



Utilに **break**, **continue** などを導入するために、Expr の定義に次のように構  
成子を追加する。また、**goto** 文を導入するため、ラベルも導入する。

ファイル ContType.hs

```

1  data Expr = Const Target | Var String
2          | If Expr Expr Expr | While Expr Expr
3          | Let [Decl] Expr | Val Decl Expr
4          | Lambda String Expr | Delay Expr | App Expr Expr
5          -- ここまでは、Util1と同じ
6          | Begin [LabeledExpr]    -- ブロック
7          | Break                  -- <span class='courierb'>break
8          | Continue               -- <span class='courierb'>conti
9          | Goto String             -- <span class='courierb'>goto<
10         deriving Show
11  type LabeledExpr = (Maybe String, Expr) -- ラベル付きの式
12

```

これに対する具象構文としては、

```

Expr          → begin LabeledExprSeq
              | break | continue | goto Var
LabeledExprSeq → LabeledExpr end | LabeledExpr ; LabeledExprSeq
LabeledExpr   → Expr | Var : Expr

```

を想定する。

実際の UtilCont では接続とともに状態や入出力も扱いたいので、計算のモナド  
は、K そのものではなく、“状態の変化”を“結果”として持つ K (ST (WithIO s)  
v) a (と同型の型) とする。ここで、s は状態の型である。

ファイル Cont.hs

```

1  newtype KST v s a = KST ( _____ )
2
3
4  unKST :: KST v s a -> (a -> WithIO s -> (v,WithIO s))
5         -> WithIO s -> (v,WithIO s)
6  unKST (KST m) = m
7

```

この KST の定義にあわせて、set など状態に関する関数も書き直しておく。

ファイル Cont.hs

```

1  instance MyState (KST v) where
2    get p   = KST (\ c (s,i,o) -> c (fst (p s)) (s,i,o))
3    set p v = KST (\ c (s,i,o) -> c () (snd (p s) v,i,o))
4
5  instance MyStream (KST v s) where
6    readChar = KST (\ c (s,ch:i,o) -> c ch (s,i,o))
7    eof      = KST (\ c (s,i,o) -> c (null i) (s,i,o))
8    writeStr v = KST (\ c (s,i,o) -> c () (s,i,o ++ v))
9
10 abort :: (WithIO s -> (v,WithIO s)) -> KST v s a
11 abort r = KST ( _____ )
12

```

すると、set p v は状態の p で表される位置に v をセットし、() と新しい状態  
を接続に渡す。



<pre> 1  foo = \ y -&gt; begin 2    set xP 1; set yP y; 3    while get yP &gt; 0 do begin 4      val x = get xP in 5      val y = get yP in 6      if y == 10 then break 7      else if y == 3 then begin 8        set yP (y - 1); continue 9      end else (); 10     set xP (x * y); 11     set yP (y - 1) 12   end; 13   get xP 14 end 15 </pre>	<pre> 1  int foo(int y) { 2    int x = 1; 3    while (y &gt; 0) { 4 5      if (y == 10) break; 6      else if (y == 3) { 7        y--; continue; 8      } 9      x = x * y; 10     y--; 11   } 12   return x; 13 } 14 </pre>
---	--

をコンパイルすると、次の Haskell プログラムが得られる。

```

1  foo = \ y ->
2    set xP 1           >>= \ _ ->
3    set yP y          >>= \ _ ->
4    KST (\ _break ->
5      let KST _while
6        = get yP           >>= \ y ->
7          if y > 0 then
8            get xP         >>= \ x ->
9            get yP         >>= \ y ->
10           (if y == 10 then abort (_break ()) else
11            if y == 3 then
12              set yP (y - 1) >>= \ _ ->
13              abort (_while _break)
14            else return () >>= \ _ ->
15            set xP (x * y) >>= \ _ ->
16            set yP (y - 1) >>= \ _ ->
17            KST _while
18          else return ()
19        in _while _break) >>= \ _ ->
20    get xP
21

```

この foo の型は Integer -> KST a (Integer,Integer) a Integer であるから、値を取り出すためには、整数と初期接続（通常、\ a s -> (a,s)）、初期状態（((0,0), "", "")）を渡す必要がある。ここで

```

1  evalKST st s = fst (unKST st (\ a s -> (a,s)) (s, "", ""))
2

```

と定義すると、evalKST (foo 9) (0,0) の結果は、\_\_\_\_\_ に、evalKST (foo 11) (0,0) は \_\_\_\_\_ になる。（参考: 9! = 362880）

さらに goto に対する意味を与えるためには、ブロック (begin ~ end) のなかで、“ラベル”に適切な接続を与える必要があるが、Begin に対する comp の定義は長くなってしまうので、ここに示さず、変換前と変換後の形の例のみを示す。

ソース (Util)	ターゲット (Haskell)
<b>begin</b>	KST (\ _end -> let
lbl1: s <sub>1</sub>	lbl1 = \ _ -> unKST s <sub>1</sub> lbl2
lbl2: s <sub>2</sub>	lbl2 = \ _ -> unKST s <sub>2</sub> lbl3
lbl3: s <sub>3</sub>	lbl3 = \ _ -> unKST s <sub>3</sub> _end
<b>end</b>	in lbl1 ())

この  $s_1, s_2, s_3$  の中には、`goto lbl1`, `goto lbl2`, `goto lbl3` が含まれているかもしれない。ターゲット (Haskell) プログラム中の識別子 `lbl1, lbl2, lbl3` に束縛されているのはそれぞれ、同名のラベル `lbl1, lbl2, lbl3` に対応する接続である。

例えば、次の UtilCont プログラム (右は対応する C プログラム) :

<pre> 1 bar = \ _ -&gt; begin 2   set xP 1; 3   label1: 4     if get xP &gt; 100 then goto label2 5       else (); 6     set xP (get xP * 2); 7     goto label1; 8   label2: 9     get xP 10  end 11 </pre>	<pre> 1 int bar(void) { 2   int x = 1; 3   label1: 4     if (x &gt; 100) 5       goto label2; 6     x = x * 2; 7     goto label1; 8   label2: 9     return x; 10 } 11 </pre>
---	--

は次のような Haskell プログラムにコンパイルされる。

```

1 bar = \ _ ->
2   KST (\ _end ->
3     let label1 = \ _ -> unKST
4       (get xP >>= \ x ->
5         (if x > 100 then abort (label2 ())
6           else return ()) >>= \ _ ->
7         get xP >>= \ x ->
8         set xP (x * 2) >>= \ _ ->
9         abort (label1 ())) label2
10    label2 = \ _ -> unKST (get xP) _end
11    in unKST (set xP 1) label1)
12

```

そして、`evalKST (bar ()) (0,0)` を評価すると、結果は      になる。

問 6.2.1 次の C の関数とほぼ同等な Haskell の関数をモナドを用いて作成せよ。

```

1 int hoge(int n) {
2   int i = 1, sum = 0;
3   while (i <= n) {
4     sum = sum + i;
5     if (sum > 21) {
6       sum = 0;
7       break;
8     }
9     i = i + 1;
10  }
11  return sum;
12 }
13

```

ごちゃごちゃと `goto` 文を多用したスパゲッティコードは、結局、相互再帰を多用することと同等であり、プログラムの意味がわかりにくくなる。

### 6.3 callcc とは

ここで紹介する `callcc` は Scheme や Ruby などが採用している、プログラマーが接続を直接操作することができるプリミティブである。Scheme では `call-`



```
5 decrease n k =
6   if n < 0 then ()
7   else begin writeStr " d:"; write n;
8     decrease (n - 1) (callcc k) end
9
```

という2つの関数を定義して

```
1 increase 0 (decrease 10)
2
```

という式を実行すると、



と出力される。

注意: 実は、この Util プログラムを Haskell に変換すると型エラーになり、そのままではコンパイル・実行できない。これは、callcc の引数の型と戻り値の型が同一になる必要があるからである。

いくつかトリッキーな変換をすると、意味を変えずに型付け可能な定義に書き換えが可能で、上記のような実行結果になる。しかし、ここはコルーチンのアイデアを説明するのが本旨なので、型付けをするためのトリックの詳細には立ち入らないことにする。

なお、Scheme では、

```
1 (define (increase n k)
2   (if (> n 10) '()
3     (begin (display " i:") (display n)
4             (increase (+ n 1) (call/cc k)))))
5 (define (decrease n k)
6   (if (< n 0) '()
7     (begin (display " d:") (display n)
8             (decrease (- n 1) (call/cc k)))))
9
```

のように対応する関数を定義して

```
1 (increase 0 (lambda (k) (decrease 10 k)))
2
```

という式を実行すると上記の出力が得られる。

ここでは increase と decrease という2つの関数が交互に実行されていることがわかる。スレッドと似ているが、2つのルーチンが同時に実行される訳ではない。

また callcc は、コルーチンの他にこれまでに紹介したエラー処理 (try ~ catch) や非決定性などのプリミティブも、callcc を用いて定義できることがわかっている。ある意味でオールマイティーなプリミティブである。しかし、その詳細の解説については、ここでは割愛する。

## 6.5 callcc の表現

我々の言語 UtilCont に callcc を導入するには、接続を関数として渡すためのコードを用意すれば良い。callcc に対応する関数の定義は次のようになる。  
 ファイル Cont.hs

```

1  callcc :: ((a -> KST v s b) -> KST v s a) -> KST v s a
2  callcc h = KST (
3
4  -- KST, unKST がなければ
5  -- callcc h = \ c -> let k a = \ d -> c a
6  --                      in h k c
7

```

この callcc の定義中で用いられている k は現在の接続 (d) を捨て、キャプチャーされた接続 (c) を呼び出すという関数である。

コンパイラーは単に callcc という名前の UtilCont の関数を Haskell の callcc にコンパイルすれば良い。

ソース (Util)	ターゲット (Haskell)
callcc m	m >>= \ _x -> callcc _x

また、head, tail, null, not, show などの 1 引数で副作用を持たない関数は、次のようにコンパイルされる。

ソース (Util)	ターゲット (Haskell)
funWithOneArg m	m >>= \ _x -> return (funWithOneArg _x)

例えば、先に紹介した UtilCont プログラム multlist をコンパイルすると、次の Haskell プログラムが得られる。

```

1  multlist = \ xs ->
2    let aux = \ xs ->
3      return (\ k ->
4        set xP 1 >>= \ _ ->
5        set yP xs >>= \ _ ->
6        KST (\ _break ->
7          let KST _while
8            = get yP >>= \ y ->
9              if not (null y) then
10               get yP >>= \ y ->
11               (if head y == 0 then k 0 else
12                get xP >>= \ x ->
13                 set xP (x * head y)
14                  >>= \ _ ->
15                  set yP (tail y) >>= \ _ ->
16                  writeStr " " >>= \ _ ->
17                  write (head y)) >>= \ _ ->
18                 KST _while
19                else return ()
20               in _while _break) >>= \ _ ->
21             get xP)
22    in callcc (\ k -> aux xs >>= \ _f ->
23      _f k) >>= \ result ->
24    writeStr "; result = " >>= \ _ ->
25    write result
26

```

このとき

```

1  evalKSTIO st s =

```



```
2 | let (_,(_,_,o)) = unKST st (\ a s -> (a,s)) (s, "", "")
3 |   in o
4 |
```

と定義すると、`evalKSTIO (multlist [1,2,3,4,5]) (0,[])` は、`" 1 2 3 4 5; result = 120"` を表示し、

一方、`evalKSTIO (multlist [1,2,3,0,4,5]) (0,[])` は`" 1 2 3; result = 0"`となる。つまり、0 が現れた時点で乗算を打ち切っていることがわかる。

## 6.6 さらに詳しく知りたい人のために...

接続に関する文献は数多くあるが、[1] は接続の「発見」について、振り返っている珍しいものである。文献 [2] は、`call/cc` がある意味で「オールマイティー」であることについての説明を与えている。文献 [3] は、Java などの命令型言語に、`call/cc` のような接続を扱うオペレーターを導入する方法を述べている。文献 [4] は `mfixU` などの不動点演算子について解説している。

1. John C. Reynolds, "The Discoveries of Continuations" *Lisp and Symbolic Computation*, 6, (233–247). 1993 年
  2. Andrzej Filinski, "Representing Monads" 21st ACM Symposium on Principles of Programming Languages. 1994 年
  3. T. Sekiguchi, T. Sakamoto, and A. Yonezawa, "Portable Implementation of Continuation Operators in Imperative Languages by Exception Handling" *Advances in Exception Handling Techniques*. Springer-Verlag, LNCS 2022. 2001年 <http://www.yl.is.s.u-tokyo.ac.jp/amo/>
  4. Levent Erköök, and John Launchbury, "Recursive Monadic Bindings" *Proc. of the International Conference on Functional Programming*. 2000 年
-