書の中から何らかの日付が書かれた箇所を探し出す．
    日付は「2009年06月12日」といった形式で書かれている．
- 日付の記述を探したいが，何年何月何日でも良い(事前に知らない)



パターン：nnnn年nn月nn日

- パターンとして探し出せることが必要

Googleでさがしてみた例(1)

- 問合せ: テキスト処理

ウェブ　検索ツールを表示　　テキスト処理 の検索結果 約

他のキーワード: ruby テキスト処理　python テキスト処理　linux テキスト処理　unix テキスト処理　java テキスト処理

テキスト処理のお手伝い

一方、表計算ソフトのデータをテキストファイルとして書き出したファイルは人間が読み書きする文書では無い場合(単なる数字データの羅列など)も多いですが、他のOSでも利用でき、テキストエディタで処理できるため、「テキストファイル」と呼ばれて ...

work.tkensaku.com/text.html - キャッシュ - 類似ページ -

perlによるテキスト処理

perlは非常に高機能な言語でテキスト処理のみならず、バイナリデータを扱ったり、プロセス間通信を利用できたりするため、幅広いユーザが利用しています。awkで書けるプログラムは必ずperlで書けますし、awkでは不可能な処理もperlでは可能なので、...

work.tkensaku.com/PERL/perl.html - キャッシュ - 類似ページ -

work.tkensaku.com からの検索結果 »

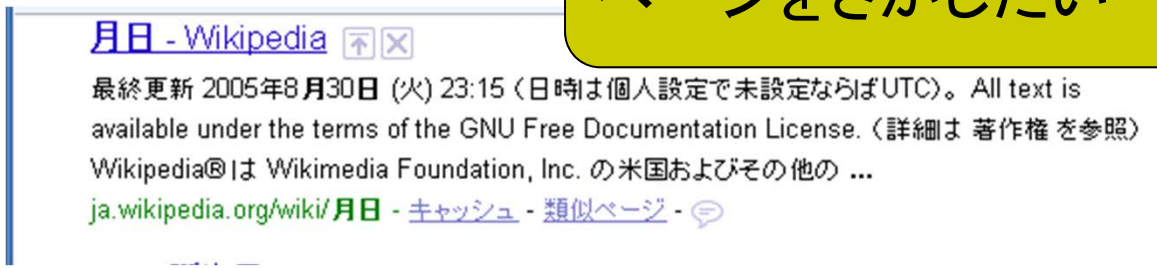
テキストエディタ (パソコン便利ツール集) フリー

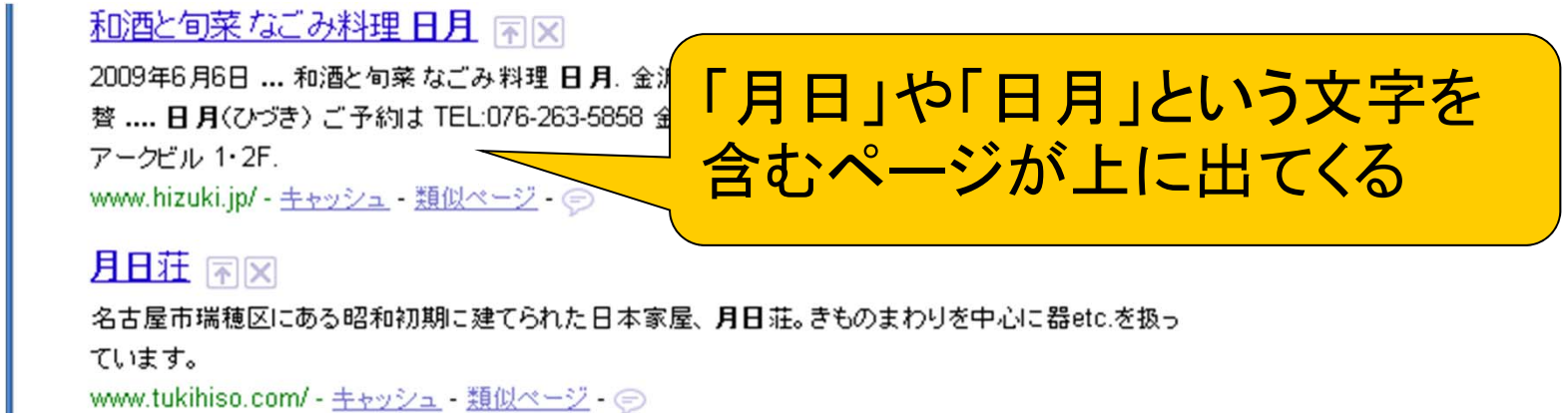
【メニュー閉じる:テキスト処理】テキスト処理のトップ テキスト・エディタ・Emエディタ・OEdit・サク

Googleでさがしてみた例(2)

- 問合せ：月 日

〇月〇日という表記を含むページをさがしたい





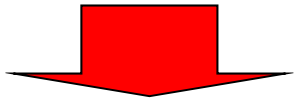
「月日」や「日月」という文字を含むページが上に出てくる

探したいパターンを
どうコンピュータに伝えるか

簡単!!

- 文字列そのものを探したい場合
  → 探したい文字列をコンピュータに入力

- パターンを探したい
  → どのようなパターンを探すかをコンピュータに指示する必要あり



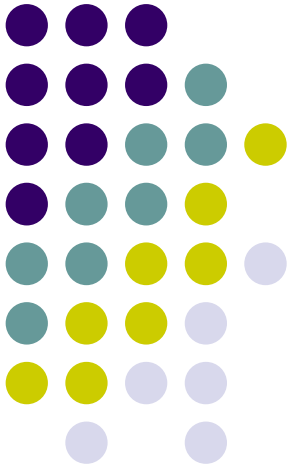
- コンピュータに指示するためのパターンの表現方法が必要

非常に原始的なパターンの指定

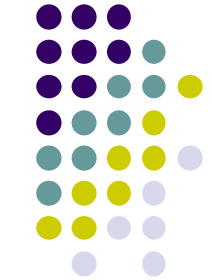
- ラ〇ン：〇の中には任意の1文字が入る
  - ラテン，ラタン，ラドン等がこのパターンにマッチ
- ラ__ン：__には任意の文字(1文字以上)が入る
  - 上記に加えて，ラーメン，ライトバン，ラストラン等がマッチ

より複雑なパターンを厳密に指定できる方法が欲しい

正規表現



正規表現とは?



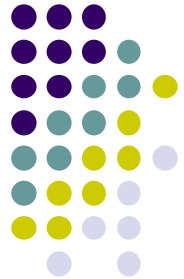
- 特定の文字列ではなく，文字列の一部を一般化して表現するための手法
- もともとはコンピュータ言語理論の分野において，字句（変数名や予約語、その他の識別子）を一般化して定義するために考案された表現手法
- 通常のコンピュータ利用では，ドキュメントからの文字列検索時などに，検索したい文字列すべてを指定するのではなく，文字列の任意の一部を置き換え可能な状態で検索する場合などに用いる

(アスキー24 デジタル用語辞典より)

正規表現(概略)

- 英文字，数字，_: その文字にマッチ
- . : 任意の一文字にマッチ
- ? : 直前の文字が0回か1回ある場合にマッチ
- + : 直前の文字が1回以上ある場合にマッチ
- *: 直前の文字が0回以上ある場合にマッチ
- | : 右のパターンか左のパターンのどちらかがある場合にマッチ
- ^: 行の先頭にマッチ
- $: 行の末尾にマッチ
- [ ]: 括弧内に列挙した文字のどれかにマッチ，
  - を用いて範囲指定可
- ( ): パターンのグループ化

正規表現の利用例



- 2003-04-25といった形式の日付にマッチ
  - 厳密版: [0-9][0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9]
  - 大雑把版: [0-9]+-[0-9]+-[0-9]+
- 日本のメイルアドレスにマッチ
  - [0-9a-zA-Z_-]+@([0-9a-zA-Z-]+¥.)+jp
- JapanとJapaneseのどちらにもマッチ
  - Japan(ese)?
- zooooooooooooooooooooom(oは2個以上)にマッチ
  - zooo*m

正規表現の詳細

- 正規表現には様々な版がある
  - 基本は同じだが，いろいろな拡張がなされている
- ここでは，perl (ver.5.005) における正規表現を取り上げる
  - Version 8 regexpルーチンで提供されているものと同等
  - perlの文法上の制約による若干の相違あり

メタ文字

- 正規表現に記述される文字は2種類ある
  - メタ文字
    - 正規表現の上で特殊な意味を持っている文字
  - それ以外＝普通の文字
    - 普通の文字は，その文字そのものにマッチする
    - 普通の文字の並びは，対象文字列の同じ文字の並びにマッチする
      - "blurfl" という文字の並びは，対象文字列の中の"blurfl"という文字の並びにマッチする

任意の1文字にマッチ

- .
  - 任意の1文字にマッチ
  - ただし，改行は除く

文字クラス

- [ ]
  - そこに含まれる文字のいずれかにマッチ
    [abcde]: a,b,c,d,eのいずれかにマッチ
  - "["の直後に"^"があると，そこに含まれる文字以外の文字のいずれかにマッチ
    [^abcde]: a,b,c,d,e以外のいずれかの文字にマッチ
  - "-"により範囲を指定できる
    [a-z]: aからzの間の任意の文字
    [0-9a-zA-Z]: 数字，英大文字，英小文字
  - "-"そのものを文字クラスに含めたい時は，先頭または末尾に置くか，"¥"によりクオートする
    - [-az], [az-], [a¥-z]はすべて等価("a","z","-"のいずれかにマッチ)

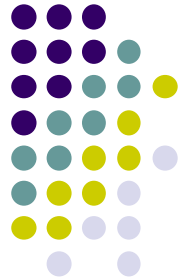
量限定子

- *
  - このメタ文字の直前に置かれている部分正規表現の0回以上の繰り返しにマッチ
    - "fo* "は "fo"にも"foo"にもマッチし，"f" (oがひとつもない)にもマッチする
- +
  - 1回以上の繰り返しにマッチ
    - "fo+"は"fo"にも"foo"にもマッチするが，"f"にはマッチしない
- ?
  - 0回または1回の生起にマッチ
    - "fo?"は"f"または"fo"にマッチする

量限定子 (続き)

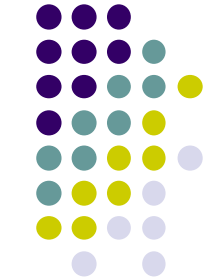
- {n}
  - ちょうどn回の繰り返しにマッチ
    - "fo{2}"は"foo"のみにマッチする
- {n,}
  - n回以上の繰り返しにマッチ
    - "fo{2,}"は"foo", "fooo", "foooo", . . . にマッチする
- {n,m}
  - n回以上，m回以下の繰り返しにマッチ
    - "fo{2,3}"は"foo"と"fooo"にマッチする
- "*"は{0,}と等価，"+"は{1,}と等価，"?"は{0,1}と等価

non-greedyな量限定子



- 前述の量限定子のマッチはgreedyである
  - 可能なかぎり多い回数の繰返しにマッチしようとする
- 可能な限り少ない回数の繰返しにマッチするnon-greedyな量限定子→?を付加
  - *?
  - +?
  - ??
  - {n}?
  - {n,}?
  - {n,m}?

greedyなマッチと non-greedyなマッチ



- 対象文字列 "zooom" とする
- 正規表現 "zo+o+m"
  - 1番目の"o+"に"oo"がマッチ
  - 2番目の"o+"に"o"がマッチ
- 正規表現 "zo+?o+m"
  - "o+?"に"o"がマッチ
  - "o+"に"oo"がマッチ
- 正規表現 "zo*o*m"
  - 1番目の"o*"に"ooo"がマッチ
  - 2番目の"o*"に""がマッチ
- 正規表現 "zo*?o*m"
  - "o*?"に""がマッチ
  - "o*"に"ooo"がマッチ

選択

- |
  - このメタ文字の左右におかれたパターンのいずれかにマッチ
  - 最初の選択肢は、それまでの最後に現れたパターンデリミタ(“(”、“[”、パターンの先頭)から最初の“|”までの全てを含む
  - 最後の選択肢は最後の“|”から次に現れるパターンデリミタまでのすべてを含む
    - "fee|fie|foe"は対象文字列中の ``fee'', ``fie'', ``foe''のいずれにもマッチする
  - ブラケットと一緒に使ったときにはリテラルとして解釈される
    - "[fee|fie|foe]"と記述した場合には [feio|]にのみマッチする
  - 選択の開始点と終点に関する間違いを減らすために，選択を括弧(後述)の中に入れることが多い

括弧によるサブパターンの指定

- ( )
  - "("と")"で囲むことにより、ある一部分の文字列や正規表現を ひとまとめに扱うことができる
    - "(bang)+"は、"bang", "bangbang", "bangbangbang"にマッチする
  - 選択の始点と終点を制御する場合にも( )で囲む
    - 前の例"fee|fie|foe"は()を使って"f(e|i|o)e"とも書ける
- ¥1,¥2,¥3,...
  - 先行して現れた ( ) で囲まれたサブパターンに実際にマッチしている文字列を表す
  - ¥に続く数字は、正規表現全体の中でそれが現れた順番を示す
  - 数字部分が0で始まってはいけない(8進数値との区別がつかないため)

ゼロ幅表明

- 文字に対するマッチではなく，文字と文字の間や先頭，末尾などの位置に対するマッチ
  - ^: 行頭にマッチ
  - $: 行末にマッチ
  - ¥b: 単語境界にマッチ (片方が¥wでもう一方が¥Wである文字の間)
  - ¥B: 単語境界以外にマッチ
  - ¥A: 文字列の先頭にマッチ
  - ¥Z:文字列の終端もしくは終端にある改行の直前にマッチ
  - ¥z:文字列の終端にのみマッチ

メタ文字とクオート

- 通常，メタ文字に"¥"を前置する(クオートする)ことで，普通の文字として解釈されるようにできる
  - ". "(ドット)は任意の1文字にマッチするという意味を持つメタ文字だが，"¥. "と書くとドットそのものにマッチする普通の文字として解釈される
  - "¥¥"は"¥"そのものにマッチする

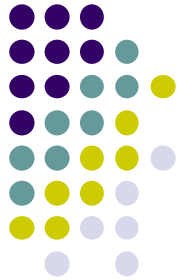
メタ文字構文による文字の指定

- (通常はunprintableな) 1つの文字をCで用いられるようなメタ文字構文により指定できる
  - ¥n: 改行
  - ¥r: リターン
  - ¥t: タブ
  - ¥f: 改ページ
  - ¥a: アラーム(ベル)
  - ¥b: バックスペース(文字クラスの中でのみ)
  - ¥e: エスケープ
  - ¥nnn: 文字コードとしてnnn(8進数字)の値を持つASCII文字
  - ¥xnn: 文字コードとしてnn(16進数字)の値を持つASCII文字
  - ¥Cx: ASCII文字におけるコントロール-x

メタ文字構文による文字の指定(続き)

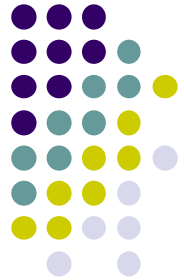
- さらに，この構文により指定できる文字クラスもある
  - ¥w: "単語"文字(アルファベット，数字，"_")にマッチ
  - ¥W: 単語文字にないものにマッチ
  - ¥s: 空白文字にマッチ
  - ¥S: 非空白文字にマッチ
  - ¥d: 数字にマッチ
  - ¥D: 非数字にマッチ
    - 注: 例えば"¥w"は単語文字の1文字にマッチするのであり，単語全体にマッチするのではない．単語全体にマッチさせるには"¥w+"と書く必要がある
- [ ]による文字クラスの指定(後述)に含めることもできる

例



- 日付とマッチ: 次のような様々な形式とマッチする正規表現は?
  - 98-5-2
  - 2003-5-2
  - 2003-05-02
  - 2003/5/2
  - 2003.5.2
  - 年は2桁か4桁の数字
  - 年，月，日の順
  - 年，月，日の区切りは，"-"か"/"か"."
  - 月，日は，1桁か2桁の数字
  - 月，日が1～9の場合に十の桁に0がつく場合もつかない場合もある

例 (続き)

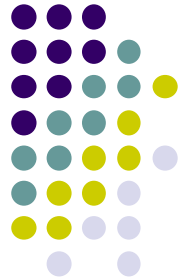


- 答え

(¥d{2}|¥d{4})[-/¥.]¥d{1,2}[-/¥.]¥d{1,2}

- (¥d{2}|¥d{4}): 年にマッチ，2桁か4桁の数字
- [-/¥.]: 区切りにマッチ，"-"か"/"か"."
- ¥d{1,2}: 月や日にマッチ，1桁か2桁の数字

例 (続き)

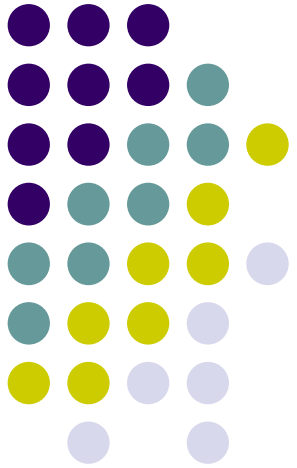


- 答え

(¥d{2}|¥d{4})([-/¥.])¥d{1,2}¥2¥d{1,2}

- (¥d{2}|¥d{4}): 年にマッチ，2桁か4桁の数字
- [-/¥.]: 区切りにマッチ，"-"か"/"か"."
- ¥d{1,2}: 月や日にマッチ，1桁か2桁の数字

プログラムの制御構造



[復習] プログラム

- コンピュータに対する指令書
- 原則として，コンピュータは書かれた命令を順番に忠実に実行
- 例：

```
A=10;
B=20;
C=A+B;
printf(”%d¥n”, C);
```

[復習] プログラム（続き）

- 書かれた命令を順番（上から下）に忠実に実行するだけでも，それなりに役に立つが．．．
- 例：１～5まで足すプログラム

```
SUM=1;
SUM=SUM+2;
SUM=SUM+3;
SUM=SUM+4
SUM=SUM+5
printf(”%d¥n”, SUM);
```

[復習] プログラム（続き）

- 例：1～100まで足すプログラム

  SUM=1;

  SUM=SUM+2;

  …

  SUM=SUM+99

  SUM=SUM+100

  printf(”%d¥n”, SUM);

- 1～N まで足すプログラム（Nはキーボードから入力して与える）

[復習] プログラム（続き）

- 1～N まで足すプログラム（Nはキーボードから入力して与える）

？？？

[復習] 単に上から下に実行するだけではダメで．．．

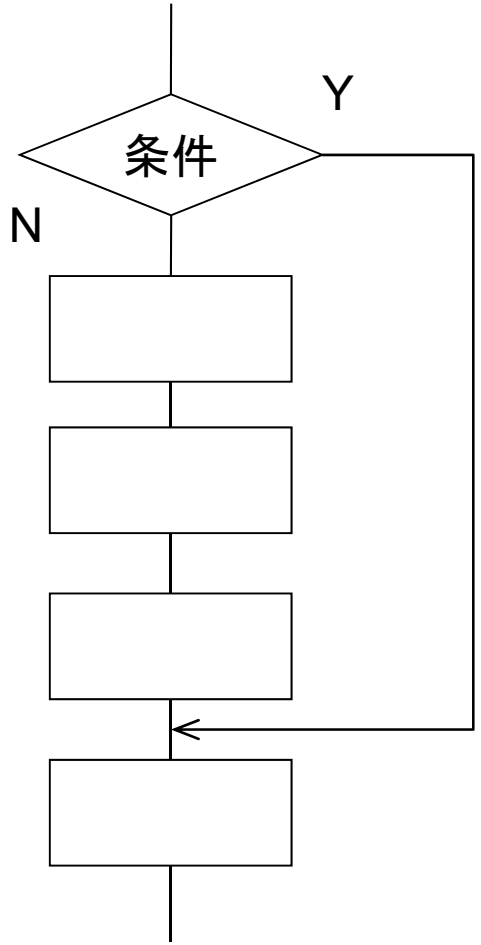
- ある（指定した）条件に基づいて，
  - 条件が成り立っていたら〇〇をする
  - 条件が成り立っていなかったら△△をする
- 指定した回数or指定した状態になるまで，〇〇といった処理を繰り返す

「制御構造」という

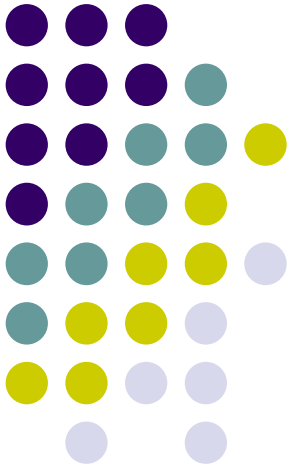
という動作をすると，圧倒的にいろんなことができるようになる

[復習] もっとも原始的な制御構造

- ある条件が
  - 成り立っていたら：指定された番地に飛ぶ
  - 成り立っていなかったら：何事もなかったかのように次（すぐ下）に書いてある命令を実行
- これをうまく組合せれば，前述のことは全部可能



[復習] 繰り返し



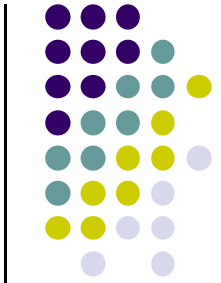
[復習] 繰り返し

- 同じ計算式を用いて，すこしずつ変数の値を変えて計算した結果を求めたい
  - 数種類の変数値ならば計算するコードを羅列すればできる
  - 非常に多くの様々な変数値に対して計算したい場合は?
  - プログラム動作中に計算したい変数値の範囲が決まる場合?
- 繰り返しというメカニズムにより行う

[復習] 繰り返し (続き)

- 例題
  - 1.0から10.0までの実数値に対し
  - その2倍の値を計算し
  - 元の値と計算された値を並べて出力するプログラム
  - ただし，きざみは1.0とする

[復習] 繰り返し (続き)



- Pascal

```
program Double;
var
    d, x: real;
begin
x := 1.0;
while x <= 10.0 do
 begin
 d := 2.0 * x;
 writeln(x:6:2,d:6:2);
 x := x + 1.0;
 end;
end.
```

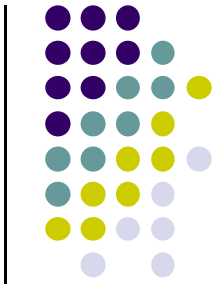
- C

```
main() {
 float d, x;
 x = 1.0;
 while(x <= 10.0) {
  d = 2.0 * x;
  printf("%6.2f,%6.2f¥n", x, d);
  x += 1.0;
 }
}
```

[復習] 繰り返しにおける動作

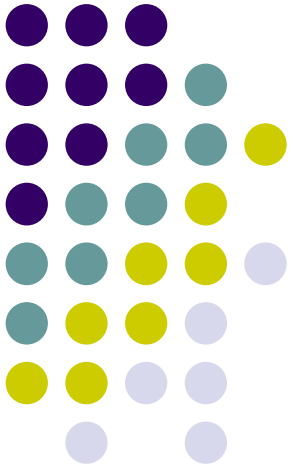
命令		x	d
x =1.0	1.0		
(test) x <= 10.0	(true)	1.0	
d = 2.0 * x		1.0	2.0
printf		1.0	2.0
x = x + 1.0		2.0	2.0
(test) x <= 10.0	(true)	2.0	2.0
d = 2.0 * x		2.0	4.0
printf		2.0	4.0

[復習] 繰り返しにおける動作(続き)

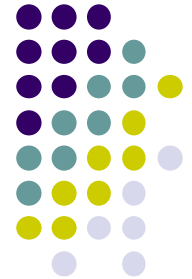


命令		x	d
:		:	:
x = x + 1.0		10.0	18.0
(test) x <= 10.0	(true)	10.0	18.0
d = 2.0 * x		10.0	20.0
printf		10.0	20.0
x = x + 1.0		11.0	20.0
(test) x <= 10.0	(false)	11.0	20.0
(終了)			

トップダウンプログラミング



複雑な問題に対するプログラム

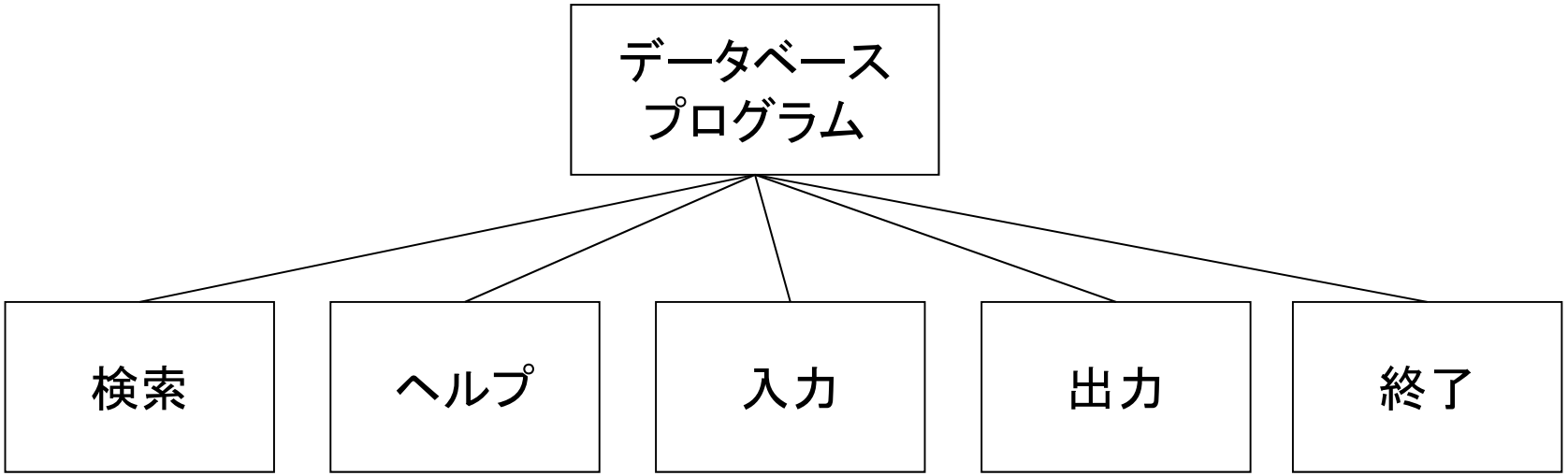


- コンピュータ科学の中心課題は複雑さとその処理
  - 単純な問題は実はコンピュータを使うまでもない
- 高度に複雑な問題をうまく処理する方法
  - 問題を扱いやすくできるよう表現する
  - トップダウンアプローチ/段階的詳細化
    - 問題をより簡単なサブタスクに分解し
    - さらに簡単に理解できるレベルにまで各サブタスクを繰り返し分解する

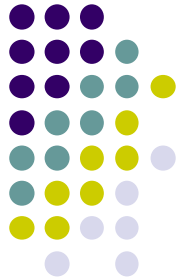
サブタスクへの分解の例

- 入力したデータを格納しておき
- 格納されたデータの一覧を出力したり
- 問合せに対して適合するデータを検索して出力する

プログラム=データベースプログラム



サブタスクでの処理



- ヘルプ: ヘルプを表示する
- 終了: プログラムを終了する
- 入力:
  - メモリ内に新しい記憶場所を確保する
  - 入力された新しいデータをそこに格納する
- 出力:
  - データが入っている記憶場所ごとに
  - その内容を出力する

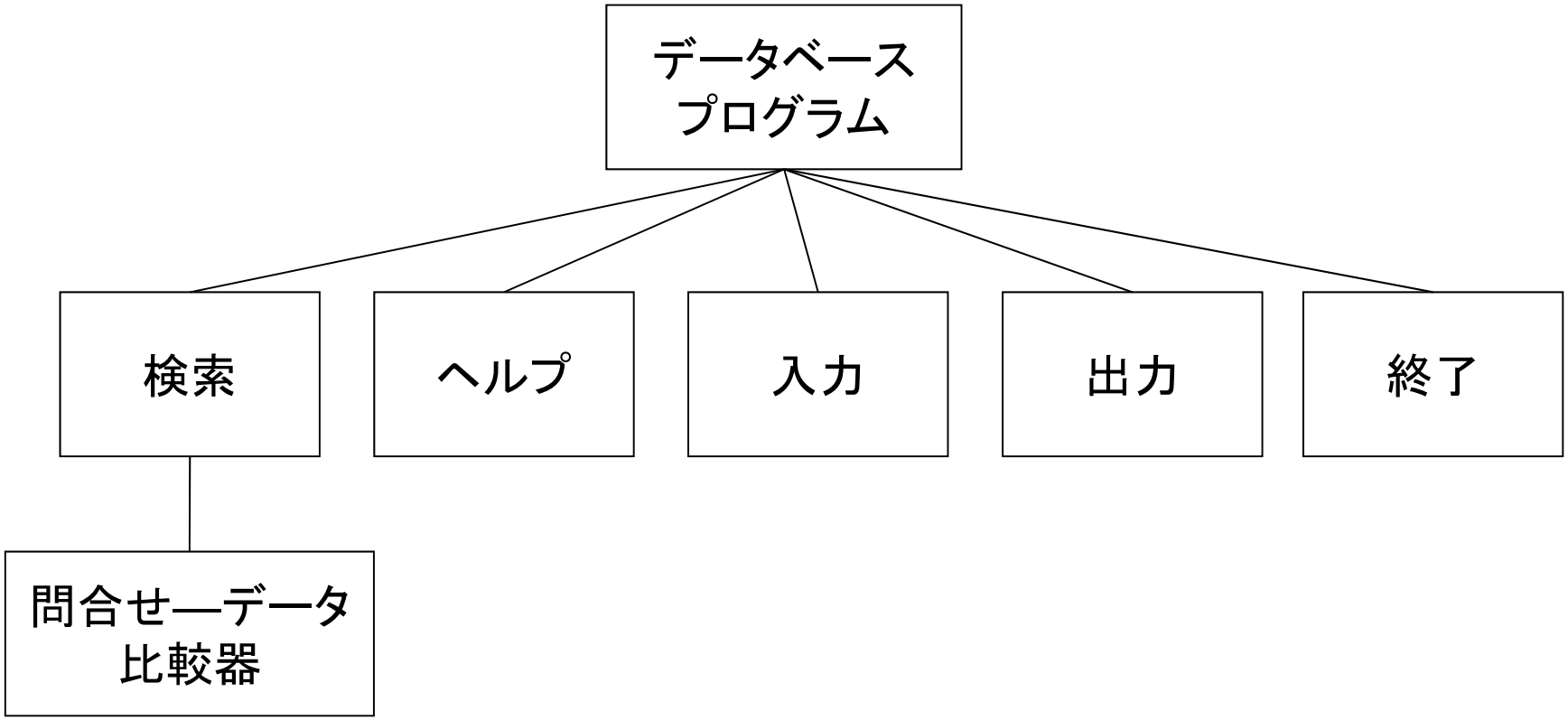
サブタスクでの処理 (続き)

- 検索:
  - ユーザからの問合せを受付ける
  - データが入っている各記憶領域を検索し
    - 問合わせに適合するデータであれば，それを出力する

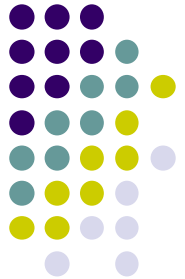
↓まだ少し複雑すぎるので．．．

- 検索
  - ユーザからの問合せを受付ける
  - データが入っている各記憶領域を検索し
    - 問合せ—データ比較器を呼び出す
    - 比較機が「match」と報告すれば
      - そのデータを出力する

さらなるサブタスクへの分解の例



サブルーチン



- 問題をサブタスクに分解

↓

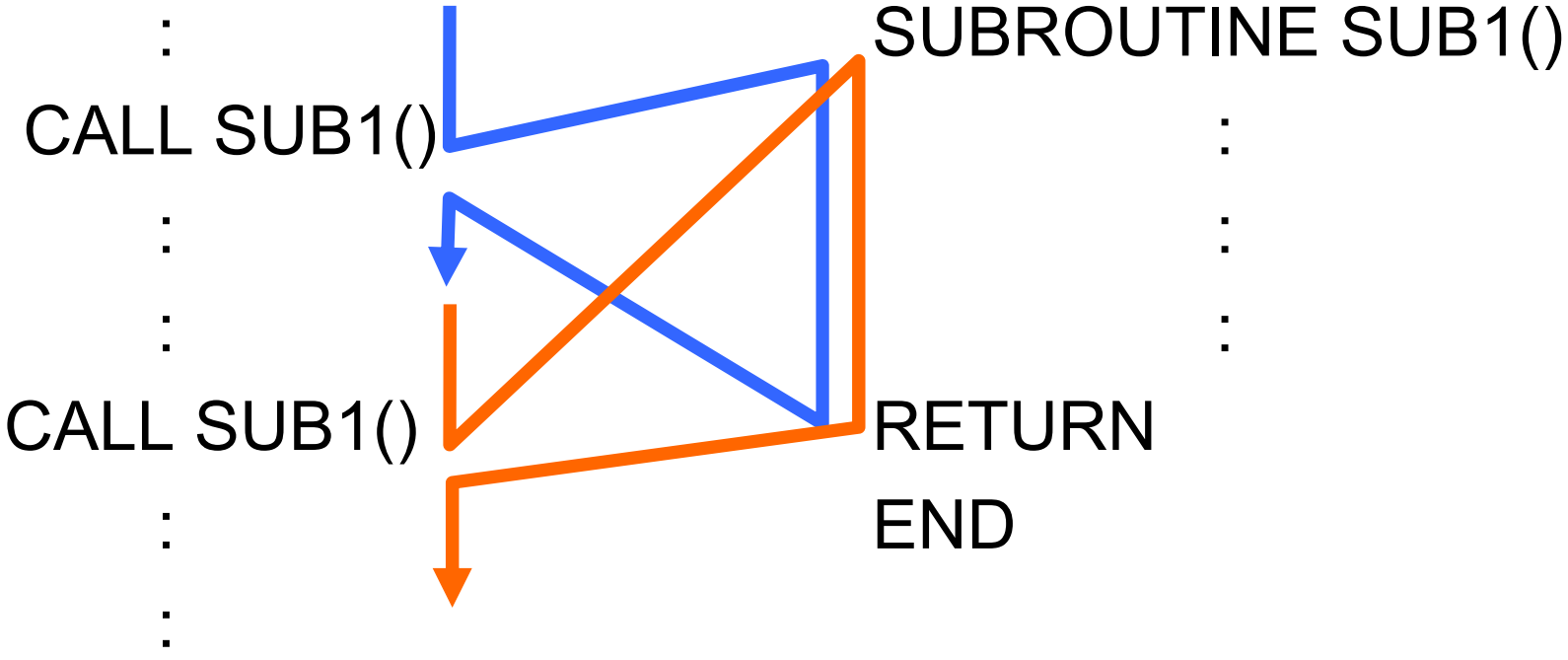
- プログラムもサブタスクごとに記述したい

↓

- サブタスクをひとつのまとまったプログラムとして記述
- 各サブタスクのプログラムを必要に応じて呼び出すプログラムを作ってプログラムを完成させる

- サブタスクのプログラム: サブルーチン
- サブルーチンを呼び出すプログラム: メインルーチン
- サブルーチンを呼び出すこと: サブルーチン呼出し，サブルーチンコール

サブルーチンコール



- CALLがあると制御が対応するサブルーチンに移る
- RETURNがあると呼んだCALL文の直後に制御が移る

今日はここまで
次回の予告

プログラミング – 再帰