Random value selection

The polymorphic unary `?` operator is used to produce random values.

If applied to an integer N > 0, an integer between 1 and N inclusive is produced:

```
][ ?10;
   r := 3  (integer)

][ ?10;
   r := 5  (integer)

][ ?10;
   r := 4  (integer)
```

Problem: Write a procedure `ab()` that, on average, returns "a" 25% of the time and "b" 75% of the time.

The same random sequence is produced every run by default, but the "generator" can be seeded by assigning a value to `&random`. A simple seeder:

```
][ &clock;
   r := "17:10:46"  (string)

][ &random := &clock[-2:0];
   r := 25  (integer)
```

Random value selection, continued

If `?` is applied to a string, a random character from the string is produced:

```
][ ?"random";
   r := "n"  (string)

][ ?"random";
   r := "m"  (string)
```

Applying `?` to a list produces a random element:

```
][ ?[10,0,"thirty"];
   r := 10  (integer)

][ ?[10,0,"thirty"];
   r := "thirty"  (string)

][ ??[10,0,"thirty"];
   r := 0.6518579154  (real)
```

If `?` is applied to zero a real number in the range 0.0 to 1.0 is produced:

```
][ ?0;
   r := 0.05072018769  (real)

][ ?0;
   r := 0.716947168  (real)
```

Problem: Write the procedure `ab()` in another way.

Random value selection, continued

When applied to strings and lists, the result of `?` is a variable, and can be assigned to. Example:

```
procedure main()
    line := "Often wrong; never unsure!"
    every 1 to 10 do {
        ?line :=: ?line
        write(line)
        }
end
```

Output:

```
Oftengwron ; never unsure!
Oftengwrnn ; oever unsure!
Oftengw nnr; oever unsure!
Ofuengw nnr; oever tnsure!
O uengw nnr; oeverftnsure!
O unngw enr; oeverftnsure!
O unngw enr; eevorftnsure!
O unngw enr; efvoretnsure!
O unngt enr; efvorewnsure!
O unngt unr; efvorewnsere!
```

Problem: Write a procedure `mutate(s,n)` that does `n` random swaps of the "words" in the string `s`.

Random value selection, continued

Problem: Write a program that generates test data for a program that finds the longest line(s) in a file.

Variable length argument lists

In some cases it is useful for a procedure to handle any number of arguments.

Here is a procedure that calculates the sum of its arguments:

```
procedure sum(nums[])
    total := 0

    every total +:= !nums
    return total
end
```

Usage:

```
][ sum(5,8,10);
   r := 23  (integer)

][ sum();
   r := 0  (integer)

][ sum(1,2,3,4,5,6,7);
   r := 28  (integer)
```

Variable length argument lists, continued

One or more parameters may precede a final parameter designated to collect additional arguments.

Consider a very simplistic C-like `printf`:

```
][ printf("e = %, pi = %\n", &e, &pi);
e = 2.718281828459045, pi = 3.141592653589793
```

Implementation:

```
procedure printf(format, vals[])
    i := 0
    every e := !split(format, "%", 1) do
        if e == "%" then
            writes(vals[i+:=1])
        else
            writes(e)
    return
end
```

Procedures as values

Icon has a *procedure* type. Names of built-in functions such as `write` and Icon procedures such as `double` are simply variables whose value is a procedure.

Suppose you'd rather use "`println`" than "`write`":

```
global println
procedure main()
    println := write
    ...
end

procedure f()
    println("in f()...")
end
```

Consider this program:

```
procedure main()
    write :=: read
    while line := write() do
        read(line)
end
```

Procedures as values, continued

A procedure may be passed as an argument to a procedure.

Here is a procedure that calls the procedure `p` with each element of `L` in turn, forming a list of the results:

```
procedure map(p, L)
    result := []
    every e := !L do
        put(result, p(e) | &null)
    return result
end
```

Usage: (with `double` from slide 42)

```
][ vals := [1, "two", 3];
   r := L1:[1,"two",3]  (list)

][ map(double, vals);
   r := L1:[2,"twotwo",6]  (list)
```

A computation may yield a procedure:

```
f()(a, b)

x := (p1 | p2 | p3)(7,11)

point: = (?[up, down])(x,y)
```

String invocation

It is possible to "invoke" a string:

```
][ "+"(3,4);
   r := 7  (integer)

][ "*"(&lcase);
   r := 26  (integer)

][ (?"+*")(12,3);
   r := 15  (integer)
```

Consider a simple evaluator:

```
Expr? 3 + 9
12
Expr? 5 ^ 10
9765625

Expr? abc repl 5
abcabcabcabcabc

Expr? xyz... trim .
xyz
```

Implementation:

```
invocable all
procedure main()
    while writes("Expr? ") &
      e := split(read()) do
        write(e[2](e[1],e[3]))
end
```

String invocation, continued

Some details on string invocation:

- Operators with unary and binary forms are distinguished by the number of arguments supplied:

```
][ star := "*";
   r := "*"  (string)

][ star(4);
   r := 1  (integer)

][ star(4,7);
   r := 28  (integer)
```

- User defined procedures can be called.

- The "`invocable all`" prevents unreferenced procedures from being discarded.

- `proc()` and `args()` are sometimes useful when using string invocation.

Mutual evaluation

One way to evaluate a series of expressions and, if all succeed, produce the value of the final expression is this:

```
expr1 & expr2 & ... & exprN
```

The same computation can be expressed with *mutual evaluation:*

```
(expr1, expr2, ..., exprN)
```

If a value other than the result of the last expression is desired, an expression number can be specified:

```
][ 3(10,20,30,40);
   r := 30  (integer)

][ .every 1(x := 1 to 10, x * 3 < 10);
   1  (integer)
   2  (integer)
   3  (integer)
```

The expression number can be negative:

```
.every (-2)(x := 1 to 10, x * 3 < 10);
```

Now you can understand error 106:

```
][ bogus();
Run-time error 106
procedure or integer expected
offending value: &null
```

Mutual evaluation, continued

One use of mutual evaluation is to "no-op" a routine.

Consider this:

```
global debug
procedure main()
    ...
    debug := write
    ...
end

procedure f(x)
    debug("In f(), x = ", x)
    ...
end
```

To turn off debugging output:

```
debug := 1
```

File I/O

Icon has a `file` type and three built-in files: `&input`, `&output`, and `&errout`. These are associated with the standard input, standard output, and error output streams.

By default:

- `read()` reads from `&input`
- `write()` and `writes()` output to `&output`
- `stop()` writes to `&errout`

The `open(name,mode)` function opens the named file for input and/or output (according to `mode`) and returns a value of type `file`. Example:

```
wfile := open("dictionary.txt", "r")
```

A `file` can be specified as the argument for `read`:

```
line := read(wfile)
```

A file can be specified as an argument to `write`:

```
logfile := open("log."||getdate(), "w")
write(logfile, "Log created at ", &dateline)
```

It is seldom used but any number of arguments to `write` can be files:

```
write("abc", logfile, "xyz", &output, "pdq")
```

This results in `"abcpdq"` being written to standard output, and `"xyz"` being written to `logfile`.

File I/O, continued

A very simple version of the `cp` command:

```
procedure main(a)
    in := open(a[1]) |
        stop(a[1], ": can't open for input")

    out := open(a[2], "w") |
        stop(a[2], ": can't open for output")

    while line := read(in) do
        write(out, line)
end
```

Usage:

```
% cp0 /etc/motd x
% cp0 /etc/motdxyz x
/etc/motdxyz: can't open for input
% cp0 x /etc/passwd
/etc/passwd: can't open for output
```

Common bug: Opening a file but forgetting to pass it to `read()`.

File I/O, continued

The `read()` function is designed for use with line by line input and handles OS-specific end-of-line issues.

The `reads(f,n)` function is designed for reading binary data. It reads `n` bytes from the file `f` and returns a string.

Here is a program that reads files named on the command line and prints out the number of bytes and null bytes (zero bytes) in the file:

```
procedure main(a)
    every fname := !a do {
        f := open(fname, "ru")
        bytes := nulls := 0
        while buf := reads(f, 1024) do {
            bytes +:= *buf
            every !buf == "\x00" do
                nulls +:= 1
            }

        write(fname, ": ", bytes, " bytes, ",
            nulls, " nulls")
        }
end
```

Usage:

```
% countnulls countnulls.icn countnulls
countnulls.icn: 289 bytes, 0 nulls
countnulls: 1302 bytes, 620 nulls
```

Other built-in functions related to files include `rename`, `remove`, `seek`, and `where`.

I/O with pipes

If the open mode includes "p", the name is considered to be a command, which is started, and a pipe is opened to the process.

Here is a program that reads the output of the who command and reports the number of users:

```
procedure main()
    who_data := open("who", "rp")

    num_users := 0
    while read(who_data) & num_users +:= 1

    write(num_users, " users logged in")
end
```

Usage:

```
% nusers
73 users logged in
```

I/O with pipes, continued

Here is a program that opens a pipe to the `ed` text editor and sends it a series of commands to delete lines from a file:

```
procedure main(a)
   ed := open("ed "||a[1]||" >/dev/null", "wp")|
       stop("oops!?")

   every num := !a[2:0] do
       write(ed, num, "d")

   write(ed, "w")
   write(ed, "q")
end
```

Usage:

```
% cat five
1
2
3
4
5
% dellines five 2 4
% cat five
1
3
4
%
```

Unfortunately, bi-directional pipes are not supported.