情報科学(12)

いろいろなプログラミング言語

今週の内容

- 目標: 「実際」のプログラミング言語について知る
- 理由:
  - 世の中には沢山のプログラミング言語がある
  - プログラムの書き方・実行方法・得手不得手は言語や処理系によって様々
  - Rubyはその一つに過ぎない
- 項目:
  - プログラミング言語の歴史
  - 代表的なプログラミング言語の特徴
  - C言語の紹介～プログラムの書き方・実行方法
  - アセンブリ言語と機械語

歴史: プログラミング言語以前

- コンピュータ発明当初は、人間が機械語のプログラムを直接書いていた
  - 機械語: CPUが理解できる命令の並び (命令=メモリ上のデータ)
  - アセンブリ言語: 機械語の命令を人間が分かるような単語に置き換えたもの (機械語命令と1対1に対応)
- 書く人はものすごく大変
  - 機械語は単純な命令しかない

  →簡単な式の計算にも命令を沢山並べなければいけない
- 互換性がない
  - 新しいCPUを持つコンピュータでプログラムを動かしたい

  → CPUが違うと機械語命令も違う

  → 機械語プログラムの書き直し!

(後で実際のアセンブリ言語プログラムを見る)

歴史:
プログラミング言語の誕生と発展

開発・公表された年代

年代	代表的な言語	特徴
’50s	FORTRAN, COBOL, LISP	(現存する)最も初期のプログラミング言語が作られる
’60s–’70s	Simula, BASIC, Pascal, Smalltalk, C, Prolog, ML	オブジェクト指向・論理型・関数型など新しい考え方をとり入れた言語が作られる
’80s–’90s	C++, Perl, Ruby, Python, Java, JavaScript, C#	色々な用途向きに発展してゆくいまどきの主流になりつつある

多くの言語は、開発から普及までに10年くらいかかっている

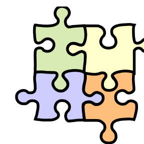
プログラミング言語の種類

分け方の軸は色々ある･厳密なものではない

- 命令型言語と宣言型言語
- オブジェクト指向言語
- スクリプト言語

命令型言語と宣言型言語

- 命令型(imperative)言語
  - 手続き型(procedural)言語とも言う
  - コンピュータが行うべき動作を順に「命令」
    - 基本要素: 変数・代入・制御構造・関数(メソッド)など
    - 機械語, Ruby, C, C++, Java等はすべて命令型言語
- 宣言型(declarative)言語
  - プログラムは物事の性質や関係を記述し、
    コンピュータに答えを探させる
  - 問題解決の試行錯誤段階などでよく用いられる
  - 具体的には論理型言語(Prologなど)・関数型言語(ML, Haskellなど)

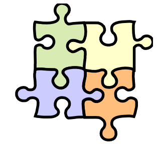


宣言型言語の例:
関数型プログラミング言語

- 素数列を求めるHaskellプログラムの例

```
primes = sieve [2..]
 where sieve (p:xs)
         = p : sieve [ x | x <- xs, x `mod` p > 0 ]
```

意味:
- 素数は2から始まる数列をふるいにかけたもの
- 先頭がp、残りがxsの数列をふるいにかけた数列とは、先頭がpで、〈残りの数列からpで割り切れる要素を除いた列〉をふるいにかけたもの



宣言型言語の例:
論理プログラミング言語

Prologによるプログラムの例

プログラムの例

```
mother(hatoyamayasuko,hatoyamayukio).      %鳩山安子は鳩山由紀夫の母
father(hatoyamaiichiro,hatoyamayukio).     %鳩山威一郎は鳩山由紀夫の父
father(hatoyamaiichiro,hatoyamakunio).     %鳩山威一郎は鳩山邦夫の父
father(ishibashishojiro,hatoyamayasuko).   %石橋正二郎は鳩山安子の父
parent(X,Y) :- father(X,Y).                %XがYの父ならばXはYの親
parent(X,Y) :- mother(X,Y).                %XがYの母ならばXはYの親
sibling(X,Y)                               %ZがXの親でZがYの親ならば
   :- parent(Z,X), parent(Z,Y).            %XはYの兄弟
grandparent(X,Y)                           %XがZの親でZがYの親ならば
   :- parent(X,Z), parent(Z,Y).            %XはYの祖父母
```

実行例

```
?- sibling(hatoyamayukio,hatoyamakunio).   %鳩山由紀夫と鳩山邦夫は兄弟か?
Yes                                        (答)YES
?- grandparent(X,hatoyamayukio).           %鳩山由紀夫の祖父母は誰?
X = ishibashishojiro                       (答)石橋正二郎
```

論理的な正しさをチェックしたり、
正しくなるような答を探してくれる

オブジェクト指向言語

- 「もの」を中心にプログラムを構成するタイプの言語
  - 命令型・宣言型とは直交する
  - 基本要素: クラス・メソッド・継承・など
- 特徴
  - ライブラリ(よく使われるプログラム集)が充実しているものが多い
  - 特にGUIのように部品を組合わせるときに活躍
  - 大規模なソフトウェア開発で用いられることが多い
- Smalltalk, C++, Java, Ruby, Python等

スクリプト言語

- 簡単な処理を簡単に書けるように工夫された言語
  - 名前の由来: 複雑な処理をするプログラム群を制御する台本(script)を書くための言語
  - プログラムの起動や文字列処理
- 特徴
  - 処理系はインタプリタ実行形式(後述)のものが多い
    ←互換性や即座に実行できるようにするため
    →実行速度はあまり速くない
  - WWWアプリケーションなどでよく用いられる
- sh, Ruby, Perl, Python, JavaScript等

プログラムの実行のされ方

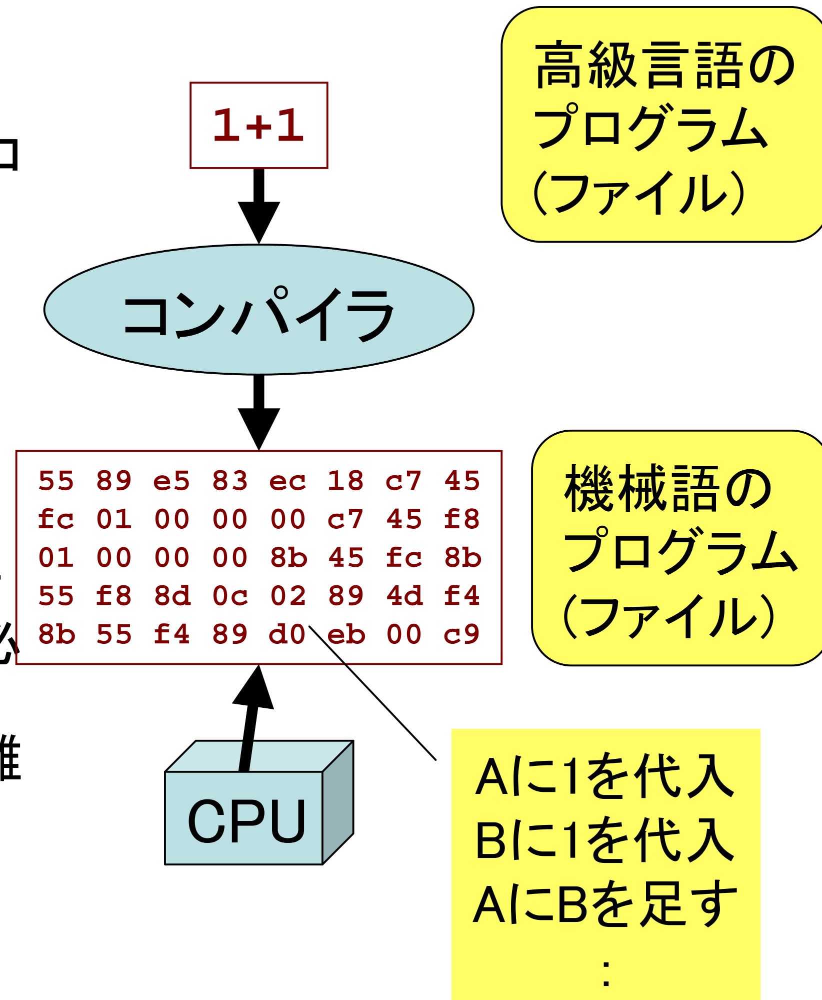
- CPUが実行できるのは機械語の命令だけなので、高級言語のプログラムは間接的に実行される
- 大きく二通りの方法がある
  - ➢コンパイル実行 (本格的なものはこっちが多い)
  - ➢インタプリタ実行 (Rubyはどちらかと言うとこっち)

  両者の中間的なものもある
- 「1+1」というプログラムはどのようにして実行されるのだろうか?

どのやり方をとるかは
言語ではなく
言語<u>処理系</u>による

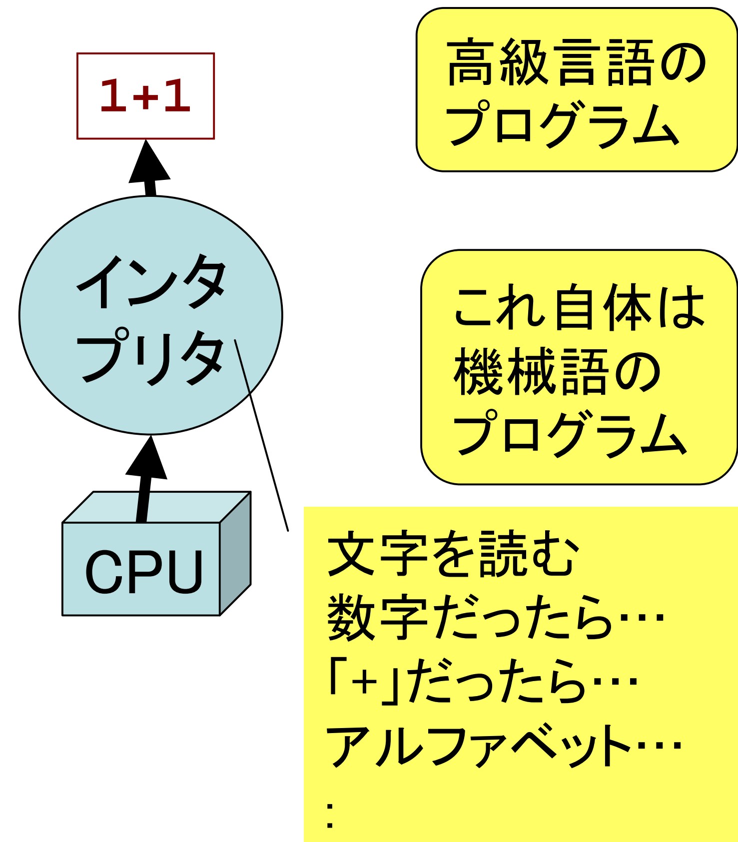
コンパイル実行方式

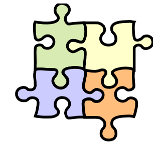
- 手順
  - プログラム全体をまず機械語プログラムに変換
  - 機械語プログラムはCPUが直接実行する
- 特徴
  - 実行は高速
  - 実行までの手間がかかる
  - CPUの違うコンピュータで動かすためにはコンパイルをやり直す必要がある
  - 誤りがあるときに発見するのが難しい
- FORTRAN, C, C++などの言語処理系はほとんどこの方式



インタプリタ実行方式

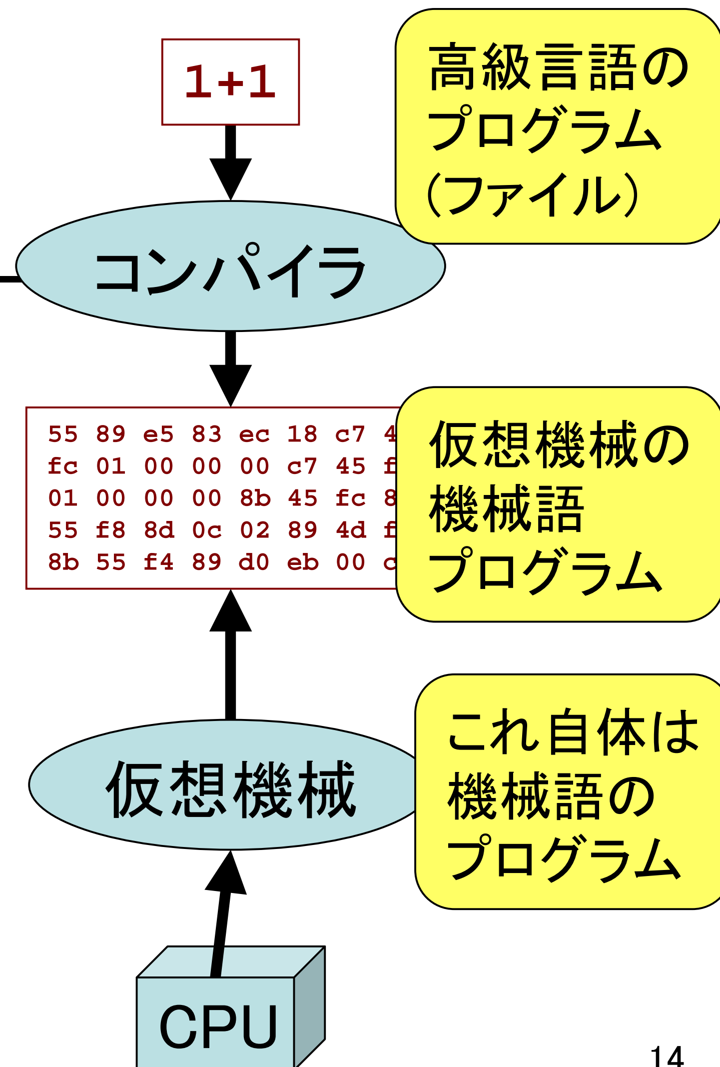
- 手順:
  - CPUはインタプリタというプログラムを実行
  - インタプリタが行う計算=高級言語プログラムの命令を一つ読んではそれに応じた処理を実行
- 特徴:
  - 実行速度は遅い
    ←間接的な実行のため
  - プログラムを即座に実行できる
  - CPUごとにインタプリタを用意しておけば、同じプログラムが色々なコンピュータで実行できる
- 学習用の処理系やスクリプト言語処理系に多い (Rubyもこれ)



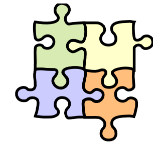


仮想機械による実行方式

- コンパイル実行とインタプリタ実行の組み合わせ
- 仮想機械 = 仮想的なコンピュータのシミュレータ
- 特徴
  - 仮想機械があれば同じ機械語プログラムが色々なコンピュータで動く
  - インタプリタ実行よりも速い・必要メモリ量が小さい
- Pascal, Smalltalk, Javaなど



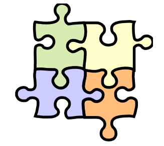
代表的なプログラミング言語: FORTRAN



- 1950年代に開発された
  - アセンブリ言語が主流だった時代
- 科学技術計算プログラムに用いられることが(今でも)多い
  - 高性能なコンパイラがあったため
  - 高性能な数値計算ライブラリが豊富なため
    ——行列計算・統計演算など

代表的なプログラミング言語: C

- 1970年代初頭にUNIX OSとそのアプリケーションの開発のために作られた
  - ハードウェアを直接操作するようなプログラムを書ける ～～ アセンブリ言語に近い
  - それでいて高級言語 ～～ 色々なCPUで動く
- 現在でも多くのソフトウェアの開発に利用
- 安全性の配慮は少ない
  - 脆弱性のあるプログラムを作ってしまいやすい
  - 例: Morrisのコンピュータ・ワーム (1988)——用意した配列よりも大きなサイズのデータを与えられると、他のデータを破壊してしまう誤りを悪用



代表的なプログラミング言語: C++

- 1980年代に開発
- Cにオブジェクト指向機能をとり入れた言語
  - オブジェクト指向以外の部分はCと同じ
- パーソナルコンピュータのOSやアプリケーションソフトウェア開発の主流言語
  - 例: Windows XP

代表的なプログラミング言語: Java

- 1990年代中頃に開発
  - Webブラウザがダウンロードできるプログラムのための言語として普及
  - 互換性・安全性への配慮
- 現在でも安全性や互換性が求められる場面で多く用いられている
  - Webサーバ上のアプリケーションプログラム
  - 携帯電話にダウンロードするアプリケーションプログラム (e.g., iアプリ)
- 仮想機械による実行方式
- 見た目はCやC++に似ている

Cプログラムの紹介

・ モンテカルロ法による円周率の計算

Rubyプログラム

```
n=1000000
m=0
n.times{
  x=rand  #0以上1未満の
  y=rand  #一様疑似乱数を返す
  if (x*x+y*y)<1.0
    m+=1
  end
}
puts 4*Float(m)/n
```

Cプログラム

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int n = 1000000;
  int m = 0;
  int i;
  double x,y;
  srand48(time(NULL));
  for (i = 0; i < n; i=i+1) {
    x = drand48();
    y = drand48();
    if ((x*x + y*y) < 1.0) {
      m += 1;
    }
  }
  printf("%f¥n", 4*((double)m)/n);
}
```

Cプログラムの特徴

- 複雑な処理を行うライブラリがある（ライブラリを使う準備が必要）
  - 乱数の生成・メッセージ表示など
- 変数は宣言してから使う
- 変数には型が付く

「int i;」=「iという名前の<u>整数</u>をしまう変数を用意せよ」

- 値をしまうメモリの大きさを決めておくため
- 計算の際に適切な機械語命令を選ぶため

←機械語では実数と整数の足し算は違う命令

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
 int n = 1000000;
 int m = 0;
 int i;
 double x, y;
 srand48(time(NULL));
 for (i = 0; i < n; i=i+1) {
   x = drand48();
   y = drand48();
   if ((x*x + y*y) < 1.0) {
     m += 1;
   }
 }
 printf("%f¥n", 4*((double)m)/n);
}
```

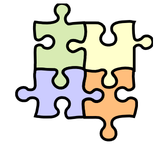
乱数系列の初期化

実数の表示

Cプログラムを実行してみよう

1. プログラムを作る
   - Emacs等を使いmc.cというファイル名で右を入力する
2. コンパイルする
   - 「ターミナル」で **gcc mc.c** と入力
   - 間違いがあるとエラーメッセージが行番号とともに表示される
   - エラーが出なくなると機械語のプログラム(ファイル名:a.out)が作られる
3. 実行する
   - 「ターミナル」で **./a.out** と入力する

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int n = 1000000;
  int m = 0;
  int i;
  double x, y;
  srand48(time(NULL));
  for (i = 0; i < n; i=i+1) {
    x = drand48();
    y = drand48();
    if ((x*x + y*y) < 1.0) {
      m += 1;
    }
  }
  printf("%f¥n", 4*((double)m)/n);
}
```



Cプログラムの紹介(その2)

- オートマトン

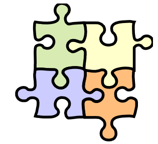
Rubyプログラム

```
$automaton = [[0,1], [2,0], [1,2]]
def make_transitions(input)
  s = 0
  for i in 0..(input.size - 1)
    c = input[i] - ?0
    s = $automaton[s][c]
  end
  s
end
```

```
make_transitions("011001")
make_transitions("011011")
make_transitions("011101")
```

Cプログラム

```
int automaton[3][2] = {{0,1},{2,0},{1,2}};

int make_transitions(char* input) {
  int s = 0, i, c;
  for (i = 0; input[i] != 0; i++) {
    c = input[i] - '0';
    s = automaton[s][c];
  }
  return s ;
}

int main(int argc, char* argv[]) {
  printf("%d¥n", make_transitions("011001"));
  printf("%d¥n", make_transitions("011011"));
  printf("%d¥n", make_transitions("011101"));
}
```



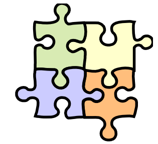
Cプログラムの特徴(その2)

- 配列 = メモリ上に並べたデータ
  - (Rubyの配列よりも)高速
  - 同じ型の値しか入れられない
  - 扱いが面倒: 使う前に大きさを決めて用意する必要がある
  - 危険が多い: はみ出しても読み書きできる → 他のデータを破壊可能
- 関数(Rubyで言うところのメソッド)の引数や返値は型を指定しなければいけない

3×2の整数型の配列を用意

```
int automaton[3][2] = {{0,1},{2,0},{1,2}};

int make_transitions(char* input) {
  int s = 0, i, c;
  for (i = 0; input[i] != 0; i++) {
    c = input[i] - '0';
    s = automaton[s][c];
  }
  return s ;
}

int main(int argc, char* argv[]) {
  printf("%d¥n", make_transitions("011001"));
  printf("%d¥n", make_transitions("011011"));
  printf("%d¥n", make_transitions("011101"));
}
```

文字列を1つ受け取り整数を返す関数



Cプログラムの紹介(その3)

- 数え上げによるフィボナッチ数の計算
- RubyとCの実行結果を比べよ
- Cのint型は32ビット符号付整数(多くの場合)
  →それ以上になると下位32ビットだけが残る
  ※fib中のintをlong longに、main中の%dを%lldに変えると64ビット整数になる
- Rubyの整数は値が大きくなると自動的に多倍長整数に変換される

Rubyプログラム

```
def fib(x)
  i=0; fi=1; fi1=1
  while i < x
    i = i+1
    f2 = fi+fi1
    fi = fi1
    fi1 = f2
  end
  fi
end
```

```
fib(30)
fib(50)
```

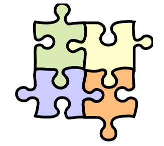
Cプログラム

```
#include <stdio.h>

int fib(int x) {
  int i = 0, fi=1, fi1 = 1, f2;
  while (i < x) {
    i = i + 1;
    f2 = fi+fi1;
    fi = fi1;
    fi1 = f2;
  }
  return fi;
}

int main(int argc, char* argv[]) {
  printf("%d¥n", fib(30));
  printf("%d¥n", fib(50));
}
```

コンパイル実行方式と誤り

- Cプログラムに誤りを混入させてコンパイルし、コンパイラがどのような検査をしてくれるのかを調べてみよう
  1. ここのmをkに変えてコンパイル（間違った変数名）
  2. ここのint m = 0;をint m = “abc”;に変えてコンパイル（数値のところに文字列）
- 同様の誤りをRubyプログラムに混入させると何が起きるか？ プログラムのどこに誤りがあると言われるか？

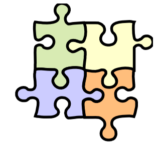
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int n = 1000000;
  int m = 0;
  int i;
  double x,y;
  srand48(time(NULL));
  for (i = 0; i < n; i=i+1) {
    x = drand48();
    y = drand48();
    if ((x*x + y*y) < 1.0) {
      m += 1;
    }
  }
  printf("%f¥n", 4*((double)m)/n);
}
```



アセンブリ言語プログラムを見てみよう

1. 「ターミナル」で<u>`gcc -O -S mc.c`</u> と入力する
   →mc.cがコンパイルされ、アセンブリ言語プログラムmc.sが作られる
2. Emacs等のエディタでmc.sを見る

(おまけ: mc.cの数式を書き換えてmc.sを作り直し、変化した場所を見てみると対応関係が分かる)



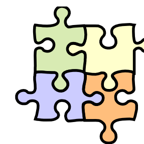
アセンブリ言語プログラム

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int n = 1000000;
  int m = 0;
  int i;
  double x,y;
  srand48(time(NULL));
  for (i = 0; i < n; i=i+1) {
    x = drand48();
    y = drand48();
    if ((x*x + y*y) < 1.0) {
      m += 1;
    }
  }
```

mc.sの一部

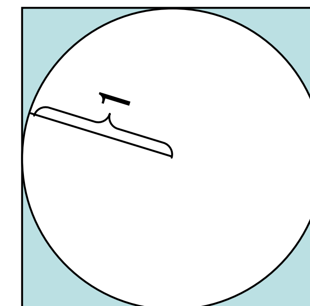
```
    :
    (f30には1.0が、r29には1000000が入っている)
L2:
    bl L_drand48$stub       drand48()を実行
    fmr f31,f1              結果をf31に保存
    bl L_drand48$stub       drand48()を実行
    fmul f1,f1,f1           結果(f1)を自乗
    fmadd f31,f31,f31,f1    f31=f31*f31+f1
    fcmpu cr7,f31,f30       f31とf30を比較
    bnl+ cr7,L3             f31が小さくなければL3へ
    addi r29,r29,1          r29=r29+1
L3:
    addic. r30,r30,-1       r30=r30+(-1)
    bne+ cr0,L2             r30がゼロでなければL2へ
    :
```

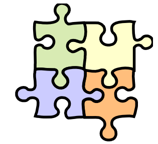
機械語にするとどのくらいの数の命令になるか?



おまけ: アセンブリ言語のプログラムを書き換えてみよう

1. Emacs等のエディタを使ってmc.sの途中にある「bnl+ cr7,L3」を「blt+ cr7,L3」に書き換える
   - 意味: 「比較結果が『小さくない』場合にL3へ飛ぶ」を『小さい』に変える
     →モンテカルロ法の不等号を逆にしたことに相当
2. 「ターミナル」で`gcc mc.s`と入力する
   →mc.sを機械語にしたa.outが作られる
3. 「ターミナル」で`./a.out`と入力して実行→結果の値は?





CPUによって機械語が違うことの確認

- CPUが違うコンピュータ上でコンパイルすると、作られる機械語(もアセンブリ言語)プログラムも違うものになる

Power (ECCのiMacのCPU)

```
L2:
    bl L_drand48$stub
    fmr f31,f1
    bl L_drand48$stub
    fmul f1,f1,f1
    fmadd f31,f31,f31,f1
    fcmpu cr7,f31,f30
    bnl+ cr7,L3
    addi r29,r29,1
L3:
    addic. r30,r30,-1
    bne+ cr0,L2
```

Intel x86 (いわゆるWindows機のCPU)

```
.L36:
    call drand48
    fstpt -16(%ebp)
    call drand48
    fldt -16(%ebp)
    fmul %st(0),%st
    fxch %st(1)
    fmul %st(0),%st
    faddp %st,%st(1)
```

```
    fld1
    fcompp
    fnstsw %ax
    andb $69,%ah
    jne .L35
    incl %esi
.L35:
    decl %ebx
    jns .L36
```

実行方式と速度

インタプリタとコンパイラは実行速度がどのくらい違うか、モンテカルロ法による円周率の計算に要する時間を比べてみよ

- 時間の測り方(以下を「ターミナル」に入力)
  - Cの場合: **time ./a.out**
  - Rubyの場合: **time ruby mc.rb**

    (mc.rbにプログラムを入力しておくこと)