CLU 言語入門　＃５

久野　靖

1993.6.24

1　例外

CLU の制御構造の特徴の１つ、例外についてなかなかやる機会がありませんでした。実は、手続きも反復子も正常に返る場合の他に複数の「例外」状態を返すことができます。例外には複数の値を付随させることもできます。どんな例外を返し、どの例外にはどんな値が付随するかは手続きや反復子のヘッダに書きます。すなわち、正確にはヘッダは次の形をしています。

```
名前 = proc(...) returns(...) signals(例外名(型,...),...)
名前 = iter(...) yields(...) signals(例外名(型,...),...)
```

そして、例外を返すには return 文の代わりに

```
signal 例外名(値, ...)
```

を使います。

一方、例外を受け止めるには例外文を使います。例外文とは「例外ハンドラのついた文」のことで、次の形をしています。

```
文
  except
    when 例外名(名前:型,...): 文...
    when 例外名(名前:型,...): 文...
    ...
    others [(名前:string)]: 文...
  end
```

これは、「文」のなかで例外が返されたらハンドラの対応する when のところへ来て処理を続けるという意味です。例えば、「隣接する２行ずつをくっつけるフィルタ」を考えてみましょう。

```
% sam12.clu -- concat adjacent lines

start_up = proc()
  pi:stream := stream$primary_input()
  po:stream := stream$primary_output()
  while ~stream$empty(pi) do
    line:string := stream$getl(pi)
    if ~stream$empty(pi) then
      stream$putl(po, line||stream$getl(pi))
    end
  end
end start_up
```

これでもいいけど、終りの判定が２箇所なので少しごちゃごちゃします。そこで、stream はファイルの終りだと end_of_file という例外を返すことを利用して構わず読むように作ります。

```
% sam12b.clu -- concat adjacent lines, with exception

start_up = proc()
  pi:stream := stream$primary_input()
  po:stream := stream$primary_output()
  while true do
    l1:string := stream$getl(pi)
    l2:string := stream$getl(pi)
    stream$putl(po, l1||l2)
  end
    except when end_of_file: end
end start_up
```

この方が「わかりやすい」ですよね? 例外を返せることの利点は

1. 例外的な制御の流れをごちゃごちゃ書かないで済む。
2. 失敗しないかどうか調べてから呼ぶという無駄をしなくて済む。
3. 呼ばれる側で期待される値が存在しない時苦し紛れの値を返さずに済む。
4. 成功/失敗を表す旗などを返さずに済む。

といったところでしょうか。なお、others は当然「その他」で、その場合例外名を文字列引数として受けとれます。

2 手続き引数

さて、CLU の型もそろそろおしまいで、最後は手続き型と反復子型です。これは簡単で、

```
proctype(型,...) returns(型,...) signals(型,...)
itertype(型,...) yields(型,...) signals(型,...)
```

です。これを使って、「整列順を渡す sort」を書いてみました。

```
% sam13.clu -- procedure parameter

start_up = proc()
  ent = record[s:string, i:int]
  aent = array[ent]
  pi:stream := stream$primary_input()
  po:stream := stream$primary_output()
  a:aent := aent$new()
  while ~stream$empty(pi) do
    s:string := stream$getl(pi)
    i:int := int$parse(stream$getl(pi))
    e:ent := ent${s:s, i:i}
    aent$addh(a, e)
  end
  sort_record(a, rec_order)
  for e:ent in aent$elements(a) do
    stream$putright(po, int$unparse(e.i), 5)
    stream$putc(po, ' ')
    stream$putl(po, e.s)
  end
end start_up

sort_record = proc(a:aent, o:proctype(ent,ent)returns(bool))
```

```
  ent = record[s:string, i:int]
  aent = array[ent]
  done:bool := false
  while ~done do
    done := true
    for i:int in int$from_to(aent$low(a), aent$high(a)-1) do
      if ~o(a[i], a[i+1]) then
        e:ent := a[i]; a[i] := a[i+1]; a[i+1] := e; done := false
      end
    end
  end
end sort_record

rec_order = proc(e1,e2:ent) returns(bool)
  ent = record[s:string, i:int]
  return(e1.s > e2.s)
end rec_order
```

3 型パラメタつきモジュール

さて、先の例だとまだ sort_record は汎用じゃありませんよね? つまり ent 型の配列しか整列できない。そうじゃなくするには、sort に型パラメタを設けて、型パラメタとして要素の型を渡すようにすればよいわけです。

```
% sam14.clu -- parametrized procedure

start_up = proc()
  ent = record[s:string, i:int]
  aent = array[ent]
  pi:stream := stream$primary_input()
  po:stream := stream$primary_output()
  a:aent := aent$new()
  while ~stream$empty(pi) do
    s:string := stream$getl(pi)
    i:int := int$parse(stream$getl(pi))
    e:ent := ent${s:s, i:i}
    aent$addh(a, e)
  end
  sort[ent](a, rec_order)
  for e:ent in aent$elements(a) do
    stream$putright(po, int$unparse(e.i), 5)
    stream$putc(po, ' ')
    stream$putl(po, e.s)
  end
end start_up

sort = proc[t:type](a:array[t], o:proctype(t,t)returns(bool))
  done:bool := false
  while ~done do
    done := true
    for i:int in int$from_to(array[t]$low(a), array[t]$high(a)-1) do
      if ~o(a[i], a[i+1]) then
        e:t := a[i]; a[i] := a[i+1]; a[i+1] := e; done := false
      end
    end
  end
end sort
```

```
rec_order = proc(e1,e2:ent) returns(bool)
  ent = record[s:string, i:int]
  return(e1.s > e2.s)
end rec_order
```

練習 1 整数の配列 $i_1, i_2, ..., i_n$ と関数 op:　int × int → int を渡すと $i_1 op i_2 op ... op i_n$ を返す手続きを書け。それを使って、配列の全要素の和、全要素の積、最大値を求めてみよ。

練習 2 上の問題で「整数」に固定する代わりに任意の型 t の配列とし、関数も t × t → t としてみよ。