[1]サンプルプログラム

マンデルブロ集合　**75 バイト**（それまでの最高記録は 10 年くらい前の 109 バイト）
回転キューブ　**340 バイト**（それまでの最高記録は 20 年くらい前の 1393 バイト）
3D-wave　**92 バイト**（坂井さんが NLL 用に書いていたものを移植：サイズ比較対象なし）

[2]可変長バイトコードの切り分けコスト問題

0～7 = 4 ビット = 0xxx 型（データ割合 3/4=75%）
8～b = 8 ビット = 10xx xxxx 型（データ割合 6/8=75%）
c～d = 12 ビット = 110x xxxx xxxx 型（データ割合 9/12=75%）
e = 16 ビット = 1110 xxxx xxxx xxxx 型（データ割合 12/16=75%）
つまりプログラムのうち 25%には意味のある値は載っていない！
単にデータの区切り位置を認識するためだけに使われている。
もしここを改良できる方法があれば、さらに小さくできる！！

[3]ウィンドウサイズ指定: [x] [y]

0-0: 256x256, **0-1**: 320x320, **0-2**: 400x400, **0-3**: 512x512, **0-4**: 640x640, **0-5**: 800x800
1-0: 256x192, **1-1**: 320x240, **1-2**: 400x300, **1-3**: 512x384, **1-4**: 640x480, **1-5**: 800x600
2-0: 256x160, **2-1**: 320x200, **2-2**: 400x250, **2-3**: 512x320, **2-4**: 640x400, **2-5**: 800x500
3-0: 256x144, **3-1**: 320x180, **3-2**: 400x225, **3-3**: 512x288, **3-4**: 640x360, **3-5**: 800x450

[4]命令コード表

0 [式]:　変数定義＆代入（＝変数番号を省略できる 1）
1 [式] [変数番号]:　代入
2 [式] [変数番号]:　+=（加算代入）
3 [API 番号] [引数式]:　API 呼び出し
4 [終値]:　変数定義＆for 文 {　（＝変数番号を省略できる 5）
5 [終値] [変数番号]:　for 文 {
6:　}
88 [式 a] [式 b]:　if (a == b) {
89 [式 a] [式 b]:　if (a != b) {
8a [式 a] [式 b]:　if (a < b) {
8b [式 a] [式 b]:　if (a >= b) {
8c [式 a] [式 b]:　if (a <= b) {
8d [式 a] [式 b]:　if (a > b) {
b7 [数 a] [数 b]:　式内で a と b の意味を入れ替える
7 [式] [配列変数番号] [添え字式]:　配列変数へ代入

[5]式の要素のコード表

0: 基本的には定数 0 を意味するが、文脈で変わる（例：加算や減算では 1 に解釈される、代入では 0）
1: 最も直前に代入された変数（repeat0）
2: 2 番目に新しい変数（repeat1）
3: 3 番目に新しい変数（repeat2）
4: + （二項演算子）
5: - （二項演算子）
6: * （二項演算子）
80～98: 定数 0～24
99～9e: 定数 32, 64, 128, 256, 512, 1024
9f: 後続の値を 65536 倍

b0～b6: / % << >> | ^ &
7 [配列変数番号] [添え字式]: 配列参照
b9 [添え字式]: 配列参照 repeat0
ba [添え字式]: 配列参照 repeat1
bb [添え字式]: 配列参照 repeat2
bc～bf: 定数 -4 ～ -1
a0～af: 整数変数 0～15 番

[6]自作バイトコードによるコードゴルフの「ズル問題」
→特定のアプリにだけ有利な仕様を選択してないか？

[3D-wave のバイトコード] （オフセットは 10 進数で書いています、4bit 単位）

```
000: 0-1-4; // [シングルウィンドウモード]  ウィンドウの大きさは 640x480.
003: 7; 2-5-6-1-0-1; 0; // int gx[4096], gy[4096];
011: dc3; // while (!isClose(win)) {
014: 2-0-0; // t++;
017: 3-bd-88; // wait(8); & ウィンドウ内を#000000 で塗りつぶす（CLS）.
022: 4-c2a; // for (y1 = 0; y1 < 42; y1++) {
026: 4-c2a; // for (x1 = 0; x1 < 42; x1++) {
030: 0-5-2-94; bc-1; // y = y1 - 20; x = x1 - 20;
038: 0-db1-9f-4-6-1-1-6-2-2; // d = ff16Sqrt((x * x + y * y) << 16);
051: 0-b0-6-db2-5-b3-6-1-e28c-8c-6-a5-e413-c64-4-1-9f-85;
         // z = ff16Sin(((d * 652) >> 12) - 1043 * t) * 100 / (d + 327680);
085: 0-4-6-a5-9a-a4; // d = y1 * 64 + x1;
094: 7-6-4-5-a3-a4-c28-88-0-1; // gx[d] = (x - y) * 8 + 320;
109: 7-4-6-4-4-a3-a4-c3c-84-2-1-1; // gy[d] = (x + y) * 4 + z + 240;
126: 8b-b4-5-a5-0-5-a6-0-0; // if (x1 >= 1 && y1 >= 1) {
139: 0-5-1-0; // z = d - 1;
143: 3-89-7-0-5-1-9a-7-1-5-1-9a-7-0-5-2-9a-7-1-5-2-9a-df3;
         // drawLine(w, gx[z - 64], gy[z - 64], gx[d - 64], gy[d - 64], 0xff00ff);
173: bc-8b-1-7-1-1-df3;
         // drawLine(w, gx[z - 64], gy[z - 64], gx[z], gy[z], 0xff00ff);
184:
```

```
    +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F  0123456789ABCDEF
 0 01 47 25 61 01 0D C3 20-03 BD 88 4C 2A 4C 2A 05  .G%a... ...L*L*.
10 29 4B C1 0D B1 9F 46 11-62 20 B0 6D B2 5B 36 1E  )K....F.b .m.[6.
20 28 C8 C6 A5 E4 13 C6 44-19 F8 50 46 A5 9A A4 76  (......D..PF...v
30 45 A3 A4 C2 88 80 17 46-44 A3 A4 C3 C8 42 11 8B  E......FD....B..
40 B4 5A 50 5A 60 00 51 03-89 70 51 9A 71 51 9A 70  .ZPZ`.Q..pQ.qQ.p
50 52 9A 71 52 9A DF 3B C8-B1 71 1D F3 |           R.qR..;..q..
```

[既存の CPU アーキテクチャで同じことをしたらどのくらいになるか？]

92 バイト	uck [ultra - compact - kharc]
276 バイト	arc-elf (HS)
286 バイト	arm-eabi (thumb2, armv7-a)
301 バイト	rx-elf
307 バイト	WCOFF x86 (注 1)
320 バイト	bfin-elf
344 バイト	arm-eabi (thumb, arm7tdmi, armv4t)
346 バイト	mn10300-elf
348 バイト	m68k-elf
360 バイト	cr16-elf
378 バイト	msp430-elf
383 バイト	i386-elf (注 1)
392 バイト	xstormy16-elf
401 バイト	x86_64-elf (x86-64)
404 バイト	arm-eabi (arm7tdmi, armv4t)
408 バイト	mips-elf (mips16)
411 バイト	vax-linux
412 バイト	rl78-elf
436 バイト	aarch64-elf (armv8-a+crc)
436 バイト	h8300-elf
444 バイト	m32r-elf
448 バイト	fr30-elf

450 バイト	riscv-elf (rv32imac)
452 バイト	nios2-elf
456 バイト	s390-linux (z900)
464 バイト	powerpc64-linux
472 バイト	mcore-elf
480 バイト	frv-elf
484 バイト	pe-i386
506 バイト	moxie-elf
512 バイト	or1k-elf
512 バイト	xtensa-elf
516 バイト	mips-elf (mips1)
524 バイト	h8300s-elf
544 バイト	mips64-elf (mips64)
556 バイト	pru-elf
568 バイト	riscv32-elf (rv32imafd)
604 バイト	hppa-linux
660 バイト	sparc64-elf (v9)
692 バイト	alpha-linux (ev4)
692 バイト	riscv64-elf (rv64imafd)
768 バイト	epiphany-elf
1232 バイト	ia64-elf

gcc でオブジェクトファイルを作り、text のサイズのみを上記にあげた。だからヘッダのサイズは影響してない。uck ではヘッダもすべて含んでいる。

謝辞：この表を作るにあたっては、kozos の坂井さんにたくさん助けていただきました。

[なぜマンデルブロ集合のアプリは 75 バイトなのか？]
→アプリケーションモード 1 を使っているから。

mandel0.uck は、「1-1-3-7; 1-0-3; 4-84; ・・・」という内容になっている
（3D-wave では最初のモード値は 0 で以下のような小細工はない）。
モード 1 は、2 つの for 文を自動生成する。

```
Int main()
{
    i63 = openWin(x 解像度, y 解像度);
    for (i21 = 0; i21 < y 解像度; i21++) {
        for (i20 = 0; i20 < x 解像度; i20++) {
```

ここまでを自動生成する。i20, i21 は、7 を指定すると自動で選ばれる。7 以外を書けばループカウンタ変数を選べる。

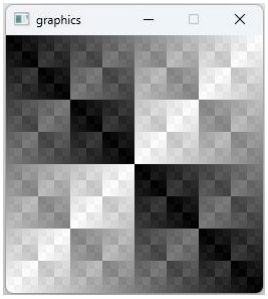
[Q]なぜこんなモードがあるの？
[A]uck は静止画を描画することがとても得意だなと思った。そして静止画を描くときは上記のような出だしになることがとても多い。じゃあそういうモードを作って「得意をさらに得意にしよう」と思った。
このモードの時は API 番号 0 が特別に用意されて、これはカラーコードだけを指定すればよい setPix で、この時ウィンドウハンドル・x 座標・y 座標はすべて自動で補われる。

[Q]なぜそこまでがんばるのか？これはもはや「ズル」ではないのか？
[A]確かにその恐れはある。もしそうであればこのモードは使わないということにするしかない。

少し考えてみてほしい。もし私がこのモード 1 を作らずに、世界の誰かが uck の類似物を作り、それにはモード 1 みたいなものを付けていたらどうだろうか。そしてその作者は言うのだ「私こそ世界一だ！」と。・・・私がその宣言の直後に「いや自分だってモード 1 を作ろうとは思ったけど、ズルかもしれないから実装しなかったのだ！」と反論したとしても、それが何になるだろう。負け犬の遠吠えではないか。
だからやるべきなのだ。少なくとも私はそう考えた。・・・世界一を守るとはそういうことなのだと思う。

[そして究極のモード 2]
モード 2 では、解像度が 256x256 に固定される。その上、ループする変数が i1 と i2 に固定される。つまりこれらを一切指定しなくてよい。API 呼び出しすら自動で行われて、いきなりカラーコードを計算するための式を書けばよい。
なぜ 256x256 なのか。それはサンプルプログラムを作っているときに、この解像度を指定することが一番多かったからだ。256x256 は、ちょっとした静止画を作るにはとても使い勝手がいい。
ループ変数の指定も、この規模になってくると自由に指定できなくていいので、その分さらにコンパクトになってくれるほうがありがたい。

こうして究極の 2 バイトプログラムや 3 バイトプログラムが生まれた！（「23 20」「20 b5 12」）



[Q] どうせマンデル描画だけが小さくなっただけでしょ？
[A] 2014 年に OSECPU-VM rev.2 を発表したとき、誰が何を作っても OSECPU-VM アプリは驚異的に小さくなるという、夢のような性質を持っていました。私は特定のアプリケーションだけが小さくなるような「ズル」をやらないという信念があり、どんなアプリでも等しく効果があるような仕様しか採用しません。・・・ということで、3D-wave や kcube（キューブ回転）も小さくなっています。

[Q] OSECPU-VM の時にすでに究極だったのに、なぜさらに小さくなったのですか？
[A] マンデル描画は 109→75 と 31%も小さくなっています。OSECPU-VM 版では「もう小さくできる余地はない」と思っていましたし、そのように主張しました。・・・uck になってさらに小さくできたのには、大ざっぱにいうと **3 つの新技術**があります。

1 つめは、式の記述方法を単純な 2〜3 項に固定していた方法をやめて、スタックマシン的な方法に改めたことです。OSECPU-VM のときは x86 や ARM などの CPU の機械語を模倣してバイトコードを設計していたので、命令のオペランドは 2〜3 個程度でした。簡単な式だけならこれはとても有利でしたが、式が複雑になってくるとテンポラリ変数を明示する分だけ不利になって、そこが弱点でした。一般的なスタックマシンでは、PUSH 命令を多用することになりそれがコードの肥大化を招くのですが、この uck では PUSH 命令の命令コードの命令長は実質的に 1 ビットでしかなく、これによりこの弱点を克服しています。・・・ちなみにこの改良は buntan-pc プロジェクトの開発をしている時に思い付きました。

2 つめは、変数番号の指定方法を工夫したことです。普通に「変数番号 12」のような指定はできますが、「3 命令前に代入した変数」のように変数を指定することもできるのです。プログラムでは 1〜8 命令前に代入した変数にアクセスする頻度が非常に高く、これによりほとんどのケースで変数番号は 2〜3 ビットで表すことができています。また「n 命令前に代入した変数」の方法で変数を参照するなら、そもそも変数番号は重複さえしなければ何でもよいということになり、だから uck が適当に割り当てても問題なくなります。こうして初出時は変数番号を省略することができるようになります。

3 つめは hh4 という可変長の機械語のエンコード方法を改良したことです。今までは 0〜e だけを使い、f は使わない（パディング用）としていました。これは柔軟性において重宝しましたが、コンパクトなプログラムを作るときにはこの柔軟性が必要になることはなく、結局究極の小ささの達成を妨げていました。今回これを見直して、0〜f のすべてを使うようにして、さらに数パーセントの性能向上に成功したのです。

なお、バイトコードにハフマン圧縮などを使えばさらに小さくなるのは確実だと思います。しかしそれはまだやっていません。ハフマンなどを入れてしまうと、もう人間がバイトコードを見ても、容易には読めなくなってしまいます。現状では容易に読めるようになっています（4 ビット単位なので少し訓練は必要ですが）。

ゆりりんが使っているC言語について

ゆりりんは、小学1年生の時からプログラミングをはじめました。小学1年生は、英語（アルファベット）が読めません。1年生は、ひらがなやカタカナを勉強する学年です。そんなときに英語を覚えるのは大変すぎて、プログラミングが嫌いになってしまいます。

C言語は英語の小文字をたくさん使いますが、キーボードに書かれているのは大文字です。これがプログラミングを低学年でやってもらうときの難しさになりました。それで、ゆりりんの父はC言語を自分で作って、そのC言語では命令を大文字で書いてもいいということにしました。だからゆりりんは大文字でC言語を書きます。中学生になったら、小文字で書く練習をする予定です。

父が作ったC言語は「イージーシー（easy-C)」というのですが、イージーシーでは最初のinclude文やmain()などをすべて省略して書き始めることができます。これは「毎回同じことを書くなんてめんどうだよ。それくらいどうにかできないの？！」という子供の意見に父が負けたからです。だからゆりりんはいつもmainの中身だけをいきなり書くのです。ゲームを作るのに便利な関数も、includeなしでいきなり使えるようになっています。