Co-Expressions

Normally, generators are completely at the mercy of their surrounding expression.

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:

```plaintext
create expr
```

For example:

```plaintext
c := create 1 to 3
```
Co-Expression Activation

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.
Co-Expression Activation

- C := create 1 to 3
- write(@C)
  1
- X := @C
- write(X)
  2
- Y := X + @C
- write(Y)
  5

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

• Activation is not a generator
• @C gets one result, or fails
• X := create !"aeiou"
• every write(@X)
  a
• while write(@X)
  e
  i
  o
  u
Bigger Example

s := "It is Hashtable or HashTable?";
caps := create !s == !&ucase;
@caps;
  (produces “I”)
cc := @caps || @caps
  (produces “HH”)
[@caps]
  (produces [“T”])
[@caps]
  (fails)
Interleaved Generators

upper := create !&ucase;
lower := create !&lcase;
while write(@upper, @lower)
    Aa
    Bb
    Cc
    Dd
    ...

Exercise

Write a generator binary() that generates an infinite sequence of binary strings (so we can test co-expressions with it). The sequence is

“0”
“1”
“10”
“11”
“100”
“101”
...

Checking binary()

bvalue := create binary()
every i := 0 to 1000 do
  if integer("2r"||@bvalue) ~= i then
    stop("Mismatch at ", i)
Co-expression size

The size of a co-expression \(*C*\) is the number of results it has produced so far.

```plaintext
C := create find(“the”, “the quick brown fox _ jumped over the lazy yellow dog”) while write(@C) 1 33 write(*C) 2
```
Exercise

Write a procedure split(s, c) that turns a string s into a list by breaking up the string into elements separated by character(s) in cset c. (So we can use it to study co-expressions.)
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
Vcycle – example run

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
Refreshing a co-expression

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

lets := create !&letters
@lets (produces “A”)
@lets (produces “B”)
rlets := ^lets (new copy/restart)
*rlets (produces 0)
@lets (produces “C”)
@rlets (produces “A”)
Co-expression Environments

A co-expression has its own independent evaluation stack, and a saved copy of the local variables of the enclosing procedure at the time it was created.

low := 1
high := 10
c1 := create low to high
low := 5
low := 5

@c1 produces 1
@c2 produces 5
@c2 produces 6
Refreshing Environments

Refresh operator restores locals from saved copy of the local variables of the enclosing procedure at the time it was created.

low := 10
c1 := ^c1
c2 := ^c2
@c1 produces 1
@c2 produces 5
Stack Refresh Does not Save Heap

Because structure types such as lists use reference semantics, using a co-expr with a (reference to a) list leads to "interesting" results:

L := []
c1 := create put(L, 1 to 10) & L
c2 := create put(L, !&lcase) & L
@c1 ====> produces [1]
@c1 ====> produces [1,2]
@c2 ====> produces [1,2,"a"]
@c1 ====> produces [1,2,"a",3]
Procedures can use co-expressions

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

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

Exercise: write a procedure Results(C) that returns a list containing the result sequence for co-expression C.
Threads

- A co-expression is a synchronous thread
- `thread(C)` turns C loose to run in parallel
- `wait(C)` waits for C to finish
- A few more functions (mutex, condition variables, locks)
- New feature
  - Ported to all major platforms
  - Not all Unicon VM's are built with threads
  - A few unfinished bits, such as @
Hello Threads

procedure main()
    write("Create two threads..")
    a := create print(1)
    b := create print(2)
    write("Fire the two threads and wait for them...")
    thread(a)
    thread(b)
    wait(a)
    wait(b)
    write("The process ended successfully, exiting now..")
end

procedure print(id)
    write("Hello world! I'm thread ", id, " I will finish now." )
end