Explicit conversions, continued

A code fragment to repeatedly prompt until a numeric value is input:

```
value := &null   # not really needed...

while /value do {
    writes("Value? ")
    value := numeric(read())
    }

write("Value is ", value)
```

Interaction:

```
Value? x
Value?
Value? 10
Value is 10
```

The `repeat` expression

An infinite loop can be produced with `while 1 do ...` but the `repeat` expression is the preferred way to indicate endless repetition.

The general form:

```
repeat expr
```

Example:

```
repeat write(1)
```

Another way to copy lines from standard input to standard output:

```
repeat {
    if not (line := read()) then
        break
    write(line)
    }
```

The `until-do` expression

General form:

```
until expr1 do
    expr2
```

`until-do` is essentially a `while-do`, but with an inverted test, terminating when the test succeeds.

This loop prints lines until a line containing only "start" is encountered:

```
until (line := read()) == "start" do
    write(line)
```

The `do` clause can be omitted:

```
until read() == "end"
```

Procedure basics

All executable code in an Icon program is contained in procedures. A procedure may take arguments and it may return a value of interest.

Execution begins by calling the procedure `main`.

A simple program with two procedures:

```
procedure main()
    while n := read() do
        write(n, " doubled is ", double(n))
end

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

Sidebar: Compilation

If `double.icn` contains the code on the previous slide, it can be compiled and linked into an *icode* executable named `double` with the `icont` command:

```
% icont double.icn
Translating:
double.icn:
  main
  double
No errors
Linking:
% ls -l double
-rwxrwxr-x   1 whm dept 969 Jan 19 15:50 double
% double
7
7 doubled is 14
15
15 doubled is 30
^D (control-D)
%
```

The source file name can be followed with `-x` to cause execution to immediately follow compilation:

```
% icont double.icn -x
Translating:
double.icn:
  main
  double
No errors
Linking:
Executing:
100
100 doubled is 200
```

Procedure basics, continued

A procedure may produce a result or it may fail.

Here is a more flexible version of `double`:

```
procedure double(x)
    if type(x) == "string" then
        return x || x
    else if numeric(x) then
        return 2 * x
    else
        fail
end
```

Usage:

```
][ double(5);
   r := 10  (integer)

][ double("xyz");
   r := "xyzxyz"  (string)

][ double(&null);
Failure

][ double(double);
Failure
```

Procedure basics, continued

If no value is specified in a `return` expression, the null value is returned.

```
procedure f()
    return
end
```

Usage:

```
][ f();
   r := &null  (null)
```

If the flow of control reaches the end of a procedure without returning, the procedure fails.

```
procedure hello()
    write("Hello!")
end
```

Usage:

```
][ hello();
Hello!
Failure
```

Procedure basics, continued

Explain the operation of this code:

```
procedure main()
    while writelong(read(), 10)
end

procedure writelong(s,n)
    if *s > n then
        write(s)
end
```

Procedures—omitted arguments

If any arguments for a procedure are not specified, the value of the corresponding parameter is null.

```
procedure wrap(s, w)
    /w := "()"  # if w is null, set w to "()"
    return w[1] || s || w[2]
end

][ wrap("x", "[]");
   r := "[x]"  (string)

][ wrap("x");
   r := "(x)"  (string)
```

Any or all arguments can be omitted:

```
procedure wrap(s, w)
    /s := ""
    /w := "()"
    return w[1] || s || w[2]
end

][ wrap("x");
   r := "(x)"  (string)

][ wrap(,"{}");
   r := "{}"  (string)

][ wrap(,);
   r := "()"  (string)

][ wrap();
   r := "()"  (string)
```

Arguments in excess of the formal parameters are simply ignored.

Omitted arguments, continued

Many built-in functions have default values for omitted arguments.

```
][ right(35, 10, ".");
   r1 := "........35"  (string)

][ right(35, 10);
   r2 :=  "        35"  (string)

][ trim("just a test   ");
   r3 := "just a test"  (string)

][ reverse(trim(reverse(r1), "."));
   r4 := "35"  (string)
```

Scope rules

In Icon, variables have either *global scope* or *local scope*.

Global variables are accessible inside every procedure in a program.

Global variables are declared with a `global` declaration:

```
global x, y
global z
procedure main()
    x := 1
    z := "zzz..."
    f()
    write("x is ", x)
end

procedure f()
    x := 2
    write(z)
end
```

Output:

```
zzz...
x is 2
```

This is no provision for initializing global variables in the `global` declaration.

Global declarations must be declared outside of procedures.

The declaration of a global does not need to precede its first use.

Scope rules, continued

The `local` declaration is used to explicitly indicate that a variable has local scope.

```
procedure x()
    local a, b

    a := g()
    b := h(a)
    f(a, b)
end
```

Local variables are accessible only inside the procedure in which they are defined (explicitly or implicitly).

Any data referenced by a local variable is free to be reclaimed when the procedure returns.

If present, local declarations must come first in a procedure.

Scope rules—a hazard

Undeclared variables default to local unless they are elsewhere defined as global. This creates a hazard:

```
here.icn:

    procedure x()
        a := g()
        b := h(a)
        f(a, b)
    end

elsewhere.icn:

    global a, b
    ...
```

A call to `x` will cause the global variables `a` and `b` to be modified.

Names of built-in functions and Icon procedures are global variables. Inadvertently using a routine name as an undeclared local variable will clobber the routine.

```
procedure f(s)
    pos := get_position(s, ...)
    ...
end
```

Unfortunately, there is a built-in function named `pos`!

Rule of thumb: Always declare local variables. (Use `icont`'s `-u` flag to find undeclared variables.)

`static` variables

The `static` declaration is used to indicate that the value of a variable, implicitly a local, is to be retained across calls.

Here is a procedure that returns the last value it was called with:

```
procedure last(n)
    static last_value

    result := last_value
    last_value := n
    return result
end
```

Usage:

```
][ last(3);
   r := &null  (null)

][ last("abc");
   r := 3  (integer)

][ last(7.4);
   r := "abc"  (string)
```

`static` variables, continued

An `initial` clause can be used to perform one-time initialization. The associated expression is evaluated on the first call to the procedure.

Example:

```
procedure log(s)
    static entry_num
    initial {
        write("Log initialized")
        entry_num := 0
        }

    write(entry_num +:= 1, ": ", s)
end

procedure main()
    log("The first entry")
    log("Another entry")
    log("The third entry")
end
```

Output:

```
Log initialized
1: The first entry
2: Another entry
3: The third entry
```

Procedures—odds & ends

For reference, here is the general form of a procedure:

```
procedure name(param1, ..., paramN)
    local-declarations
    initial-clause
    procedure-body
end
```

The *local-declarations* section is any combination of local and static declarations.

A minimal procedure:

```
procedure f()
end
```

Proper terminology:

> Built-in routines like `read` and `write` are called functions.
>
> Routines written in Icon are called procedures.
>
> `type()` returns "procedure" for both functions and procedures.

Note that every procedure and function either returns a value or fails.

More on compilation

An Icon program may be composed of many procedures. The procedures may be divided among many source files.

If more than one file is named on the `icont` command line, the files are compiled and linked into a single executable. The command

```
% icont roaster.icn db.icn iofuncs.icn
```

compiles the three `.icn` files and produces an executable named `roaster`.

Linking can be suppressed with the `-c` option,

```
% icont -c db.icn iofuncs.icn
```

producing the *ucode* files `db.u1`, `db.u2`, `iofuncs.u1`, and `iofuncs.u2`.

Then, use the `link` directive in the source file: `roaster.icn`:

```
link db, iofuncs
procedure main()
    ...
```

and compile it:

```
% icont roaster.icn
```

`icont` searches the directories named in the `IPATH` environment variable for ucode files named in `link` directives.

ie's .inc command

ie does not currently allow procedures to be defined interactively, but it can load an Icon source file with the .inc (include) command.

Assuming that the procedure double is in the file double.icn, it can be used like this:

```
][ .inc double.icn
][ double(5);
   r := 10  (integer)
][ double("abc");
   r := "abcabc"  (string)
```

With .inc, the included file is recompiled automatically—you can edit in one window, run ie in another, and the latest saved version is used each time.

Procedures—call tracing

One of Icon's debugging facilities is call tracing.

```
1   procedure main()
2       write(sum(3))
3   end
4
5   procedure sum(n)
6       return if n = 0 then 0
7                       else n + sum(n-1)
8   end
```

Execution with tracing:

```
% setenv TRACE -1
% sum
                :       main()
sum.icn         :    2  | sum(3)
sum.icn         :    7  | | sum(2)
sum.icn         :    7  | | | sum(1)
sum.icn         :    7  | | | | sum(0)
sum.icn         :    6  | | | | sum returned 0
sum.icn         :    6  | | | sum returned 1
sum.icn         :    6  | | sum returned 3
sum.icn         :    6  | sum returned 6
6
sum.icn         :    3  main failed
% setenv TRACE 0
% sum
15
```

Handy `csh` aliases:

```
alias tn setenv TRACE -1
alias tf unsetenv TRACE
```

Inside a program, `&trace := -1` turns on tracing.

Augmented assignment

Aside from the assignment and swap operators, every infix operator can be used in an *augmented assignment*.

Examples:

```
i +:= 1

s ||:= read()

x /:= 2

y ^:= 3

i <:= j

s1 >>:= s2
```

There are no unary increment/decrement operators such as i++, but at one point, this was valid:

```
i++++
```

Comments

Icon's only commenting construct is #, which indicates that the rest of the line is a comment:

```
#
# The following code will initialize i
#
i := 0 # i is now initialized
```

In lieu of a block comment capability, Icon's preprocessor can be used:

```
write(1)
$ifdef DontCompileThis
write(2)
write(3)
$endif
write(4)
```

Assuming that `DontCompileThis` hasn't been defined with a `$define` directive, the enclosed `write` statements are excluded from the compilation.

Multi-line string literals

String literals can be continued across lines by ending the line with an underscore. The first non-whitespace character resumes the literal:

```
s := "This is a long _
        literal\n right here   _
          ."
write(s)
```

Output:

```
This is a long literal
 right here   .
```

Note that whitespace preceding the underscore is preserved, but whitespace at the start of a line is elided.

Less efficient, but easier to remember:

```
s := "This is a long " ||
     "literal\n right here   " ||
     "."
write(s)
```

Be sure to put the concatenation operators at the end of a line, not at the beginning!

Substrings

A substring of a string `s` is the string that lies between two positions in `s`.

Positions are thought of as being between characters and run in both directions:

```
  1   2   3   4   5   6   7   8
  |   |   |   |   |   |   |   |

    t   o   o   l   k   i   t

  |   |   |   |   |   |   |   |
 -7  -6  -5  -4  -3  -2  -1   0
```

One way to create a substring is with the form `s[i:j]`, which specifies the portion of `s` between the positions `i` and `j`:

```
][ s := "toolkit";
   r := "toolkit"  (string)

][ s1 := s[2:4];
   r := "oo"  (string)

][ s1;
   r := "oo"  (string)

][ s[-6:-4];
   r := "oo"  (string)

][ s[5:0];
   r := "kit"  (string)

][ s[0:5];
   r := "kit"  (string)
```

Substrings, continued

For reference:

```
  1   2   3   4   5   6   7   8
  |   |   |   |   |   |   |   |
    t   o   o   l   k   i   t
  |   |   |   |   |   |   |   |
 -7  -6  -5  -4  -3  -2  -1   0
```

The form `s[i]` is in fact an abbreviation for `s[i:i+1]`:

```
][ s[1];   (Equivalent to s[1:2])
   r := "t"  (string)

][ s[-1];
   r := "t"  (string)

][ s[-2];
   r := "i"  (string)
```

A substring can be specified as the target of an assignment:

```
][ s[1] := "p";
   r := "p"  (string)

][ s[5:0] := "";
   r := ""  (string)

][ s[-1] := "dle";
   r := "dle"  (string)

][ s;
   r := "poodle"  (string)
```

Substrings, continued

Note that a null substring can be assigned to:

```
][ s := "xy";
   r := "xy"  (string)

][ s[2:2];
   r := ""  (string)

][ s[2:2] := "-";
   r := "-"  (string)

][ s;
   r := "x-y"  (string)
```

Assignment of string values does not cause sharing of data:

```
][ s1 := "string 1";
   r := "string 1"  (string)

][ s2 := "string 2";
   r := "string 2"  (string)

][ s1 := s2;
   r := "string 2"  (string)

][ s1[1:3] := "";
   r := ""  (string)

][ s2;
   r := "string 2"  (string)
```

(In other words, strings use value semantics.)

Substrings, continued

For reference:

```
  1   2   3   4   5   6   7   8
  |   |   |   |   |   |   |   |
    t   o   o   l   k   i   t
  |   |   |   |   |   |   |   |
 -7  -6  -5  -4  -3  -2  -1   0
```

Another subscripting syntax is `s[i+:n]`, which is equivalent to `s[i:i+n]`:

```
][ s[4+:2];
   r := "lk"  (string)

][ s[-3+:3];
   r := "kit"  (string)

][ s[-5+:3];
   r := "olk"  (string)
```

A related form is `s[i-:n]`, which is equivalent to `s[i:i-n]`:

```
][ s[5-:4];
   r := "tool"  (string)

][ s[0-:3];
   r := "kit"  (string)

][ s[-2-:2];
   r := "lk"  (string)
```

In essence, all substring specifications name the string of characters between two positions.

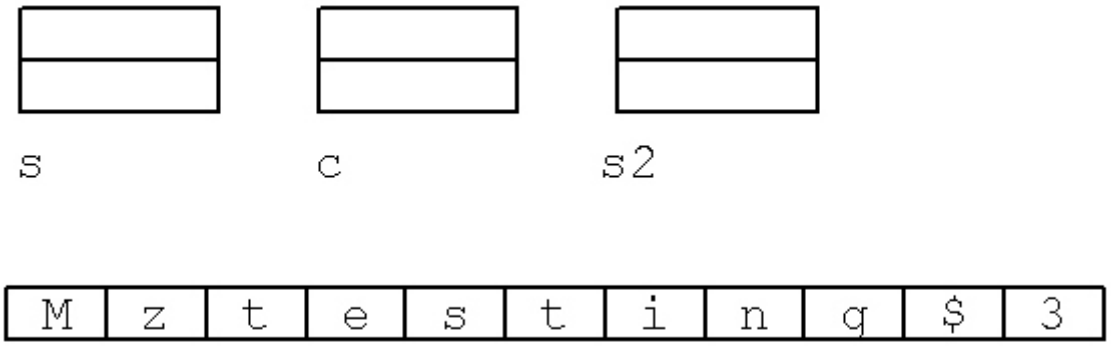
Sidebar: Implementation of substrings

Problem: Speculate on how substrings are implemented.

Code to work with:

```
s := "testing"
c := s[1]
s2 := s[2:-1]
```

Memory:



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
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 can produce no more results and fails.

To put it anthropomorphically, `every` is never satisfied with the result of the control expression.

`while` repeatedly evaluates its control expression, executing the loop until the expression fails.

An infinite loop:

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

Output:

```
Gen: Starting up...
Result = 3
Gen: Starting up...
Result = 3
```

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...
```

Generator basics, continued

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
0
Gen: Starting up...
Gen: More computing...
Gen: Still computing...
Gen: Out of gas...
0 not found
```

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

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

One way to print the odd numbers from 1 to 100:

```
every write(1 to 100 by 2)
```

Problems:

(1) Do it without "`every`"

(2) Instead of using "`by`" use the remainder operator (%) and goal directed evaluation.

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

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
```

How could the vowel counter be changed 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 !3.141592;
   "3"  (string)
   "."  (string)
   "1"  (string)
   "4"  (string)
   "1"  (string)
   "5"  (string)
   "9"  (string)
   "2"  (string)
```

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

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)
```