Generator basics

In most languages, evaluation of an expression always produces one result. In Icon, an expression can produce zero, one, or many results.

Consider the following program. The procedure `Gen` is said to be a *generator*.

```
procedure Gen()
    write("Gen: Starting up...")
    suspend 3

    write("Gen: More computing...")
    suspend 7

    write("Gen: Still computing...")
    suspend 13

    write("Gen: Out of gas...")
    fail  # not really needed
end

procedure main()
    every i := Gen() do
        write("Result = ", i)
end
```

Execution:

```
Gen: Starting up...
Result = 3
Gen: More computing...
Result = 7
Gen: Still computing...
Result = 13
Gen: Out of gas...
```

Generator basics, continued

The `suspend` control structure is like `return`, but the procedure remains active with all state intact and ready to continue execution if it is *resumed*.

Program output with call tracing active:

```
                :    main()
gen.icn :    2  | Gen()
Gen: Starting up...
gen.icn :    8  | Gen suspended 3
Result = 3
gen.icn :    3  | Gen resumed
Gen: More computing...
gen.icn :   10  | Gen suspended 7
Result = 7
gen.icn :    3  | Gen resumed
Gen: Still computing...
gen.icn :   12  | Gen suspended 13
Result = 13
gen.icn :    3  | Gen resumed
Gen: Out of gas...
gen.icn :   14  | Gen failed
gen.icn :    4  main failed
```

Generator basics, continued

Recall the `every` loop:

```
    every i := Gen() do
        write("Result = ", i)
```

`every` is a control structure that looks similar to `while`, but its behavior is very different.

`every` evaluates the control expression and if a result is produced, the body of the loop is executed. Then, the control expression is resumed and if another result is produced, the loop body is executed again. This continues until the control expression fails.

Anthropomorphically speaking, `every` is never satisfied with the result of the control expression.

Generator basics, continued

For reference:

```
every i := Gen() do
    write("Result = ", i)
```

It is said that `every` drives a generator to failure.

Here is another way to drive a generator to failure:

```
write("Result = " || Gen()) & 1 = 0
```

Output:

```
Gen: Starting up...
Result = 3
Gen: More computing...
Result = 7
Gen: Still computing...
Result = 13
Gen: Out of gas...
```

Note: The preferred way to cause failure in an expression is to use the `&fail` keyword. Evaluation of `&fail` always fails:

```
][ &fail;
Failure
```

Generator basics, continued

If a failure occurs during evaluation of an expression, Icon will resume a suspended generator in hopes that another result will lead to success of the expression.

A different `main` program to exercise `Gen`:

```
procedure main()
    while n := integer(read()) do {
        if n = Gen() then
            write("Found ", n)
        else
            write(n, " not found")
        }
end
```

Interaction:

```
3
Gen: Starting up...
Found 3
10
Gen: Starting up...
Gen: More computing...
Gen: Still computing...
Gen: Out of gas...
10 not found
13
Gen: Starting up...
Gen: More computing...
Gen: Still computing...
Found 13
```

This is an example of *goal directed evaluation* (GDE).

Generator basics, continued

A generator can be used in <u>any context</u> that an ordinary expression can be used in:

```
][ write(Gen());
Gen: Starting up...
3
   r := 3  (integer)

][ Gen() + 10;
Gen: Starting up...
   r := 13  (integer)

][ repl("abc", Gen());
Gen: Starting up...
   r := "abcabcabc"  (string)
```

There is no direct way to whether a procedure's result was produced by `return` or `suspend`.

This version of `double` works just fine:

```
procedure double(n)
    suspend 2 * n
end
```

Usage:

```
][ double(double(10));
   r2 := 40  (integer)
```

The generator `to`

Icon has many built-in generators. One is the `to` operator, which generates a sequence of integers. Examples:

```
][ every i := 3 to 7 do
...     write(i);
3
4
5
6
7
Failure

][ every i := -10 to 10 by 7 do
...     write(i);
-10
-3
4
Failure

][ every write(10 to 1 by -3);
10
7
4
1
Failure

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

][ 8 < (1 to 10);
   r := 9  (integer)

][ every write(8 < (1 to 10));
9
10
Failure
```

The generator `to`, continued

Problem: Without using `every`, write an expression that prints the odd integers from 1 to 100.

Problem: Write an Icon procedure `ints(first, last)` that behaves like to-by with an assumed "by" of 1.

Backtracking and bounded expressions

Another way to print the odd integers between 1 and 100:

```
i := 1 to 100 & i % 2 = 1 & write(i) & &fail
```

This expression exhibits *control backtracking*—the flow of control sometimes moves backwards.

In some cases backtracking is desirable and in some cases it is not.

Expressions appearing as certain elements of control structures are *bounded*. A bounded expression can produce at most one result, thus limiting backtracking.

One example: Each expression in a compound expression is bounded.

Contrast:

```
][ i := 1 to 3 & write(i) & &fail;
1
2
3
Failure

][ { i := 1 to 3; write(i); &fail };
1
Failure
```

Bounded expressions, continued

The mechanism of expression bounding is this: if a bounded expression produces a result, generators in the expression are discarded.

In `while` <u>*`expr1`*</u> `do` <u>*`expr2`*</u>, both expressions are bounded.

In `every` *`expr1`* `do` <u>*`expr2`*</u>, only *`expr2`* is bounded.

Consider

```
    every i := 1 to 10 do write(i)
```

and

```
    while i := 1 to 10 do write(i)
```

The latter is an infinite loop!

In an if-then-else, only the control expression is bounded:

    if <u>*expr1*</u> then *expr2* else *expr3*

See page 91 in the text for the full list of bounded expressions.

Bounded expressions, continued

Here is a generator that simply prints when it is suspended and resumed:

```
procedure sgen(n)
    write(n, " suspending")
    suspend
    write(n, " resumed")
end
```

Notice the behavior of `sgen` with `every`:

```
][ every sgen(1) do sgen(2);
1 suspending
2 suspending
1 resumed
Failure
```

Note that there is no way for a generator to detect that it is being discarded.

Here is `sgen` with `if-then-else`:

```
][ (if sgen(1) then sgen(2) else sgen(3)) & &fail;
1 suspending
2 suspending
2 resumed
Failure

][ (if \sgen(1) then sgen(2) else sgen(3)) & &fail;
1 suspending
1 resumed
3 suspending
3 resumed
Failure
```

What would `while sgen(1) do sgen(2)` output?

The generator "bang" (!)

Another built-in generator is the unary exclamation mark, called "bang".

It is polymorphic, as is the size operator (*). For character strings it generates the characters in the string one at a time.

```
][ every c := !"abc" do
...     write(c);
a
b
c
Failure

][ every write(!"abc");
a
b
c
Failure

][ every write(!"");
Failure
```

A program to count vowels appearing on standard input:

```
procedure main()
    vowels := 0
    while line := read() do {
        every c := !line do
            if c == !"aeiouAEIOU" then
                vowels +:= 1
        }

    write(vowels, " vowels")
end
```

The generator "bang" (`!`), continued

If applied to a value of type `file`, `!` generates the lines remaining in the file.

The keyword `&input` represents the file associated with standard input.

A simple line counter (`lcount.icn`):

```
procedure main()
    lines := 0

    every !&input do
        lines +:= 1

    write(lines, " lines")
end
```

Usage:

```
% lcount < lcount.icn
8 lines
% lcount < /dev/null
0 lines
% lcount < /etc/passwd
1620 lines
```

Problem: Change the vowel counter to use generation of lines from `&input`?

The generator "bang" (!), continued

The line counter extended to count characters, too:

```
procedure main()
    chars := lines := 0

    every chars +:= *!&input + 1 do
        lines +:= 1

    write(lines, " lines, ", chars,
        " characters")
end
```

If ! is applied to an integer or a real, the value is first converted to a string and then characters are generated:

```
][ .every !1000;
   "1"  (string)
   "0"  (string)
   "0"  (string)
   "0"  (string)

][ .every !&pi;
   "3"  (string)
   "."  (string)
   "1"  (string)
   "4"  (string)
   "1"  (string)
   "5"  (string)
   "9"  (string)
   "2"  (string)
   "6"  (string)
   ...
```

Note that `.every` is an `ie` directive that drives a generator to exhaustion, showing each result.

Multiple generators

An expression may contain any number of generators:

```
][ every write(!"ab", !"+-", !"cd");
a+c
a+d
a-c
a-d
b+c
b+d
b-c
b-d
Failure
```

Generators are resumed in a LIFO manner: the generator that most recently produced a result is the first one resumed.

Another example:

```
][ x := 1 to 10 & y := 1 to 10 & z := 1 to 10 &
    x*y*z = 120 & write(x, " ", y, " ", z);
2 6 10
```

Problem: What are the result sequences of the following expressions?

```
(0 to 20 by 2) = (0 to 20 by 3)


1 to !"1234"


(1 to 3) to (5 to 10)
```

Multiple generators, continued

Problem: Write an expression that succeeds if strings `s1` and `s2` have any characters in common.

Problem: Write a program to read standard input and print all the vowels, one per line.

Multiple generators, continued

A program to show the distribution of the sum of three dice:

```
procedure main()
  every N := 1 to 18 do {
     writes(right(N,2), " ")
     every (1 to 6) + (1 to 6) + (1 to 6) = N do
         writes("*")
     write()
     }
end
```

Output:

```
  1
  2
  3 *
  4 ***
  5 ******
  6 **********
  7 ***************
  8 *********************
  9 *************************
 10 ***************************
 11 ***************************
 12 *************************
 13 *********************
 14 ***************
 15 **********
 16 ******
 17 ***
 18 *
```

Problem: Generalize the program to any number of dice.

Alternation

The alternation control structure looks like an operator:

```
    expr1 | expr2
```

This creates a generator whose result sequence is the
the result sequence of `expr1` followed by the result sequence of `expr2`.

For example, the expression

```
    3 | 7
```

has the result sequence {`3`, `7`}.

The procedure `Gen` is in essence equivalent to the expression:

```
    3 | 7 | 13
```

The expression

```
    (1 to 5) | (5 to 1 by -1)
```

has the result sequence {`1`, `2`, `3`, `4`, `5`, `5`, `4`, `3`, `2`, `1`}.

What are the result sequences of these expressions?

```
    (1 < 0) | (0 = 1)

    (1 < 0) | (0 ~= 1)

    Gen() | (Gen() > 10) | (Gen() + 1)
```

Alternation, continued

A result sequence may contain values of many types:

```
][ every write(1 | 2 | !"ab" | real(Gen()));
1
2
a
b
Gen: Starting up...
3.0
Gen: More computing...
7.0
Gen: Still computing...
13.0
Gen: Out of gas...
Failure
```

Alternation used in goal-directed evaluation:

```
procedure main()
  while time := (writes("Time? ") & read()) do {
     if time = (10 | 2 | 4) then
         write("It's Dr. Pepper time!")
     }
end
```

Interaction:

```
% dptime
Time? 1
Time? 2
It's Dr. Pepper time!
Time? 3
Time? 4
It's Dr. Pepper time!
```

Alternation, continued

A program to read lines from standard input and write out the first twenty characters of each line:

```
procedure main()
    while line := read() do
        write(line[1:21])
end
```

Program output when provided the program itself as input:

```
    while line := re
        write(line[1
```

What happened?

Solution:

```
procedure main()
    while line := read() do
        write(line[1:(21|0)])
end
```

Output:

```
procedure main()
    while line := re
        write(line[1
end
```

What does this expression do?

```
write((3 | 7 | 13) > 10)
```

Repeated alternation

An infinite result sequence can be produced with *repeated alternation*,

```
| expr
```

which repeatedly generates the result sequence of `expr`.

The expression `|1` has this result sequence:

```
{1, 1, 1, ...}
```

The expression `|!"abc"` has this result sequence:

```
{"a", "b", "c", "a", "b", "c", "a", ...}
```

What are the result sequences of the following expressions?

```
|1 = 2

9 <= |(1 to 10)
```

Limitation

The limitation construct can be used to restrict a generator to a maximum number of results.

General form:

```
    expr1 \ expr2
```

One way to see if an "e" appears in the first twenty characters of a string:

```
    "e" == !s[1:20]
```

Another way:

```
    "e" == !s\20
```

Which is better?

A common use of limitation is to restrict a computation to the first result of a generator:

```
    if f(Gen()\1) = n then ...
```

Problem: Using limitation create an expression whose result sequence is {a, a, b, a, b, c, a, b, c, d} (all strings).

More on `suspend`

If `suspend`'s expression is a generator, each result is suspended in turn.

Example:

```
procedure updown(N)
    suspend 1 to N-1
    suspend N to 1 by -1
end
```

Usage:

```
][ every write(updown(3));
1
2
3
2
1
Failure
```

The full form of `suspend` is similar to `every`:

```
suspend expr1 do
    expr2
```

If *`expr1`* yields a result, the value is suspended. When the procedure is resumed, *`expr2`* is evaluated, and the process repeats until *`expr1`* fails.