「今度こそわかるＰ≠ＮＰ」（渡辺治）の補足説明
講談社サイエンティフィク, 2014
(2014.3.9)

原始計算機・原始プログラムの補足（タイプ１）

この補足は，プログラミングの経験が無い，もしくは浅い人たちのためのもの[1]．計算とは何かを原始計算機と原始プログラムを用いて簡単に説明する．

コンピュータサイエンスにおいて「計算」とは情報処理のこと．日常用語で使っている「計算」より非常に広い概念である．と言っても基本は，(i) 数に対する四則演算．(ii) 大きさの比較．そして (iii)（指定した条件が成り立つまでの）繰り返し，の 3 種類である．すべての「計算」（＝情報処理）は，これを組合わせることで実現できてしまう，という大発見（つまり，コンピュータの発明）が今日の情報社会を生み出したのだ．

この「計算」を表わす方法にはいろいろある．原始計算機と原始計算機上で動く原始プログラムもその１つ．以下では，原始計算機と原始プログラムを用いて，どのように「計算」を表わすのかを具体的に見て行こう．

まず，原始計算機について．これは 4 個のレジスタとデータ記憶用のメモリからなる仮想的な計算機である（本書の図 2.1）．上で「計算」は (i) ～ (iii) で表わすことができる，と断言したが，もう少し補足が必要である．それは「変数」とか「レジスタ」とか「メモリ」と呼ばれるもので，入力されるデータ（つまり問題例）や計算の途中結果などを記憶しておく「入れ物」である．

通常のプログラミング言語では,「変数」（数学で使う変数とは少し違う）がその入れ物として使われる．原始計算機では 4 つの レジスタ と メモリ がその入れ物だ．レジスタには自然数（0 以上の整数）が格納できる．一方，メモリは 1 つには 0 か 1 の 1 ビットの情報しか入れられない.（なぜ，こういう設定にしたかは，次の補足説明を参照.）レジスタには番号が付いていて，レジスタ #1 とか，レジスタ #3 などと番号で示すことができる．一方，メモリには，0 から順に番号が付いている．これを 番地 という．たとえば「5 番地のメモリ 」というと，番号 5 が付いた「入れ物」（0 か 1 しか入れられない）を指す.

原始計算機では，これら 4 つのレジスタと（多量の）メモリに数を格納し，それらに対して上記の (i) ～ (iii) の操作を行うのである．その操作の手順を書いた「手順書」のが原始プログラムである．つまり，原始計算機はすべて同じような形をしている．ただ，それを動かすための「手順書」（＝原始プログラム）だけが異なるのである．本書で「原始計算機械 M」などと出てくるが，各々の機械で異なるのは（今回説明を省略する定数レジスタの値と）原始プログラムだけだ.

今回は次の原始プログラムを使って説明しよう．この原始プログラムが内臓された原始計算機 M を考える．プログラムは 1 行に 1 命令が書かれている．命令の前にある “1:” などの数字は 行番号 で，命令には直接関係はない.

```
1: get#1    [1]
2: mult2#1
3: add#1    [2]
4: sub#1     3
5: halt      #1
```

[1]ただし，あくまで補足程度なので，計算の基礎に関する本格的な勉強をするには，プログラミングの本やコンピュータサイエンスの入門書を見て欲しい.

原始計算機は，メモリに入力列を与えて（細かく言えば定数レジスタの値もセットして）ボタンか何かを押すと，内臓されたプログラムを最初の行から実行し，（もし正常終了したならば）何らかの計算結果を出すのである．

この機械 M の場合，行う計算は非常に単純だ．与えられた入力列の最初の 2 ビットだけを見て，それを 2 進数だと解釈し，その値から 1 だけ引いた値を求める計算[2]である．たとえば，入力列の最初の 2 ビットが 11 だったとしよう（つまり，3 の 2 進数表現である）．この各ビットがメモリの 1 番地目と 2 番地目に格納された状態で M の実行が始まるのである．もしかすると，3 番地以降にも 0 や 1 が入っているかもしれないが，それらは計算には影響ない．

プログラムの実行は上の行から順に進む．まず `get#1 [1]` という命令が実行される．これはメモリの 1 番地目の値（0 または 1）をレジスタ #1 に取って来いという命令だ[3]．この `[1]` が「メモリの 1 番目の値」を意味している．以下，次のように実行が進む．

	結果（今の例では）
(1) メモリ [1] の値を #1 へ持って来る	#1 の値は 1 となる
(2) #1 の値を 2 倍する	#1 の値は 2 となる
(3) #1 の値にメモリ [2] の値を加える	#1 の値は 3 となる
(4) #1 の値から 3 引く	#1 の値は 0 となる
(5) #1 の値を答えとして停止	#1 の値，つまり 2 が計算結果

このように実行して M が答えを出すことを，本書では「M が入力列 11 に対して 2 を出力した」と言い，$M(11) = 2$ と書くのである．（ちなみに，$M(110) = 2$, $M(111) = 2$, $M(1100) = 2$, ... である．M は最初の 2 ビットが同じならば同じ値を出力するので．）

各々の命令の意味は，この実行例からほぼ明らかだろう．少し補足しよう．まず，`add#1` や `sub#1` だが，これはレジスタ #1 用の命令．レジスタ #2 ～ #4 にも同様の命令が用意されているものとする．それに対し `sub#1 3` の “3” などは命令の引数（目的語？）である．この場合は「数 3」，つまり「3 を引け」という意味だ．一方，引数に `[2]` などと書くと，これはメモリ 2 番地の値を意味する．同様に，引数の `#2` は，レジスタ #2 の値，という意味である．なお，計算は自然数上に限るので，`sub#1` のような命令で，もしも引く数の方が大きい場合には値は 0 になるものとする．したがって，$M(10) = 0$ である．

ところで，`mult2#1` は「レジスタ #1 の値を 2 倍せよ」という命令．足し算，引き算の命令は引数を取るが，掛け算は好きな数をかけることはできない（割り算も同様）．その理由については本書（p.29）にも書いたが，補足説明 `ot2-compmodel_type2.pdf` でも説明する．

次に，もう少し複雑な計算を考えてみよう．今度は次ページのようなプログラムである（本書の図 2.2）．これは，入力列のうち，最初の 2 ビットが表わす数 x と，次の 2 ビットが表わす数 y の積を求めるプログラムである．このプログラムを内蔵した原始計算機械を M_2 とすると，たとえば，$M_2(0110) = 2$, $M_2(1011) = 6$ などとなる．（この例では 2 進列 01 は数 1 と見なしている．）

---

[2]「それってどんな意味があるの？」とは聞かないで欲しい．あくまで例として単純なものを考えたに過ぎないので．

[3]こうした命令には，どのようなものがあって，各々どんな意味を持つか，は本書では詳しく定めていない．「まあ，妥当なものを想定して下さい」と読者に丸投げしてしまったのだ．いくつか注意点を除いては本書の議論にはあまり影響がないからである．ただ「それでは気持ちが悪い」という人のために，本ウェブの「こだわり補足」のコーナーで基本形を 1 つ紹介する（予定である）．

```
@main
1: get#1   [1]
2: mult2#1
3: add#1   [2]
4: get#2   [3]
5: mult2#2
6: add#2   [4]
7: get#3    0
8: @mult
9: halt     #3

@mult   #3 に #1 x #2 を計算するサブプログラム
1: 0eq#2
2: return
3: add#3    #1
4: add#2    1
5: goto     1
```

「プログラムが 2 つある？」と思われるかもしれない．長くなるので，掛け算を求める部分計算を `@mult` という サブプログラム に分離して書いたからだ．これは，あくまで見易さの方便のため．実際には `@mult` と書かれている 8 行目に，`@mult` のプログラムを埋め込んだ，次のようなプログラムと思ってもらってもよい．以下では，こちらの方を使って説明しよう．

```
@main
1: get#1   [1]
2: mult2#1
3: add#1   [2]
4: get#2   [3]
5: mult2#2
6: add#2   [4]
7: get#3    0
8: 0eq#2
9: goto     13
10: add#3    #1
11: add#2    1
12: goto     8
13: halt     #3
```

このプログラムを持つ M_2 に，たとえば入力列 1101 を与えて実行すると，7 行目が終わった時点で，

```
#1 = 3,  #2 = 2,  #3 = 0
```

となっている．それ以後，8 行目からの実行の結果を書くと次のようになる．

	結果（今の例では）
(1) #2 の値が 0 か？を調べる	#2 の値は 0 でないので 10 行目へ進む
(2) #3 の値に #1 の値を加える	#3 = 3 となる
(3) #2 の値から 1 引く	#2 = 1 となる
(4) 8 行目へ行く	8 行目に進む
(5) #2 の値が 0 か？を調べる	#2 の値は 0 でないので 10 行目へ進む
(6) #3 の値に #1 の値を加える	#3 = 6 となる
(7) #2 の値から 1 引く	#2 = 0 となる
(8) 8 行目へ行く	8 行目に進む
(9) #2 の値が 0 か？を調べる	#2 の値が 0 なので 9 行目へ進む
(10) 13 行目へ行く	13 行目に進む
(11) #3 の値を答えとして停止	#3 の値，つまり 6 が計算結果

まず，8 行目の命令 0eq#2 について．これは「レジスタ #2 の値が 0 と等しいか？」という命令だ．この種の命令では，もしも条件が成り立つならば次の行に進み，そうでない場合には次の次の行に進む，という定めになっていることが多い．本書でもそういう解釈にする．現在の #2 の値は 2 なので，次の行は飛ばして 10 行目に進み，計算を続ける．次に重要なのは goto 8 という 12 行目の命令．これは「8 行目に進め」という命令だが，これにより，同じ計算の繰り返しを実現できる．実際，実行課程の (5), (6), (7), (8) は，(1), (2), (3), (4) の繰り返しである．これが最初に述べた，(iii)（指定した条件が成り立つまでの）繰り返し，なのである．

以上，少々，面倒だったかもしれないが，ほぼこれですべてである．これだけで，すべての「計算」が書けてしまうのだ！(まあ，その点を説明し出すときりがないので，信じてもらうことで勘弁願いたい.)

おっと，もう 1 つ，重要な点を忘れた．たとえば，

```
get#1   [#2]
```

という命令の書き方もある．これはレジスタ #1 に値を持って来て格納する命令だが，持って来るデータが，レジスタ #2 の値で指定しているメモリの番地の値である点が重要．番地を直接書くのではなく，レジスタの値を使って指定することも可能なのである．これは少し古いいい方かもしれないが「間接番地指定」と呼ばれている．あるいは，C 言語のポインタ変数の使い方に相当する．このような使い方ができると大変便利なのである．

補足説明 ot1-compmodel_type1 終了