デュアルスタック電卓のプログラミング技法*

和田英一†
IIJ イノベーションインスティテュート‡

何回か前の冬のプロシンでデモを行った iPod touch, iPhone 用のスタック電卓を, デュアルスタックにし, プログラムの機能を追加することを検討している.
デュアルスタックは, 下と上の 2 つのスタックで構成し, 通常のスタック演算は下のスタックの最上部で実行する. その他, 下のスタックの最上段をポップアップし, 上のスタックにプッシュダウンする命令と, その逆命令を用意する.
スタックをこのように接続すると, Turing マシンの無限テープと同様な作業領域になり, 手間を考えなければ, 相当な計算が出来るようになる. 配列やリストには対応していない.
プログラムとしては, 電卓の押しボタンのそれぞれに, ASCII 文字を対応させ, その文字列で, 電卓の機能を実行出来るようにした. そういう文字列に, 1 文字の名前をつけて記憶すると, その文字で対応する文字列のプログラムが実行出来る. ループや判定を可能にすべく, 条件付き命令も用意した. 文字列に中に, 自分の名前の文字を書くと, 再帰呼出しが出来る.
例えば階乗のプログラムを '("_6_G1+5_"G"1-!*)!=' と定義すると, '10!' で 3628800 が得られる. こういうプログラムの考え方, 書き方を説明する.
現在はシミュレータでしか動作していないが, 電卓に組み込んだ時のインターフェースも設計中で, それについても述べる予定である.

iPod 用電卓 Happy Hacking Calculator

Happy Hacking Calculator (HHCalc) は, 逆ポーランド記法 (Reverse Polish Notation (RPN)) を採用した整数計算用の電卓で, iPod touch と iPhone 用に実装した. 式の処理に必要なスタックの要素は 2 の補数の 64 ビット. スタックの深さは 16 段で, 16 回を超えてプッシュすると, スタックの底は壊れる. 一方, スタックの底を超えてポップすると, 下から 0 が無限に現れる.

HHCalc には '入力中 (Input)' と '演算後 (Ready)' の 2 つの状態をとる基本状態 (basic state) と, '十六進 (Hex)', '十進 (Dec)', '八進 (Oct)' の 3 つの状態をとる基数状態 (radix state) とがある.

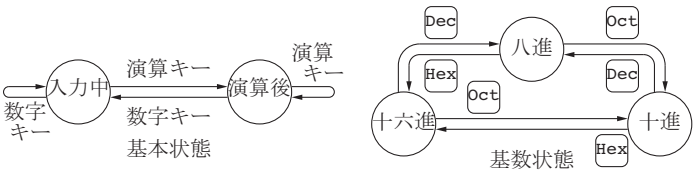

図 **0** HHCalc の状態遷移

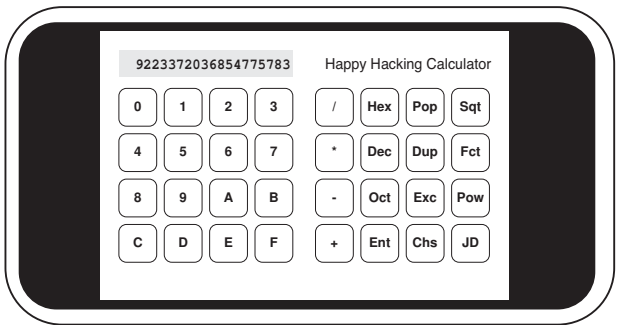

図 **1** HHCalc のインターフェース

HHCalc のインターフェースは, 左半分に 0, 1, ···, F の数字キーと, 右半分に /, *, ···, JD の演算キーがそれぞれ 16 個あり, 数字キーの上に演算結果を示す窓がある.

基数状態は, 外部の基数進法と内部の二進法との変換に使われる. 状態切替えの演算キーは Hex, Dec, Oct で, それぞれ基数 (radix) を 16, 10, 8 に設定する. 基数状態は *ir* に記憶する.

基数の現在の状態は, 基数変更のいずれかのキーの文字の緑で示す. 起動時の基数状態は '十進' に設定する.

*The Art of Happy Hacking Calculator Programming
†Eiiti Wada
‡IIJ Innovation Institute

‘演算後’ 状態で, どれかの数字キーが押されると, 基本状態は ‘入力中’ になり, スタックを 1 段押し下げ, 次の演算キーが押されるまでの数字キーの列が新しいスタックトップに入る. 基数状態に関係なく, 数字キー 0 から 9 は整数 9 から 9 を入力し, A から F は 10 から 15 を入力する.
演算キーは, 基本状態がいずれにあっても, その時点のスタックトップの被演算子について演算を行い, 演算結果をスタックトップに置き, 窓に表示する. 基本状態を ‘演算後’ に設定する.
以下では, $s[ix]$ はスタックトップの要素を示す; $s[ix-1]$ はトップのすぐ下の要素を示す. ix はスタックトップの添字, ir は基数のレジスタである. (ix は 16 を法として増減される.)
注意: HHCalc にはバックスペースキーがないので, 数値入力を間違えたとき, [Pop]キーを押し, スタックトップの数値を捨てる.
注意: 殆んどの演算は瞬時に終るが, 2^{45} を超える素数の素因数分解にはかなりの時間がかかる. 素数 $2^{47}-115$ で 7 秒程度かかる. 演算実行中は[Fct] が青になっているので分かる.

演算

[+] 加算. $s[ix-1] \leftarrow s[ix-1]+s[ix]$, $s[ix] \leftarrow 0$, $ix \leftarrow ix-1$.
[−] 減算. $s[ix-1] \leftarrow s[ix-1]-s[ix]$, $s[ix] \leftarrow 0$, $ix \leftarrow ix-1$.
[*] 乗算. $s[ix-1] \leftarrow s[ix-1] \times s[ix]$, $s[ix] \leftarrow 0$, $ix \leftarrow ix-1$.
[/] 除算. $n \leftarrow s[ix-1]$, $d \leftarrow s[ix]$. $d \neq 0$ なら, $r \leftarrow n\%d$, $q \leftarrow (n-r)/d$, ($0 < d$ の時, $0 \leq r < d$; $d < 0$ の時, $d \leq r < 0$.) $s[ix] \leftarrow q$, $s[ix-1] \leftarrow r$. $d = 0$ なら, エラー *.
[Hex] 十六進. $ir \leftarrow 16$.
[Dec] 十進. $ir \leftarrow 10$.
[Oct] 八進 l. $ir \leftarrow 8$.
[Ent] 何もしない.
[Pop] 削除. $s[ix] \leftarrow 0$, $ix \leftarrow ix-1$.
[Dup] 複製. $s[ix+1] \leftarrow s[ix]$, $ix \leftarrow ix+1$.
[Exc] 交換. $t \leftarrow s[ix]$, $s[ix] \leftarrow s[ix-1]$, $s[ix-1] \leftarrow t$.
[Chs] 符号反転. $s[ix] \leftarrow -s[ix]$.
[Sqt] 開平. $n \leftarrow s[ix]$. $n \geq 0$ なら, $q \leftarrow \lfloor\sqrt{n}\rfloor$, $r \leftarrow n-q \times q$, $s[ix+1] \leftarrow q$, $s[ix] \leftarrow r$, $ix \leftarrow ix+1$. $n < 0$ なら, エラー *.
[Fct] 素因数分解. $n \leftarrow s[ix]$. $n \geq 2$ なら, $n = r \times p^k$, (ただし $p \geq 2$ は n の最小素因数; r は p を素因数に持たない.) $s[ix+2] \leftarrow p$, $s[ix+1] \leftarrow k$, $s[ix] \leftarrow r$, $ix \leftarrow ix+2$. $n < 2$ なら, エラー *.
[Pow] 冪乗. $b \leftarrow s[ix-1]$, $p \leftarrow s[ix]$. $p > 0$ または ($p = 0$ かつ $b \neq 0$) なら, $s[ix-1] \leftarrow b^p$, $ix \leftarrow ix-1$. $p < 0$ または ($p = 0$ かつ $b = 0$) なら, エラー *.
[JD] ユリウス日. $s[ix] \leftarrow y$ 年 m 月 d 日世界時正午のユリウス日数. y, m, d は $s[ix]$ に $y \times 10000 + m \times 100 + d$ で表わす.
* エラーの場合, 演算キーの名前が赤で示され, スタックは変らない.

シミュレータ

この個人用電卓の, 機能を拡張したヴァージョンが, シミュレータ上で快適に稼働している.
Scheme で書いたシミュレータは, スタックの代りにリストを使う. car 側がスタックトップである. 電卓実機はスタックを突き破ってポップも出来るが, リストではそうはいかぬ. また実機は 2 の補数の 64 ビットであったが, シミュレータは bignum である.
シミュレータには押しボタンはないから, それに相当する文字を決める. 0〜9 は文字も 0〜9. 十六進の A, B,..., F は小文字で a, b,..., f.
算術演算+, −, *, /はそのまま. 基数変換 Hex, Dec, Oct は H, D, O. 数値を区切る Ent は E. Pop, Dup, Exc, Chs はそれぞれ ^, ", :, _で, Sqt, Fct, Pow, JD はそれぞれ S, F, P, J である.

従って 3*(2+1) は, ‘3E2E1+*’ と入力する. リターンは無視.
拡張機能には, まず補助スタックが出来た. これも実態はリストである. 命令は 2 個
• U スタックをポップし, そこにあったものを補助スタックにプッシュする.
• N 補助スタックをポップし, そこにあったものをスタックにプッシュする.
こうすることで, 例えばスタックトップの 4 個を回転する: ‘:U:U:NN’ で (d c b a ...) を (c b a d ...) とすることが出来る. まず ‘:’ でスタックは (c d b a ...) になる; ‘U’ で c は補助スタックへ移り, (... c) (d b a ...) になる. 左側にあるリストが補助スタックで, スタックトップが右に書いてある. 2 つ目の ‘:’ でスタックは (b d a ...) になり, ‘U’ で (... c b) (d a ...) になり, 3 つ目の ‘:’ で (a d ...) になり, ‘NN’ で (...) (c b a d ...) になるのである.

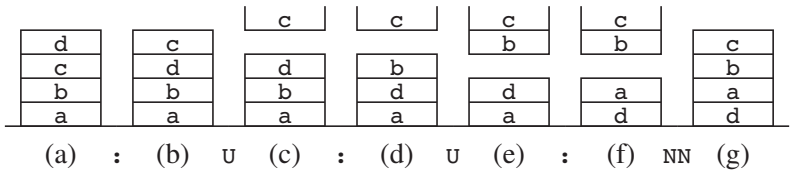

図 **2** スタック内の回転

こんなことをやっている内に, このプログラム列を記憶し, 繰り返し使いたくなった.
• = (,) 内の文字列を, 直前の文字命令として記憶する.
そこで ‘(:U:U:NN)R=’ と入力して, プログラム列を ‘R’ という名前で記憶する. ‘R’ で (d c b a ...) が (c b a d ...) になるので, ‘RRR’ とすると (d c b a ...) がこの 4 段の回転を 3 回実行し, (a d c b ...) になるのである.
また条件ジャンプが欲しくなった. EDSAC の negative jump の ‘G’ を使い,
• G スタックのトップから 2 段目が負なら, スタックトップの数だけプログラムを戻る.

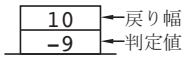

図 **3** ジャンプ命令

10 から 1 までの和

10 から 1 までの和を計算するプログラムは次のようだ.

```
 0123456789abcde ← 文字位置
(0Ua"N+U1-"_aG^N)X= と ‘X’ を定義し, ‘X’ を起動すると,

  0 (...)(0 ...)
  U (... 0)(...)
  a (... 0)(10 ...)       2 回目                    10 回目
a " (... 0)(10 10 ...)    (... 10)(9 9 ...)         (... 54)(1 1 ...)
9 N (...)(0 10 10 ...)    (...)(10 9 9 ...)         (...)(54 1 1 ...)
8 + (...)(10 10 ...)      (...)(19 9 ...)           (...)(55 1 ...)
7 U (... 10)(10 ...)      (... 19)(9 ...)           (... 55)(1 ...)
6 1 (... 10)(1 10 ...)    (... 19)(1 9 ...)         (... 55)(1 1 ...)
5 - (... 10)(9 ...)       (... 19)(8 ...)           (... 55)(0 ...)
4 " (... 10)(9 9 ...)     (... 19)(8 8 ...)         (... 55)(0 0 ...)
3 _ (... 10)(-9 9 ...)    (... 19)(-8 8 ...)        (... 55)(0 0 ...)
2 a (... 10)(10 -9 9 ...)(... 19)(10 -8 8 ...)      (... 55)(10 0 0 ...)
1 G (... 10)(9 ...)       (... 19)(8 ...)           (... 55)(0 ...)
0 ^                                                 (... 55)(...)
  N                                                 (...)(55 ...)
```

のように 10 までの和 55 がスタックに得られる. 文字位置 0 の ‘0’ は和の格納場所ですぐ補助スタックに移される. ‘a’ で 10 を置き, 複製し, 上の 10 を和に加え (‘+’) 補助スタックへ戻す. 残った 10 から 1 を引き, 複製し符号を変え, 10 文字戻る準備をして文字位置 c の ‘G’ 命令に来る. 判定値は負なので, スタックトップを複製する位置 3 まで 10 文字戻る. (位置勘定の基準は ‘G’ の次の d から.) これを繰り返し, 最後 0 になると負にしても 0 なのでジャンプせず, 0 を棄て, 補助スタックから 55 を取り下ろして終わる.

factorial

情報科学標準問題の 1 つに factorial がある.

```
(define (factorial n)
  (if (= n 0) 1
      (* n (factorial (- n 1)))))
```

個人用電卓のプログラムでも, この程度のことはやってみたい. プログラム名としては '!' を使う.
n がスタックトップにあるとき, これを複製し, 1 を引き, '!' で factorial をとり, '*' で掛ければよい. 問題は $n = 0$ の時には 1 を返す仕掛けを組み込むことである. 0 と >1 の判定は, 符号を反転して負になれば元は >1 であったわけだ. 従ってプログラムはこうなる.

```
("_6_G1+5_"G"1-!*)!=
      0123456        ← 文字位置
            012345   ← 文字位置
```

このプログラムの開始時, n を複製し, 符号反転する. その上に -6 を置き, ジャンプの準備をする. 'G' 命令で 6 文字右へ飛ぶ. プログラムの下の 'G' の隣りから 012... とあるのが飛び幅で, 次の 'G' の右の '"' まで行く.
そこで複製し, 1 を引き '!' で factorial を再帰呼出しする. $n-1$ の階乗がとれて戻ってきたら, '*' で掛ける. 一方 0 だったら, 'G' ではジャンプせず, n(つまり 0) に 1 を足し, 返す値 1 を作る. 次に -5 を 2 個積んで 'G' 命令でプログラムの終りまで飛ぶ. -5 を 2 個置くのは, 上は飛び幅, 下は負ジャンプを絶対ジャンプにする負数である.
factorial 3 の実行の様子は下の図の通り.

```
   ! (3)
   " (3 3)       (2 2 3)       (1 1 2 3)       (0 0 1 2 3)
   _ (-3 3)      (-2 2 3)      (-1 1 2 3)      (0 0 1 2 3)
   6 (6 -3 3)    (6 -2 2 3)    (6 -1 1 2 3)    (6 0 0 1 2 3)
   _ (-6 -3 3)   (-6 -2 2 3)   (-6 -1 1 2 3)   (-6 0 0 1 2 3)
   G (3)         (2 3)         (1 2 3)         (0 1 2 3)
 0 1                                           (1 0 1 2 3)
-1 +                                           (1 1 2 3)
-2 5                                           (5 1 1 2 3)
-3 _                                           (-5 1 1 2 3)
-4 "                                           (-5 -5 1 1 2 3)
-5 G                                           (1 1 2 3)
-6 " (3 3)       (2 2 3)       (1 1 2 3)
-1 1 (1 3 3)     (1 2 2 3)     (1 1 1 2 3)
-2 - (2 3)       (1 2 3)       (0 1 2 3)
-3 ! (2 3)       (1 2 3)       (1 1 2 3)
-4 * (6)         (2 3)         (1 2 3)
```

一番左は 'G' からの相対位置. 次は命令文字. '(3)' から始まる列は最初の実行トレース. 'G' 命令で下へ飛ぶ. そのうち, '!' 命令になると, 次に右の列の '"' から階乗を再帰実行する. 3 回目の再帰実行は, スタックが 0 の時で, 処理の流れが異なる. つまり 'G' 命令でジャンプせず, 1 を置いて (正確には 0 に 1 を足して) 飛び出す. 戻ると 1 つ左のの列の最下行 '*' に来, 1 と 1 を掛け 1 にして飛び出す. また次の左の列の最下行 '*' で 1 と 2 を掛け, さらに左の列で 2 と 3 を掛け, 結果は 6 になった.

復活祭公式

もう少し複雑な計算の例として, (TAOCP の演習問題 1.3.2-14)Date of Easter をやってみよう. Scheme 版のアルゴリズムは以下のようだ.

```
(define (date-of-easter year)
 (let* ((golden-number (+ (modulo year 19) 1))
        (century (+ (quotient year 100) 1))
        (x (- (quotient (* 3 century) 4) 12))
```

```
      (z (- (quotient (+ (* 8 century) 5) 25) 5))
      (domino (- (quotient (* 5 year) 4) x 10))
      (epact (modulo (+ (* 11 golden-number) 20 z (- x)) 30)))
 (if (or (and (= epact 25) (> golden-number 11)) (= epact 24))
     (set! epact (+ epact 1)))
 (let ((n (- 44 epact)))
  (if (< n 21) (set! n (+ n 30)))
  (set! n (+ n 7 (- (modulo (+ domino n) 7))))
  (if (> n 31) (list 'april (- n 31)) (list 'march n)))))
```

上の復活祭公式は次のようなプログラムに変換される.

```
("U"100/1+:^"3*4/12-"N5*4/:^:-10-U:^U)A=
(8*5+25/5-U^19/^1+"11*20+N+N-30/^:U)B=
("24-27_G"_25+18_G"25-9_GN"U12-2_G1+N^)C=
(44:-"_20+3_G30+"U7+NN+7/^-"32-3_G69+300+)K=
```

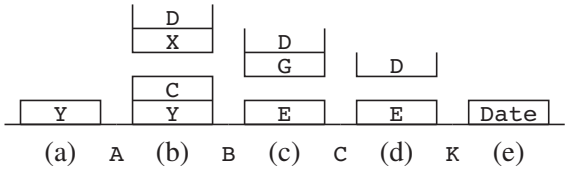

図 4 復活祭公式計算の流れ

計算の進行状況は次のようだ. 左端の斜体は行番号, 次が入力で, その後はリストが 2 個なら, 演算後の補助スタックとスタックの内容である. リストが 1 個なら補助スタックは空である.

0 2011(2011) スタックに year(Y) を置き
1 A(2011) A を起動
2 "(2011 2011)
3 U(2011)(2011)
4 "(2011)(2011 2011)
5 100(2011)(100 2011 2011)
6 /(2011)(20 11 2011) $\lfloor Y/100 \rfloor$
7 1(2011)(1 20 11 2011)
8 +(2011)(21 11 2011) $+1 = C$
9 :(2011)(11 21 2011)
10 ^(2011)(21 2011) 剰余を棄てる
11 "(2011)(21 21 2011)
12 3(2011)(3 21 21 2011)
13 *(2011)(63 21 2011) 3C
14 4(2011)(4 63 21 2011)
15 /(2011)(15 3 21 2011) $\lfloor 3C/4 \rfloor$
16 12(2011)(12 15 3 21 2011)
17 -(2011)(3 3 21 2011) $-12 = Z$
18 "(2011)(3 3 3 21 2011)
19 N(2011 3 3 3 21 2011)
20 5(5 2011 3 3 3 21 2011)
21 *(10055 3 3 3 21 2011) 5Y
22 4(4 10055 3 3 3 21 2011)
23 /(2513 3 3 3 3 21 2011) $\lfloor 5Y/4 \rfloor$
24 :(3 2513 3 3 3 21 2011)
25 ^(2513 3 3 3 21 2011) 剰余を棄てる
26 :(3 2513 3 3 21 2011)
27 -(2510 3 3 21 2011) $-X$
28 10(10 2510 3 3 21 2011)
29 -(2500 3 3 21 2011) $-10 = D$
30 U(2500)(3 3 21 2011)
31 :(2500)(3 3 21 2011)
32 ^(2500)(3 21 2011) 3C/4 の剰余を棄てる
33 U(2500 3)(21 2011) (D X) (C Y)
34 B(2500 3)(21 2011) B を起動
35 8(2500 3)(8 21 2011)
36 *(2500 3)(168 2011) 8C
37 5(2500 3)(5 168 2011)
38 +(2500 3)(173 2011) +5
39 25(2500 3)(25 173 2011)
40 /(2500 3)(6 23 2011) $\lfloor (8C+5)/25 \rfloor$
41 5(2500 3)(5 6 23 2011)
42 -(2500 3)(1 23 2011) $-5 = Z$
43 U(2500 3 1)(23 2011)
44 ^(2500 3 1)(2011) 剰余を棄てる
45 19(2500 3 1)(19 2011) (D X Z) (Y)
46 /(2500 3 1)(105 16) $\lfloor Y/19 \rfloor$
47 ^(2500 3 1)(16) 商を棄てる
48 1(2500 3 1)(1 16)
49 +(2500 3 1)(17) $+1 = G$
50 "(2500 3 1)(17 17)
51 11(2500 3 1)(11 17 17)
52 *(2500 3 1)(187 17) 11G
53 20(2500 3 1)(20 187 17)
54 +(2500 3 1)(207 17) +20
55 N(2500 3)(1 207 17)
56 +(2500 3)(208 17) +Z
57 N(2500)(3 208 17)
58 -(2500)(205 17) $-X$
59 30(2500)(30 205 17)
60 /(2500)(6 25 17) $\lfloor (11G+20+Z-X)/30 \rfloor$
61 ^(2500)(25 17) 商を棄てる
62 :(2500)(17 25)
63 U(2500 17)(25) (D G) (E)
64 C(2500 17)(25) C を起動
65 "(2500 17)(25 25)
66 24(2500 17)(24 25 25)
67 -(2500 17)(1 25) E-24
68 27(2500 17)(27 1 25)
69 _(2500 17)(-27 1 25) 飛先き 94 行目
70 G(2500 17)(25) E<24 なら飛ぶ
71 "(2500 17)(25 25)
72 _(2500 17)(-25 25)
73 25(2500 17)(25 -25 25)
74 +(2500 17)(0 25) 25-E
75 18(2500 17)(18 0 25)
76 _(2500 17)(-18 0 25) 飛先き 94 行目
77 G(2500 17)(25) 25<E なら飛ぶ
78 "(2500 17)(25 25) E=24or25
79 25(2500 17)(25 25 25)

80 -(2500 17)(0 25) E-25
81 9(2500 17)(9 0 25)
82 _(2500 17)(-9 0 25) 飛先き 92 行目
83 G(2500 17)(25) E¡25 なら飛ぶ
84 N(2500)(17 25)
85 "(2500)(17 17 25)
86 U(2500 17)(17 25)
87 12(2500 17)(12 17 25)
88 -(2500 17)(5 25) G-12
89 2(2500 17)(2 5 25)
90 _(2500 17)(-2 5 25) 飛先き 94 行目
91 G(2500 17)(25) G¡12 なら飛ぶ
92 1(2500 17)(1 25)
93 +(2500 17)(26) E+1
94 N(2500)(17 26)
95 ^(2500)(26) G を棄てる (D) (E)
96 K(2500)(26) K を起動
97 44(2500)(44 26)
98 :(2500)(26 44)
99 -(2500)(18) 44−E=N
100 "(2500)(18 18)
101 _(2500)(-18 18)
102 20(2500)(20 -18 18)
103 +(2500)(2 18) 20−N
104 3(2500)(3 2 18)
105 _(2500)(-3 2 18) 飛先き 109 行目
106 G(2500)(18) 20 <N なら飛ぶ
107 30(2500)(30 18)
108 +(2500)(48) +30
109 "(2500)(48 48)
110 U(2500 48)(48) (D N) (N)
111 7(2500 48)(7 48)
112 +(2500 48)(55) N+7
113 N(2500)(48 55)
114 N(2500 48 55)
115 +(2548 55) N+D
116 7(7 2548 55)
117 /(364 0 55) ⌊(N+D/7⌋
118 ^(0 55) 商は棄てる
119 -(55)
120 "(55 55)
121 32(32 55 55)
122 -(23 55) N+7+(N+D)%7-32
123 3(3 23 55)
124 _(-3 23 55) 飛先き 128 行目
125 G(55) < 32 なら飛ぶ
126 69(69 55) 69=100-31
127 +(124)
128 300(300 124) 300=3 月
129 +(424) 4 月 24 日

アドレス計算とコンパイラ

例えば 1 から 10 までの和のプログラムは, '(0Ua"N+U1-"_aG^N)X=', factorial は '("_6_G1+5_"G"1-!*)!=' という具合いである.
'G' の命令について復習すると, スタックの上から 2 段目の内容が負なら, プログラム制御をスタック最上段の数だけバックする (左へ進む), 非負ならそのまま進むというのであった. そこで, プログラムを書くのに面倒なのは, 飛び幅の計算である. factorial のプログラムの左から 5 文字目の 'G' はスタックの最上段に −6 があるので, その下が負なら, 右へ '(1+5_"G)' の 6 文字を飛び越えることを示す. つまり右の 'G' の次の '"' のところだ. 当時のプログラムでは, 長さのところを空けておき, 他が出来てから手作業で幅を数えていた.

図 5 飛び先のアドレス

またこの図の左のようだと, 'G' 命令から 10 文字左へ飛び, 右のようだと 9 文字左へ行く. それが同じ場所である. 従って, 工夫によってはプログラムが短く出来る. これをなんとか自動化したい, つまりコンパイラを書きたいとかねがね考えていた.

飛び先までの間に条件付きの branch 命令や無条件の jump 命令があると, そのパラメータの文字数が未定であったりするので, 自縄自縛 (という表現が適切かな) になり, コンパイラの書き方が決らなかった.
最近, 思いついた方法があり, それでプログラムを生成できるようになった.
コンパイルされるプログラムはこんな形をしている.

```
(define sum '(0 up 10 'foo dup down + up 1 - dup neg
 (branch foo) pop down 'exit))

(define factorial '(dup neg (branch bar) 1 + (jump exit) 'bar
 dup 1 - ! * 'exit))

(define ackermann '(dup 1 - (branch a) exch dup 1 - (branch b)
exch dup 2 - (branch c)
up dup 1 - exch down 1 - A A (jump exit)
'c pop pop 2 ent (jump exit)
'b pop 2 * (jump exit)
'a pop pop 0
'exit))
```

```
(define (A x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (else (A (- x 1)
                 (A x (- y 1))))))
```

sum が 10 までの和のプログラムで, クォートされているリスト内がプログラム本体である. 0 とか 10 は定数で, そのまま文字に置き換わる. 今の版ではコンパイラの関数によって十進だったり十六進数だったりする. up, dup, down, +, -, neg などが, プログラムの命令だ. (この Ackermann 関数は, SICP の ex1.10 にあるものだ.)
'(branch foo)' は 'foo' への branch 命令を挿入することを示す. その飛び先は 'foo' のところである. jump 命令では, スタックの判定用に無理に負の数をおいて常に飛ぶのである. (branch, jump 命令は数字で始まるので, その前が数値なら ent が必要.)

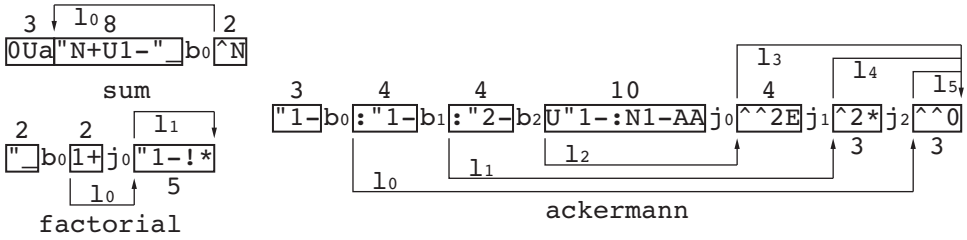

図 **6** アドレス計算図

それぞれの図の箱は, 長さ固定のプログラム. その間の b_m やその間の j_m は branch や jump 命令. 矢印はそこから飛ぶ飛び先で, l_m は飛ぶ幅である.
b_m や j_m は, 対応する飛ぶ幅 l_m を文字に変換した l について,
a. 左への branch は l の次に 'G' の 1 文字,
b. 右への branch は l の次に '_G' の 2 文字,
c. 左への jump は '1_' の 2 文字と l の次に 'G' の 1 文字,
d. 右への jump は l の次に '_"G' の 3 文字
をつける. つまり追加文字数は, a, b, c, d のそれぞれで, 1, 2, 3, 3 である.
ackermann のプログラムは, 二進十進 (あるいは十六進) 変換を (define (f n) (if (< n 10) 1 (if (< n 100) 2 3))) と定義し,

```
(define (iter b0 b1 b2 j0 j1 j2)
 (let* ((l0 (+ 4 b1 4 b2 10 j0 4 j1 3 j2)) (l1 (+ 4 b2 10 j0 4 j1))
        (l2 (+ 10 j0)) (l3 (+ 4 j1 3 j2 3)) (l4 (+ 3 j2 3)) (l5 3)
        (c0 (+ (f l0) 2)) (c1 (+ (f l1) 2)) (c2 (+ (f l2) 2))
        (k0 (+ (f l3) 3)) (k1 (+ (f l4) 3)) (k2 (+ (f l5) 3)))
  (if (equal? (list c0 c1 c2 k0 k1 k2) (list b0 b1 b2 j0 j1 j2))
      (list b0 b1 b2 j0 j1 j2 l0 l1 l2 l3 l4 l5)
      (iter c0 c1 c2 k0 k1 k2))))
```

のプログラムで '(iter 0 0 0 0 0 0)' とすると, (4 4 4 5 5 4 47 32 15 19 10 3) が得られる. したがって b_0 は '47_G', b_1 は '32_G', b_2 は '15_G', j_0 は '19_"G', j_1 は '10_"G', j_2 は '3_"G ' となる.
factorial は

```
(define (iter b0 j0)
 (let* ((l0 (+ 2 j0)) (l1 5)
        (c0 (+ (f l0) 2)) (k0 (+ (f l1) 3)))
  (if (equal? (list c0 k0) (list b0 j0))
      (list b0 j0 l0 l1)
      (iter c0 k0))))
```

とし, (iter 0 0) で, (3 4 6 5) が得られるから, b_0 は '6_G', j_0 は '5_"G' でよい.
10 までの和は,(2 9) が得られ, b_0 は '9G' となる.
以上は, それぞれの関数用のプログラムを実行した例だが, プログラムの半ばコンパイルした中間出力のリストを使うことも出来るようになった. ackermann 関数の例では, b_m や j_m をリストにし, プログラムの文字列 (図 6 の箱) にその長さ (箱の上の数) を追加した中間出力

```
(("1- 3) (br a 1 14 0) (:"1- 4) (br b 3 11 0) (:"2- 4) (br c 5 8 0)
 (U"1-:N1-AA 10) (jr exit 7 16 0) (quote c) (^^2E 4) (jr exit 10 16 0)
 (quote b) (^2* 3) (jr exit 13 16 0) (quote a) (^^0 3) (quote exit))
```

から

```
"1-47_G:"1-32_G:"2-15_GU"1-:N1-AA19_"G^^2E10_"G^2*3_"G^^0
```

を得, これを ‘A=’ で定義し, ‘1E10A’ と起動すると 1024 になる.

新インターフェース

デュアルスタック, プログラム機能つき電卓を, iPhone のアプリとして開発するのは難しいらしい. こういうアプリを認めていないからだ. しかしインターフェースを考えるのは楽しい.

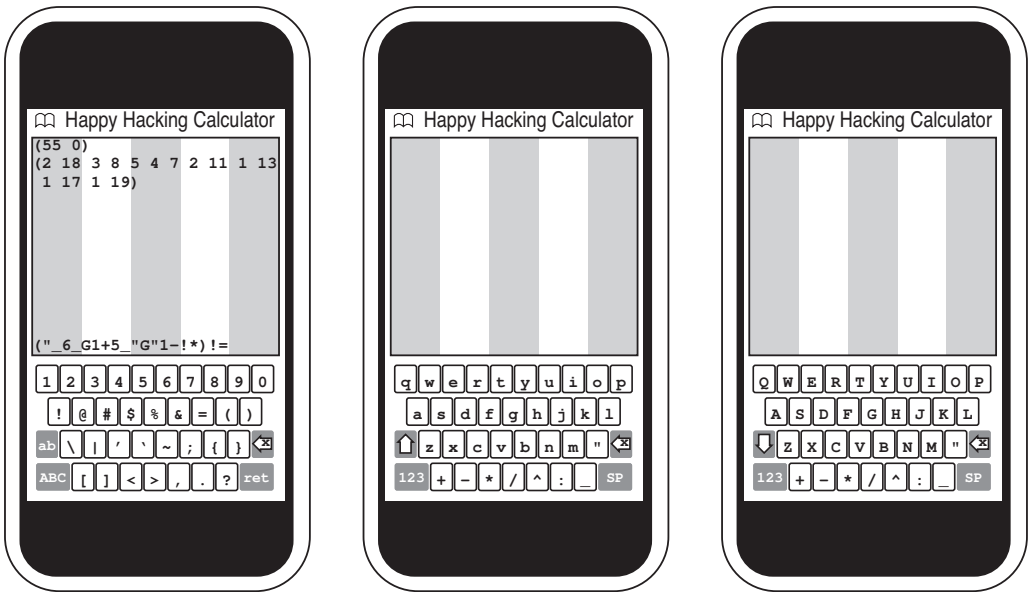

図 7 新インターフェースのデザイン

左から記号モード, 小文字モード, 大文字モードの画面である. それぞれキーは上段から下段へ 10 個, 9 個, 8 個, 7 個だから計 34 個. モード全体で 102 個になる. ASCII コードの図形文字は 94 個だから 8 個多いが, それは下段の ‘+ - * / ^ : _’ とその上右端の ‘"’ の 8 個 (四則演算と ‘Pop’, ‘Exc’, ‘Chs’, ‘Dup’) が小文字, 大文字の両方にあるからだ.
中央上部の窓は, 横 25 字, 縦 12 行の入出力表示用である. また, 左上の 🕮 は, インターネット接続その他のメニュー表示用である.

参考文献

Happy Hacking Calculator の英文と和文のマニュアルは
http://www.iijlab.net/~ew/manualen.pdf
http://www.iijlab.net/~ew/manualjp.pdf
においてある.