Example: Recognizing phone numbers

Consider the problem of recognizing phone numbers in a variety of formats:

```
555-1212
(520) 555-1212
520-555-1212
<any of the above formats> x <number>
```

This problem can be approached by using procedures that execution can backtrack through.

Here is a procedure that matches a series of N digits:

```
procedure digits(N)
    suspend (move(N) -- &digits) === ''
end
```

If a series of N digits is not found, `digits(N)` fails and the move is undone:

```
][ "555-1212" ? { digits(3) & snap() } ;
&subject =  5 5 5 - 1 2 1 2
&pos =  4         |

][ "555-1212" ? { digits(4) & snap() } ;
Failure
```

Phone numbers, continued

For reference:

```
procedure digits(N)
    suspend (move(N) -- &digits) === ''
end
```

Using `digits(N)` we can build a routine that recognizes numbers like 555-1212:

```
procedure Local()
    suspend digits(3) & ="-" & digits(4)
end
```

If `Local()` is resumed, the moves done in both `digits()` calls are undone:

```
][ "555-1212" ? { Local() & snap() } ;
&subject =  5 5 5 - 1 2 1 2
&pos =  9
          |
][ "555-1212" ? { Local() & snap("A") & &fail;
                  snap("B") } ;
A
&subject =  5 5 5 - 1 2 1 2
&pos =  9                  |
B
&subject =  5 5 5 - 1 2 1 2
&pos =  1  |
```

IMPORTANT:
Using `suspend`, rather than `return`, creates this behavior.

Phone numbers, continued

Numbers with an area code such as 520-555-1212 are recognized with this procedure:

```
procedure ac_form1()
    suspend digits(3) & ="-" & Local()
end
```

The (520) 555-1212 case is handled with these routines:

```
procedure ac_form2()
    suspend =”(" & digits(3) & =")" &
            optblank() & Local()
end

procedure optblank()
    suspend =(" "|"")
end
```

All three forms are recognized with this procedure:

```
procedure phone()
    suspend Local() | ac_form1() | ac_form2()
end
```

Phone numbers, continued

A driver:

```
procedure main()
    while writes("Number? ") &
            line := read() do {
        line ? if phone() & pos(0) then
                    write("yes")
                else
                    write("no")
        }
end
```

Usage:

```
% phone
Number? 621-6613
yes
Number? 520-621-6613
yes
Number? 520 621-6613
no
Number? (520) 621-6613
yes
Number? (520)  621-6613
no
Number? 555-1212x
no
```

Phone numbers, continued

Problem: Extend the program so that an extension can be optionally specified on any number. All of these should work:

```
621-6613 x413

520-621-6613 x413

(520)621-6613 x 27

520-555-1212

621-6613x13423
```

Co-expression basics

Icon's *co-expression* type allows an expression, usually a generator, to be "captured" so that results may be produced as needed.

A co-expression is created using the `create` control structure:

```
create expr
```

Example:

```
][ c := create 1 to 3;
   r := co-expression_2(0)  (co-expression)
```

A co-expression is *activated* with the unary `@` operator.

When a co-expression is activated the captured expression is evaluated until a result is produced. The co-expression then becomes dormant until activated again.

```
][ x := @c;
   r := 1  (integer)

][ y := @c;
   r := 2  (integer)

][ z := x + y + @c;
   r := 6  (integer)

][ @c;
Failure
```

Activation fails when the captured expression has produced all its results.

Co-expression basics, continued

Activation is not generative. At most one result is produced by activation:

```
][ vowels := create !"aeiou";
   r := co-expression_6(0)  (co-expression)

][ every write(@vowels);
a
Failure
```

Another example:

```
][ s := "It is Hashtable or HashTable?";
   r := "It is Hashtable or HashTable?"

][ caps := create !s == !&ucase;
   r := co-expression_3(0)  (co-expression)

][ @caps;
   r := "I"  (string)

][ cc := @caps || @caps;
   r := "HH"  (string)

][ [@caps];
   r := ["T"]  (list)

][ [@caps];
Failure
```

Co-expression basics, continued

Co-expressions can be used to perform generative computations in parallel:

```
][ upper := create !&ucase;
   r := co-expression_4(0)  (co-expression)

][ lower := create !&lcase;
   r := co-expression_5(0)  (co-expression)

][ while write(@upper, @lower);
Aa
Bb
Cc
Dd
...
```

Here is a code fragment that checks the first 1000 elements of a binary number generator:

```
bvalue := create binary() # starts at "1"

every i := 1 to 1000 do
    if integer("2r"||@bvalue) ~= i then
        stop("Mismatch at ", i)
```

Co-expression basics, continued

The "size" of a co-expression is the number of results it has produced.

```
][ words := create !split("just a test");
   r := co-expression_5(0)  (co-expression)

][ while write(@words);
just
a
test
Failure

][ *words;
   r := 3  (integer)

][ *create 1 to 10;
   r := 0  (integer)
```

Problem: Using a co-expression, write a program to produce a line-numbered listing of lines from standard input.

Example: `vcycle`

This program uses co-expressions to conveniently cycle through the elements in a list:

```
procedure main()
    vtab := table()

    while writes("A or Q: ") & line := read() do {
        parts := split(line,'=')

        if *parts = 2 then {
            vname := parts[1]
            values := parts[2]

            vtab[vname] :=
                create |!split(values, ',')
            }
        else
            write(@vtab[line])
        }
end
```

Interaction:

```
% vcycle
A or Q: color=red,green,blue
A or Q: yn=yes,no
A or Q: color
red
A or Q: color
green
A or Q: yn
yes
A or Q: color
blue
A or Q: color
red
```

Problem: Get rid of those integer subscripts!

"Refreshing" a co-expression

A co-expression can be "refreshed" with the unary ^ (caret) operator:

```
][ lets := create !&letters;
   r := co-expression_4(0)   (co-expression)

][ @lets;
   r := "A"  (string)

][ @lets;
   r := "B"  (string)

][ rlets := ^lets;
   r := co-expression_5(0)   (co-expression)

][ *rlets;
   r := 0  (integer)

][ @lets;
   r := "C"  (string)

][ @rlets;
   r := "A"  (string)
```

In fact, the "refresh" operation produces a new co-expression with the same initial conditions as the operand.

"refresh" better describes this operation:

```
][ lets := ^lets;
   r := co-expression_6(0)   (co-expression)

][ @lets;
   r := "A"  (string)
```

Co-expressions and variables

The environment of a co-expression includes a copy of all the non-static local variables in the enclosing procedure.

```
][ low := 1;

][ high := 10;

][ c1 := create low to high;

][ low := 5;

][ c2 := create low to high;

][ @c1;
   r := 1  (integer)

][ @c2;
   r := 5  (integer)

][ @c2;
   r := 6  (integer)
```

Refreshing a co-expression restores the value of locals at the time of creation for the co-expression:

```
][ low := 10;
][ c1 := ^c1;

][ c2 := ^c2;

][ @c1;
   r := 1  (integer)

][ @c2;
   r := 5  (integer)
```

Co-expressions and variables, continued

Because structure types such as lists use reference semantics, using a local variable with a list value leads to "interesting" results:

```
][ L := [];
   r := []  (list)

][ c1 := create put(L, 1 to 10) & L;
   r := co-expression_8(0)  (co-expression)

][ c2 := create put(L, !&lcase) & L;
   r := co-expression_9(0)  (co-expression)

][ @c1;
   r := [1]  (list)

][ @c1;
   r := [1,2]  (list)

][ @c2;
   r := [1,2,"a"]  (list)

][ @c1;
   r := [1,2,"a",3]  (list)
```

Procedures that operate on co-expressions

Here is a procedure that returns the length of a co-expression's result sequence:

```
procedure Len(C)
    while @C
    return *C
end
```

Usage:

```
][ Len(create 1 to 10);
   r := 10  (integer)

][ Len(create !&cset);
   r := 256  (integer)
```

Problem: Write a routine `Results(C)` that returns the result sequence of the co-expression C:

```
][ Results(create 1 to 5);
   r := [1,2,3,4,5]  (list)
```

PDCOs

By convention, routines like `Len` and `Results` are called *programmer defined control operations*, or PDCOs.

Icon provides direct support for PDCOs with a convenient way to pass a list of co-expressions to a procedure:

```
proc{expr1, expr2, ..., exprN}  # Note: curly braces!
```

This is a shorthand for:

```
proc([create expr1, ..., create exprN])
```

Revised usage of `Len` and `Results`:

```
][ Len{!&lcase};
   r := 26  (integer)

][ Results{1 to 5};
   r := [1,2,3,4,5]  (list)
```

Revised version of `Len`:

```
procedure Len(L)
    C := L[1]

    while @C
    return *C
end
```

PDCOs, continued

Imagine a PDCO named `Reduce` that "reduces" a result sequence by interspersing a binary operation between values:

```
][ Reduce{"+", 1 to 10};
   r := 55  (integer)

][ Reduce{"*", 1 to 25};
   r := 15511210043330985984000000  (integer)

][ Reduce{"||", !&lcase};
   r := "abcdefghijklmnopqrstuvwxyz"  (string)
```

Implementation:

```
procedure Reduce(L)
    op := @L[1]

    result := @L[2] | fail

    while result := op(result,@L[2])

    return result
end
```

PDCOs, continued

Problem: Write a PDCO that interleaves result sequences:

```
][ .every Interleave{1 to 3, !&lcase,
                     ![10,20,30,40]};
   1  (integer)
   "a"  (string)
   10  (integer)
   2  (integer)
   "b"  (string)
   20  (integer)
   3  (integer)
   "c"  (string)
   30  (integer)
```

`Interleave` should fail upon the first occurrence of an argument expression failing.

Modeling control structures

Most of Icon's control structures can be modeled with a PDCO. Example:

```
procedure Every(L)
   while @L[1] do @^L[2]
end
```

A simple test: (Note that `i`and `c` are globals.)

```
global i,c
procedure main()

    Every{i := 1 to 5, write(i)}

    Every{i := ![10, 20, 30],
        Every{c := !"abc", write(i, " ", c)}}
end
```

Output:

```
1
2
3
4
5
10 a
10 b
10 c
20 a
20 b
20 c
30 a
30 b
30 c
```

Modeling control structures, continued

Here is a model for limitation from `pdco.icn` in the Icon Procedure Library:

```
procedure Limit(L)
   local i, x

   while i := @L[2] do {
      every 1 to i do
         if x := @L[1] then suspend x
         else break
      L[1] := ^L[1]
      }
end
```

Usage:

```
][ .every Limit{!"abc", 1 to 3};
   "a"  (string)
   "a"  (string)
   "b"  (string)
   "a"  (string)
   "b"  (string)
   "c"  (string)

][ .every !"abc" \ (1 to 3);
   "a"  (string)
   "a"  (string)
   "b"  (string)
   "a"  (string)
   "b"  (string)
   "c"  (string)
```

Modeling control structures, continued

Problem: Model the `if` and `while` control structures. Here's a test program:

```
global line, sum
procedure main()
    sum := 0

    While{line := read(),
        If{numeric(line), sum +:= line}}

    write("Sum: ", sum)
end
```

Here are the bounding rules:

```
while expr1 do expr2
if expr1 then expr2
```

Restriction: You can't use a control structure in its own model.