Fundamentals of the Icon Programming Language

Lecture Slides

Author: William. H. Mitchell (whm)
Mitchell Software Engineering (.com)

What is Icon?

Icon is a high-level, general purpose, imperative language with a traditional appearance, but with several interesting aspects:

- A rich set of built-in data types
- A rich but cohesive and orthogonal set of operators and functions
- A novel expression evaluation mechanism
- An integrated facility for analysis of strings
- Automatic memory management (garbage collection)
- A small "mental footprint"

The philosophy of Icon: (in my opinion)

- Provide a “critical-mass” of types and operations
- Give the programmer as much freedom as possible
- Put the burden of efficiency on the language implementation

Another opinion: Every programmer should have a language like Icon in their “toolbox”.

A little history

Icon is a descendent of SNOBOL4 and SL5.

Icon was designed at the University of Arizona in the late 1970s by a team lead by Ralph Griswold.

Last major upheaval in the language itself was in 1982, but a variety of minor elements have been added in the years since.

Idol, an object-oriented derivative was developed in 1988 by Clint Jeffery.

Graphics extensions evolved from 1990 through 1994.

Unicon (Unified Extended Icon) evolved from 1997 through 1999 and incremental change continues. Unicon has support for object-oriented programming, systems programming, and programming-in-the-large.

The origin of the name "Icon" is clouded. Some have suggested it comes from "iconoclast".

Running Icon

One way to work with Icon is to put an entire program into a file, translate it into a bytecode executable, and run it.

A more interactive option is the Icon Evaluator, `ie`, which evaluates Icon expressions:

```
% ie
Icon Evaluator, Version 0.8.1, ? for help
][ 3+4;
   r1 := 7  (integer)

][ 3.4*5.6;
   r2 := 19.04  (real)

][ "x" || "y" || "z";
   r3 := "xyz"  (string)

][ reverse(r3);
   r4 := "zyx"  (string)

][ center("hello",20,".");
   r5 := ".......hello........"  (string)

][ ^D     (control-D to exit)

%
```

Variables

Variables can be declared explicitly but the more common practice is to simply name variables when needed.

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

][ x;
   r := 7  (integer)

][ y := x + 10;
   r := 17  (integer)

][ y;
   r := 17  (integer)
```

Variable names may consist of any number of letters, digits, and underscores and must start with letter or underscore.

Variable names, along with everything else in Icon, are case-sensitive.

Note that the result of assignment is the value assigned.

Variables, continued

Uninitialized variables have a null value:

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

A variable may be assigned the null value:

```
][ x := 30;
   r := 30  (integer)

][ x := &null;
   r := &null  (null)

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

`&null` is one of many Icon *keywords*—special identifiers whose name is prefixed with an ampersand.

Variables, continued

Icon variables have no type associated with them. Instead, types are associated with values themselves.

Any variable may be assigned a value of any type and then later assigned a value of a different type:

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

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

][ x := 3.4;
   r := 3.4  (real)

][ x;
   r := 3.4  (real)

][ x := 100;
   r := 100  (integer)

][ x;
   r := 100  (integer)
```

Note that there is no way to declare the type of a variable.

Variables, continued

The type of a value can determined with the `type` function:

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

][ type(3/4);
   r := "integer"  (string)

][ type(3.0/4.0);
   r := "real"  (string)

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

If the argument of `type` is a variable, it is the type of the value held by the variable that is reported:

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

][ type(type);
   r := "procedure"  (string)

][ type(xyz); (no value assigned...)
   r := "null"  (string)
```

Arithmetic operations

Integers and reals are collectively referred to as numeric types.

Icon's arithmetic operators for numeric types:

+	addition
-	subtraction
*	multiplication
/	division
%	remaindering (reals are allowed)
^	exponentiation
-	negation (unary operator)
+	(unary operator)

Examples:

```
][ 30 / 4;
   r := 7  (integer)

][ 30 / 4.0;
   r := 7.5  (real)

][ 2.3 % .4;
   r := 0.3  (real)

][ -r;
   r := -0.3  (real)

][ +-3;
   r1 := -3  (integer)
```

A binary arithmetic operator produces an integer result only if both operands are integers.

Arithmetic operations, continued

Exponentiation:

```
][ 2 ^ 3;
   r := 8  (integer)

][ 100 ^ .5;
   r := 10.0  (real)
```

Some implementations of Icon support infinite precision integer arithmetic:

```
][ x := 2 ^ 70;
   r := 1180591620717411303424  (integer)

][ y := 2 ^ 62;
   r := 4611686018427387904  (integer)

][ x / y;
   r := 256  (integer)
```

`integer` is the only integer type in Icon; `real` is the only floating point type in Icon.

Conversion between types

Icon freely converts between integers, reals, and strings if a supplied value is not of the required type:

```
][ x := 3.4 * "5";
   r := 17.0  (real)

][ x := x || x;
   r := "17.017.0"  (string)

][ x;
   r := "17.017.0"  (string)

][ q := "100"/2;
   r := 50  (integer)

][ q := "100.0"/2;
   r := 50.0  (real)

][ q := "1e2"/2;
   r := 50.0  (real)

][ q := q || q;
   r := "50.050.0"  (string)
```

Icon never converts `&null` to a value of an appropriate type:

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

][ xyz + 10;

Run-time error 102
numeric expected
offending value: &null
```

Strings

The `string` type represents character strings of arbitrary length.

String literals are delimited by double quotes:

```
"just a string right here"
```

Any character can appear in a string.

Characters can be specified using escape sequences:

`\n`	newline
`\t`	tab
`\"`	double quote
`\\`	backslash
`\`*ooo*	octal character code
`\x`*hh*	hexadecimal character code
`\^`*c*	control character c

Example:

```
][ "\n\012\x0a\^j";
   r := "\n\n\n\n"  (string)

][ "A\x41\101 Exterminators";
   r := "AAA Exterminators"  (string)
```

For the full set of string literal escapes, see page 254 in the text.

Strings, continued

The string concatenation operator is `||` (two "or" bars):

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

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

][ s3 := s1 || " " || s2;
   r := "Fish Knuckles"  (string)
```

The unary `*` operator is used throughout Icon to calculate the "size" of a value. For strings, the size is the number of characters:

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

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

][ *( s || s);
   r := 6  (integer)

][ *s || s;
   r := "3abc"  (string)
```

The operator `*` is said to be *polymorphic* because it can be applied to values of many types.

Strings, continued

Strings can be subscripted with the `[]` operator:

```
][ letters := "abcdefghijklmnopqrstuvwxyz";
   r := "abcdefghijklmnopqrstuvwxyz"  (string)

][ letters[1];
   r := "a"  (string)

][ letters[*letters];
   r := "z"  (string)

][ letters[5] || letters[10] || letters[15];
   r := "ejo"  (string)
```

The first character in a string is at position 1, not 0.

A character can be changed with assignment:

```
][ letters[13] := "X";
   r := "X"  (string)

][ letters;
   r := "abcdefghijklXnopqrstuvwxyz"  (string)
```

A little fun—Icon has a swap operator:

```
][ letters[1] :=: letters[26];
   r := "z"  (string)

][ letters;
   r := "zbcdefghijklXnopqrstuvwxya"  (string)
```

Note that there is no character data type in Icon; single characters are simply represented by one-character strings.

Strings, continued

Icon has a number of built-in functions and a number of them operate on strings.

Appendix A in the text enumerates the full set of built-in functions starting on page 275[1]. The function descriptions take this form:

repl(s1, i) : s2 replicate string

repl(s1, i) produces a string consisting of i concatenations of s1

Errors:

101	i not integer
103	s1 not string
205	i < 0
306	inadequate space in string region

Usage of `repl`:

```
][ repl("x", 10);
   r := "xxxxxxxxxx"  (string)

][ *repl(r, 100000);
   r := 1000000  (integer)
```

[1] Icon reference material is on the Web at
http://www.cs.arizona.edu/icon/refernce/ref.htm

Failure

A unique aspect of Icon is that expressions can fail to produce a result. A simple example of an expression that fails is an out of bounds string subscript:

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

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

][ s[50];
Failure
```

It is said that "`s[50]` fails"—it produces no value.

If an expression produces a value it is said to have *succeeded*.

When an expression is evaluated it either succeeds or fails.

Failure, continued

An important rule:

> An operation is performed only if a value is present for all operands. If a value is not present for all operands, the operation fails.

Another way to say it:

> If evaluation of an operand fails, the operation fails.

Examples:

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

][ "x" || s[50];
Failure

][ reverse("x" || s[50]);
Failure

][ s := reverse("x" || s[50]);
Failure

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

Note that failure propagates.

Failure, continued

Another example of an expression that fails is a comparison whose condition does not hold:

```
][ 1 = 0;
Failure

][ 4 < 3;
Failure

][ 10 >= 20;
Failure
```

A comparison that succeeds produces the value of the right hand operand as the result of the comparison:

```
][ 1 < 2;
   r := 2  (integer)

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

][ 10 ~= 20;
   r := 20  (integer)
```

What do these expressions do?

```
max := max < n

x := 1 + 2 < 3 * 4 > 5
```

Failure, continued

Fact:

Unexpected failure is the root of madness.

Consider this code:

```
write("Before make_block")
text := make_block(x, y, z)
write(text[10])
write("After make_block")
```

Output:

```
Before make_block
After make_block
```

Problem:

Contrast expression failure to Java's exception handling facility.

Producing output

The built-in function `write` prints a string representation of each of its arguments and appends a final newline.

```
][ write(1);
1
   r := 1  (integer)

][ write("r is ", r);
r is 1
   r := 1  (integer)

][ write(r, " is the value of r");
1 is the value of r
   r := " is the value of r"  (string)

][ write(1,2,3,"four","five","six");
123fourfivesix
   r := "six"  (string)
```

`write` returns the value of the last argument.

If an argument has the null value, a null string is output:

```
][ write("x=", x, ",y=", y, ".");
x=,y=.
   r := "."  (string)
```

The built-in function `writes` is identical to `write`, but it does not append a newline.

Reading input

The built-in function `read()` reads one line from standard input.

```
][ line := read();
Here is some input   (typed by user)
   r := "Here is some input"  (string)

][ line2 := read();
                        (user pressed <ENTER>)
   r := ""  (string)
```

On end of file, such as a control-D from the keyboard, `read` fails:

```
][ line := read();
^D
Failure
```

Question: What is the value of `line`?

The `while` expression

Icon has several traditionally-named control structures, but they are driven by success and failure.

The general form of the `while` expression is:

```
while expr1 do
    expr2
```

If *`expr1`* succeeds, *`expr2`* is evaluated. This continues until *`expr1`* fails.

Here is a loop that reads lines and prints them:

```
while line := read() do
    write(line)
```

If no body is needed, the `do` clause can be omitted:

```
while write(read())
```

What does the following code do?

```
while line := read()
    write(line)
```

Problem: Write a loop that prints "yes" repeatedly.

Compound expressions

A compound expression groups a series of expressions into a single expression.

The general form of a compound expression is:

```
{ expr1; expr2; ...; exprN }
```

Each expression is evaluated in turn. The result of the compound expression is the result of exprN, the last expression:

```
][ { write(1); write(2); write(3)};
1
2
3
   r := 3  (integer)
```

A failing expression does not stop evaluation of subsequent expressions:

```
][ { write(1); write(2 < 1); write(3)};
1
3
   r := 3  (integer)
```

Compound expressions, continued

Recall the general form of the `while` expression:

```
while expr1 do
    expr2
```

Here the body of a `while` loop is a compound expression:

```
line_count := 0;

while line := read() do {
    write(line);
    line_count := line_count + 1;
    }

write(line_count, " lines read");
```

Semicolon insertion

The Icon translator will "insert" a semicolon if an expression ends on one line and the next line begins with another expression.

Given this multi-line input:

```
{
    write(1)
    write(2)
    write(3)
}
```

The translator considers it to be:

```
{
    write(1);
    write(2);
    write(3)
}
```

It is standard practice to rely on the translator to insert semicolons. But, there is a danger of an unexpected insertion of a semicolon:

```
][ { x := 3
...       - 2 };
   r := -2  (integer)
```

A good habit: Always break expressions <u>after</u> an operator:

```
][ { x := 3 -
...       2 };
   r := 1  (integer)
```

Problem: Reversal of line order

Write a segment of code that reads lines from standard input and upon end of file, prints the lines in reverse order.

For this input:

```
line one
the second line
#3
```

The output is:

```
#3
the second line
line one
```

Problem: Line numbering with a twist

Write a segment of code that reads lines from standard input and produces a numbered listing of those lines on standard output.

For this input:

```
just
testing
this
```

We want this output:

```
     1  just
     2  testing
     3  this
```

Line numbers are to be right justified in a six-character field and followed by two spaces. The text from the input line then follows immediately.

Ignore the possibility that more than 999,999 lines might be processed.

The twist: Don't use any digits in your code.

Handy: The `right(s,n)` function right-justifies the string `s` in a field of width `n`:

```
][ right("abc", 5);
   r := "  abc"  (string)
```

if-then-else

The general form of the `if-then-else` expression is

```
if expr1 then expr2 else expr3
```

If *expr1* succeeds the result of the if-then-else expression is the result of *expr2*. If *expr1* fails, the result is the result of *expr3*.

```
][ if 1 < 2 then 3 else 4;
   r := 3  (integer)

][ if 1 > 2 then 3 else 4;
   r := 4  (integer)

][ if 1 < 2 then 2 < 3 else 4 < 5;
   r := 3  (integer)

][ if 1 > 2 then 2 > 3 else 4 > 5;
Failure
```

Explain this expression:

```
label := if min < x < max then
            "in range"
         else
            "out of bounds"
```

`if-then-else`, continued

There is also an if-then expression:

```
if expr1 then expr2
```

If *`expr1`* succeeds, the result of the if-then expression is the result of *`expr2`*. If *`expr1`* fails, the if-then fails.

Examples:

```
][ if 1 < 2 then 3;
   r := 3  (integer)

][ if 1 > 2 then 3;
Failure
```

What is the result of this expression?

```
x := 5 + if 1 > 2 then 3
```

One way to nest if-then-elses:

```
if (if x < y then x else y) > 5 then
     (if x > 6 then 7)
else
    (if x < 8 then 9)
```

The if-then-else and if-then expressions are considered to be control structures rather than operators.

A characteristic of a control structure is that a constituent expression can fail without terminating evaluation of the containing expression (i.e., the control structure).

The `break` and `next` expressions

The `break` and `next` expressions are similar to `break` and `continue` in Java.

This is a loop that reads lines from standard input, terminating on end of file or when a line beginning with a period is read. Each line is printed unless the line begins with a # symbol.

```
while line := read() do {
    if line[1] == "." then
        break

    if line[1] == "#" then
        next

    write(line)
    }
```

The operator == tests equality of two strings.

The `not` expression

The `not` expression, a control structure, has this form:

```
not expr
```

If *expr* produces a result, the `not` expression fails.

If *expr* fails, the `not` expression produces the null value.

Examples:

```
][ not 0;
Failure

][ not 1;
Failure

][ not (1 > 2);
   r := &null  (null)

][ if not (1 > 2) then write("ok");
ok
   r := "ok"  (string)
```

`not` has very high precedence. As a rule its *expr* should always be enclosed in parentheses.

Question: Could `not` be implemented as an operator rather than a control structure?

The & operator

The general form of the & operator:

```
expr1 & expr2
```

expr1 is evaluated first. If *expr1* succeeds, *expr2* is evaluated. If *expr2* succeeds, the entire expression succeeds and produces the result of *expr2*. If either *expr1* or *expr2* fails, the entire expression fails.

Examples:

```
][ 1 & 2;
   r := 2  (integer)

][ 0 & 2 < 4;
   r := 4  (integer)

][ r > 3 & write("r = ", r);
r = 4
   r := 4  (integer)

][ while line := read() & line[1] ~== "." do
...     write(line);
a
a
test
test
.here
Failure
```

& has the lowest precedence of any operator.

Problem: Describe the implementation of the & operator.

Comparison operators

There are six operators for comparing values as numeric quantities:

```
<     >     <=     >=     =     ~=
```

There are six operators for comparing values as strings:

```
<<     >>     <<=     >>=     ==     ~==
```

Question: Why aren't the comparison operators overloaded so that one set of operators would suffice for both numeric and string conversions?

Comparison operators, continued

Analogous comparison operators can produce differing results for a given pair of operands:

```
][ "01" = "1";
   r := 1  (integer)

][ "01" == "1";
Failure

][ "01" < "1";
Failure

][ "01" << "1";
   r := "1"  (string)
```

The === and ~=== operators test for exact equivalence—both the type and value must be identical:

```
][ 2 === "2";
Failure

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

][ "xyz" === "x" || "y" || "z";
   r := "xyz"  (string)
```

Comparison operators, continued

The unary operators `/` and `\` test to see if a value is null or not, respectively.

The expression `/`*`expr`* succeeds and produces *`expr`* if *`expr`* has a null value.

The expression `\`*`expr`* succeeds and produces *`expr`* if *`expr`* has a non-null value.

Examples:

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

][ \x;
Failure

][ /x;
   r := &null  (null)

][ /x := 5;
   r := 5  (integer)

][ /x := 10;
Failure

][ x;
   r := 5  (integer)
```

As a mnemonic aid, think of `/x` as succeeding when `x` is null because the null value allows the slash to fall flat.

Explicit conversions

In addition to the implicit conversions that Icon automatically performs as needed, there are conversion functions to produce a value of a specific type from a given value.

The functions `integer` and `real` attempt to produce a integer or real value, respectively. `numeric` produces either an integer or a real, preferring integers. Examples:

```
][ integer("12");
   r := 12  (integer)

][ integer(.01);
   r := 0  (integer)

][ real("12");
   r := 12.0  (real)

][ real("xx");
Failure

][ numeric("12");
   r := 12  (integer)

][ numeric("12.0");
   r := 12.0  (real)
```

The `string` function produces a string corresponding to a given value.

```
][ string(2^32);
   r := "4294967296"  (string)

][ string(234.567e-30);
   r := "2.34567e-28"  (string)
```

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:

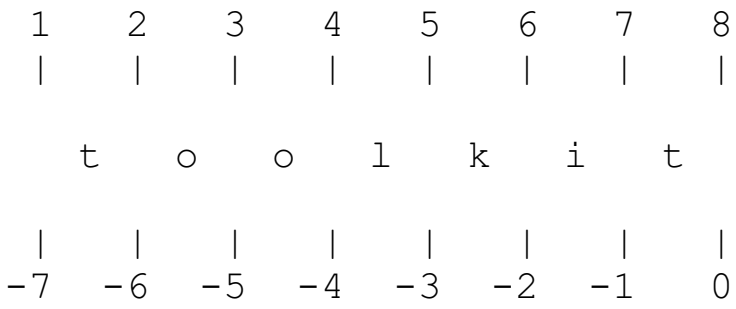
```
s := "This is a long " ||
     "literal\n right here   " ||
     "."
write(s)
```

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:



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.

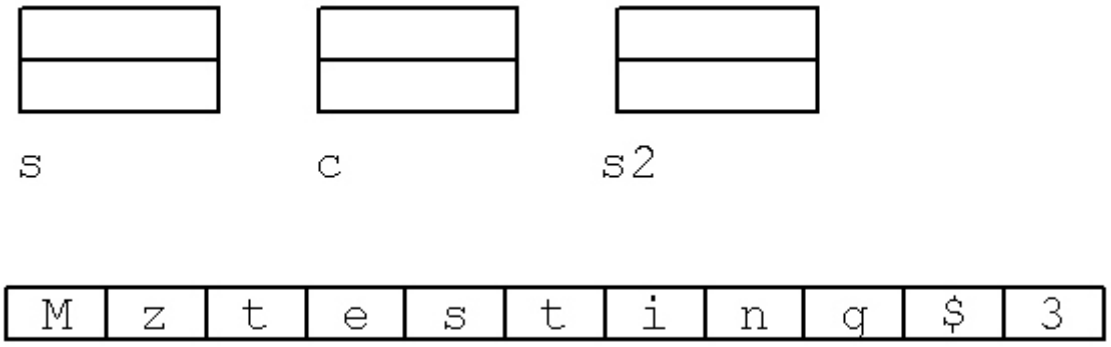
Sidebar: Implementation of substrings

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  # 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 any context 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` *expr1* `do` *expr2*, both expressions are bounded.

In `every` *expr1* `do` *expr2*, 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 expr1 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.

Lists

Icon has a `list` data type. Lists hold sequences of values.

One way to create a list:

```
][ [1,2,3];
   r := [1,2,3]  (list)
```

A given list may hold values of differing types:

```
][ L := [1, "two", 3.0, []];
   r := [1,"two",3.0,[]]  (list)
```

An element of a list may be referenced by subscripting:

```
][ L[1];
   r := 1  (integer)

][ L[2];
   r := "two"  (string)

][ L[-1];
   r := []  (list)

][ L[10];
Failure
```

The other way to create a list:

```
][ list(5, "a");
   r := ["a","a","a","a","a"]  (list)
```

Lists, continued

A *list section* may be obtained by specifying two positions:

```
][ L := [1, "two", 3.0, []];
   r := [1,"two",3.0,[]]  (list)

][ L[1:3];
   r := [1,"two"]  (list)

][ L[2:0];
   r := ["two",3.0,[]]  (list)
```

Note the asymmetry between subscripting and sectioning: subscripting produces an element, sectioning produces a list.

```
][ L[2:3];
   r := ["two"]  (list)

][ L[2];
   r := "two"  (string)

][ L[2:2];
   r := []  (list)
```

Contrast with strings:

```
][ s := "123";
   r := "123"  (string)

][ s[2:3];
   r := "2"  (string)

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

Question: What is the necessary source of this asymmetry?

Lists, continued

Recall `L`:

```
][ L;
   r := [1,"two",3.0,[]]  (list)
```

Lists may be concatenated with `|||`:

```
][ [1] ||| [2] ||| [3];
   r := [1,2,3]  (list)

][ L[1:3] ||| L[2:0];
   r := [1,"two","two",3.0,[]]  (list)
```

Concatenating lists is like concatenating strings—a new list is formed:

```
][ L := [1, "two", 3.0, []];
   r := [1,"two",3.0,[]]  (list)

][ L := L ||| [9999] ||| L ||| [];
   r := [1,"two",3.0,[],9999,1,"two",3.0,[]]
```

For the code below, what is the final value of `nines`?

```
nines := []
every nines |||:= |[9] \ 7
```

Lists, continued

The number of top-level elements in a list may be calculated with *:

```
][ L := [1, "two", 3.0, []];
   r := [1,"two",3.0,[]]  (list)

][ *L;
   r := 4  (integer)

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

Problem: What is the value of the following expressions?

```
*[[1,2,3]]

*[L, L,[[]]]

*[,,]

**[[],[]]

*(list(1000000,0) ||| list(1000000,1))
```

Lists, continued

List elements can be changed via assignment:

```
][ L := [1,2,3];
   r := [1,2,3]  (list)

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

][ L[-1] := "last element";
   r := "last element"  (string)

][ L;
   r := [10,2,"last element"]  (list)
```

List sections cannot be assigned to:

```
][ L[1:3] := [];
Run-time error 111
variable expected
...
```

Problem: Write a procedure `assign(L1, i, j, L2)` that approximates the operation `L1[i:j] := L2`.

Complex subscripts and sections

Lists within lists can be referenced by a series of subscripts:

```
][ L := [1,[10,20],[30,40]];
   r := [1,[10,20],[30,40]]  (list)

][ L[2];
   r := [10,20]  (list)

][ L[2][1];
   r := 10  (integer)

][ L[2][1] := "abc";
   r := "abc"  (string)

][ L;
   r := [1,["abc",20],[30,40] ]  (list)
```

A series of subscripting operations to reference a substring of a string-valued second-level list element:

```
][ L[2][1];
   r := "abc"  (string)

][ L[2][1][2:0] := "pes";
   r := "pes"  (string)

][ L;
   r := [1,["apes",20],[30,40]]  (list)

][ every write(!L[2][1][2:4]);
p
e
Failure
```

Lists as stacks and queues

The functions `push`, `pop`, `put`, `get`, and `pull` provide access to lists as if they were stacks, queues, and double-ended queues.

`push(L, expr)` adds `expr` to the left end of list `L` and returns `L` as its result:

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

][ push(L, 1);
   r := [1]  (list)

][ L;
   r := [1]  (list)

][ push(L, 2);
   r := [2,1]  (list)

][ push(L, 3);
   r := [3,2,1]  (list)

][ L;
   r := [3,2,1]  (list)
```

Lists as stacks and queues, continued

`pop(L)` removes the leftmost element of the list `L` and returns that value. `pop(L)` fails if `L` is empty.

```
][ L;
   r := [3,2,1]  (list)

][ while e := pop(L) do
...     write(e);
3
2
1
Failure

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

Note that the series of pops clears the list.

A program to print the lines in a file in reverse order:

```
procedure main()
    L  := []
    while push(L, read())
    while write(pop(L))

end
```

With generators:

```
procedure main()
    L := []
    every push(L, !&input)
    every write(!L)
end
```

Lists as stacks and queues, continued

`push` returns its first argument:

```
][ x := push(push(push([],10),20),30);
   r := [30,20,10]  (list)

][ x;
   r := [30,20,10]  (list)
```

`put(L, expr)` adds `expr` to the right end of `L` and returns `L` as its result:

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

][ put(L, "b");
   r := ["a","b"]  (list)

][ every put(L, 1 to 3);
Failure

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

Lists as stacks and queues, continued

`get(L)`, performs the same operation as `pop(L)`, removing the leftmost element of the list `L` and returning that value. `get(L)` fails if `L` is empty.

Yet another way to print the numbers from 1 to 10:

```
L := []
every put(L, 1 to 10)
while write(get(L))
```

`pull(L)` removes the rightmost element of the list `L` and returns that value. `pull(L)` fails if `L` is empty.

```
][ L := [1,2,3,4];
   r := [1,2,3,4]  (list)

][ while write(pull(L));
4
3
2
1
Failure

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

Any of the five functions `push`, `pop`, `put`, `get`, and `pull` can be used in any combination on any list.

A visual summary:

```
push  ==>           ==> pull
pop   <==  List    <== put
get   <==
```

List element generation

When applied to lists, `!` generates elements:

```
][ .every ![1, 2, ["a", "b"], 3.0, write];
   1  (integer)
   2  (integer)
   ["a","b"]  (list)
   3.0  (real)
   function write  (procedure)
```

Problem: Write a procedure `common(L1, L2, L3)` that succeeds if the three lists have an integer value in common. Easy: Assume that the lists contain only integers. Hard: Don't assume that.

Problem: Write procedures `explode(s)` and `implode(L)` such as those found in ML.

```
][ explode("test");
   r := ["t","e","s","t"]  (list)

][ implode(r);
   r := "test"  (string)
```

Sorting lists

The function `sort(L)` produces a sorted copy of the list `L`. `L` is not changed.

```
][ L := [5,1,10,7,-15];
   r := [5,1,10,7,-15]  (list)

][ Ls := sort(L);
   r := [-15,1,5,7,10]  (list)

][ L;
   r := [5,1,10,7,-15]  (list)

][ Ls;
   r := [-15,1,5,7,10]  (list)
```

Lists need not be homogeneous to be sorted:

```
][ sort(["a", 10, "b", 1, 2.0, &null]);
   r := [&null,1,10,2.0,"a","b"]  (list)
```

Values are ordered first by type, then by value. Page 161 in the text shows the type ordering used for heterogenous lists.

A program to sort lines of standard input:

```
procedure main()
    L := []
    while put(L, read())
    every write(!sort(L))
end
```

Problem: Describe two distinct ways to sort lines in descending order.

Sorting lists, continued

Sorting a list of lists orders the lists according to their order of creation—not usually very useful.

The `sortf(L, i)` function sorts a list of lists according to the `i`-th element of each list:

```
][ L := [[1, "one"], [8, "eight"], [2, "two"]];
   r := [[1,"one"], [8,"eight"], [2,"two"]]

][ sortf(L, 1);
   r := [[1,"one"],[2,"two"],[8,"eight"]]

][ sortf(L, 2);
   r := [[8,"eight"], [1,"one"], [2,"two"]]
```

The value `i` can be negative, but not zero.

Lists without an `i`-th element sort ahead of other lists.

Lists in a nutshell

- Create with `[`*`expr`*`, ...]` and `list(`*`N`*`, `*`value`*`)`
- Index and section like strings
  Can't assign to sections
- Size and element generation like strings
- Concatenate with `|||`
- Stack/queue access with `put`, `push`, `get`, `pop`, `pull`
  Parameters are consistent: list first, then value
- Sort with `sort` and `sortf`

Challenge:

Find another language where equivalent functionality can be described as briefly.

Reference semantics for lists

Some types in Icon use *value semantics* and others use *reference semantics*.

Strings use value semantics:

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

][ s2 := s1;
   r := "string 1"  (string)

][ s2[1] := "x";
   r := "x"  (string)

][ s1;
   r := "string 1"  (string)
][ s2;
   r := "xtring 1"  (string)
```

Lists use reference semantics:

```
][ L1 := [1,2,3];
   r := [1,2,3]  (list)

][ L2 := L1;
   r := [1,2,3]  (list)

][ L2[1] := "x";
   r := "x"  (string)

][ L1;
   r := ["x",2,3]  (list)

][ L2;
   r := ["x",2,3]  (list)
```

Reference semantics for lists, continued

Earlier examples of list operations with `ie` have been edited. What `ie` really shows for list values:

```
][ lst1 := [1,2,3];
   r := L1:[1,2,3]  (list)

][ lst2 := [[],[],[]];
   r := L1:[L2:[],L3:[],L4:[]]  (list)
```

The `L`*n* tags are used to help identify lists that appear multiple times:

```
][ [lst1, lst1, [lst1]];
   r := L1:[L2:[1,2,3],L2,L3:[L2]]  (list)
```

Consider this:

```
][ lst := [1,2];
   r := L1:[1,2]  (list)

][ lst[1] := lst;
   r := L1:[L1,2]  (list)
```

Then this:

```
][ lst[1][2] := 10;
   r := 10  (integer)

][ lst;
   r := L1:[L1,10]  (list)
```

Reference semantics for lists, continued

More:

```
][ X := [1,2,3];
   r := L1:[1,2,3]  (list)

][ push(X,X);
   r := L1:[L1,1,2,3]  (list)

][ put(X,X);
   r := L1:[L1,1,2,3,L1]  (list)

][ X[3] := [[X]];
   r := L1:[L2:[L3:[L3,1,L1,3,L3]]]  (list)

][ X;
   r := L1:[L1,1,L2:[L3:[L1]],3,L1]  (list)
```

Explain this:

```
][ L := list(5,[]);
   r := L1:[L2:[],L2,L2,L2,L2]  (list)
```

Reference semantics for lists, continued

An important aspect of list semantics is that equality of two list-valued expressions is based on whether the expressions reference the same list object in memory.

```
][ lst1 := [1,2,3];
   r := L1:[1,2,3]  (list)

][ lst2 := lst1;
   r := L1:[1,2,3]  (list)

][ lst1 === lst2;
   r := L1:[1,2,3]  (list)

][ lst2 === [1,2,3];
Failure

][ [1,2,3] === [1,2,3];
Failure

][ [] === [];
Failure
```

Reference semantics for lists, continued

Icon uses call-by-value for transmission of argument values to a procedure.

However, an argument is a type such as a list, which uses reference semantics, the value passed is a reference to the list itself. Changes made to the list will be visible to the caller.

An extension of the procedure `double` to handle lists:

```
procedure double(x)
    if type(x) == "string" then
        return x || x
    else if numeric(x) then
        return 2 * x
    else if type(x) == "list" then {
        every i := 1 to *x do
            x[i] := double(x[i])
        return x
        }
    else
        fail
end
```

Usage: (note that `L` is changed)

```
][ L := [3, "abc", [4.5, ["xx"]]];
   r := [3, "abc", [4.5, ["xx"]]]  (list)

][ double(L);
   r := [6, "abcabc", [9.0, ["xxxx"]]]  (list)

][ L;
   r := [6, "abcabc", [9.0, ["xxxx"]]]  (list)
```

image and Image

Lists cannot be output with the write function. To output lists, the image and Image routines may be used.

The built-in function image(X) produces a string representation of any value:

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

][ image("s");
   r := "\"s\""  (string)

][ write(image("s"));
"s"
   r := "\"s\""  (string)

][ image(write);
   r := "function write"  (string)

][ image([1,2,3]);
   r := "list_13(3)"  (string)
```

For lists, image only shows a "serial number" and the size.

`image` and `Image`, continued

The Icon procedure `Image` can be used to produce a complete description of a list (or any value):

```
][ write(Image([1,2,[],4]));
L3:[
  1,
  2,
  L4:[],
  4]
   r := "L3:[\n  1,\n  2,\n  L4:[],\n  4]"
```

Note that `Image` produces a string, which in this case contains characters for formatting.

An optional second argument of 3 causes `Image` to produce a string with no formatting characters:

```
][ write(Image([1,2,[],4], 3));
L8:[1,2,L9:[],4]
   r := "L8:[1,2,L9:[],4]"  (string)
```

`Image` is not a built-in function; it must be linked:

```
link image
procedure main()
   ...
end
```

Simple text processing with `split`

A number of text processing problems can be addressed with a simple concept: splitting a line into pieces based on delimiters and then processing those pieces.

There is a procedure `split(s, delims)` that returns a list consisting of the portions of the string `s` delimited by characters in `delims`:

```
][ split("just a test here ", " ");
   r := ["just","a","test","here"]  (list)

][ split("...1..3..45,78,,9 10  ", "., ");
   r := ["1","3","45","78","9","10"]  (list)
```

`split` is not a built-in function; it must be linked:

```
link split
procedure main()
  ...
end
```

`split`, continued

Consider a file whose lines consist of zero or more integers separated by white space:

```
5 10 0 50

200
1 2 3 4 5 6 7 8 9 10
```

A program to sum the numbers in such a file:

```
link split
procedure main()
    sum := 0
    while line := read() do {
        nums := split(line, " \t")
        every num := !nums do
            sum +:= num
        }

    write("The sum is ", sum)
end
```

Problem: Trim down that flabby code!

```
procedure main()
    sum := 0

    write("The sum is ", sum)
end
```

If `split` has a third argument that is non-null, both delimited and delimiting pieces of the string are produced:

```
][ split("520-621-6613", "-", 1);
   r := ["520","-","621","-","6613"]  (list)
```

`split`, continued

Write a procedure `extract(s, m, n)` that extracts a portion of a string `s` that represents a hierarchical data structure. `m` is a major index and `n` is a minor index. Major sections of the string are delimited by slashes and are composed of minor sections separated by colons. Here is a sample string:

`/a:b/apple:orange/10:2:4/xyz/`

It has four major sections which in turn have two, two, three and one minor sections.

A call such as `extract(s, 3, 2)` locates the third major section (`"10:2:4"`) and return the second minor section (`"2"`).

```
][ extract(s, 1, 2);
   r := "b"  (string)

][ extract(s, 4, 1);
   r := "xyz"  (string)

][ extract(s, 4, 2);
Failure
```

Command line arguments

The command line arguments for an Icon program are passed to `main` as a list of strings.

```
procedure main(a)
    write(*a, " arguments:")
    every write(image(!a))
end
```

Execution:

```
% args just "a test" right here
4 arguments:
"just"
"a test"
"right"
"here"
% args
0 arguments:
%
```

Problem: Write a program `picklines` that reads lines from standard input and prints ranges of lines specified by command line arguments. Lines may be referenced from the end of file, with the last line being -1.

Examples:

```
picklines 1 2 3 2 1 < somefile

picklines 1..10 30 40 50 < somefile

picklines 1..10 -10..-1 < somefile
```

picklines—Solution

```
link split
procedure main(args)
    lines := []
    while put(lines, read())

    picks := []
    every spec := !args do {
        w := split(spec, ".")
        every put(picks, lines[w[1]:w[-1]+1])
        }

    every write(!!picks)
end
```

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.

Tables

Icon's `table` data type can be thought of as an array that can be subscripted with values of any type.

The built-in function `table` is used to create a table:

```
][ t := table();
   r := T1:[]  (table)
```

To store values in a table, simply assign to an element specified by a subscript (sometimes called a *key*):

```
][ t[1000] := "x";
   r := "x"  (string)

][ t[3.0] := "three";
   r := "three"  (string)

][ t["abc"] := [1];
   r := L1:[1]  (list)
```

Values are referenced by subscripting.

```
][ t["abc"];
   r := L1:[1]  (list)

][ t[1000];
   r := "x"  (string)
```

Tables, continued

Tables can't be output with `write()`, but `Image` can describe the contents of a table:

```
][ write(Image(t));
T1:[
  1000->"x",
  3.0->"three",
  "abc"->L1:[1]
  ]
```

Assigning a value using an existing key simply causes the old value to be replaced:

```
][ t[3.0] := "Here's 3.0";
   r := "Here's 3.0"  (string)

][ t["abc"] := "xyz";
   r := "xyz"  (string)

][ t[1000] := &null;
   r := &null  (null)

][ write(Image(t));
T2:[
  1000->&null,
  3.0->"Here's 3.0",
  "abc"->"xyz"]
```

Tables, continued

If a non-existent key is specified, the table's *default value* is produced. The default default-value is `&null`:

```
][ t := table();
   r := T1:[]  (table)

][ t[999];
   r := &null  (null)
```

A default value may be specified as the argument to `table`:

```
][ t2 := table(0);
   r := T1:[]  (table)

][ t2["xyz"];
   r := 0  (integer)

][ t2["abc"] +:= 1;
   r := 1  (integer)

][ t2["abc"];
   r := 1  (integer)

][ t3 := table("not found");
   r := T1:[]  (table)

][ t3[50];
   r := "not found"  (string)
```

Language design issue: References to non-existent list elements fail, but references to non-existent table elements succeed and produce an object that can be assigned to. Is that good or bad?

Tables, continued

A key quantity represented with multiple types produces multiple key/value pairs.

```
][ t := table();
   r := T1:[]  (table)

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

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

][ t[1.0] := "real";
   r := "real"  (string)

][ write(Image(t));
T1:[
  1->"integer",
  1.0->"real",
  "1"->"string"]

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

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

Be wary of using `reals` as table keys. Example:

```
][ t[1.000000000000001];
   r := &null  (null)

][ t[1.0000000000000001];
   r := "real"  (string)
```

Table application: word usage counter

A simple program to count the number of occurrences of each "word" read from standard input:

```
link split, image
procedure main()
    wordcounts := table(0)

    while line := read() do
        every word := !split(line) do
            wordcounts[word] +:= 1

    write(Image(wordcounts))
end
```

Interaction:

```
% wordtab
to be or
not to be
^D
T1:[
  "be"->2,
  "not"->1,
  "or"->1,
  "to"->2]
```

Question: How could we also print the number of distinct words found in the input?

`Image` is great for debugging, but not suitable for end-user output.

Table sorting

Applying the `sort` function to a table produces a list consisting of two-element lists holding key/value pairs.

Example:

```
][ write(Image(wordcounts));
T1:[
  "be"->2,
  "not"->1,
  "or"->1,
  "to"->2]

][ write(Image(sort(wordcounts)));
L1:[
  L2:["be", 2],
  L3:["not", 1],
  L4:["or", 1],
  L5:["to", 2]]
```

`sort` takes an integer-valued second argument that defaults to 1, indicating to produce a list sorted by keys. An argument of 2 produces a list sorted by values:

```
][ write(Image(sort(wordcounts,2)));
L1:[
  L2:["not", 1],
  L3:["or", 1],
  L4:["to", 2],
  L5:["be", 2]]
```

`sort`'s second argument may also be 3 or 4, which produces "flattened" versions of the results produced with 1 or 2, respectively.

Table sorting, continued

An improved version of `wordtab` that uses `sort`:

```
link split, image
procedure main()
    wordcounts := table(0)

    while line := read() do
        every word := !split(line) do
            wordcounts[word] +:= 1

    pairs := sort(wordcounts, 2)
    every pair := !pairs do
        write(pair[1], "\t", pair[2])
end
```

Output:

```
not     1
or      1
to      2
be      2
```

Problem: Print the most frequent words first rather than last.

Tables—default value pitfall

Recall this pitfall with the `list(N, value)` function:

```
][ list(5,[]);
   r1 := L1:[L2:[],L2,L2,L2,L2]  (list)
```

There is a similar pitfall with tables:

> If `[]` is specified as the default value, all references to non-existent keys produce the **same** list.

Example:

```
][ t := table([]);
   r := T1:[]  (table)

][ put(t["x"], 1);

][ put(t["y"], 2);

][ t["x"];
   r := L1:[1,2]  (list)

][ t["y"];
   r := L1:[1,2]  (list)

][ [t["x"], t["y"]];
   r := L1:[L2:[1,2],L2]  (list)

][ [t["x"], t["y"], t["z"]];
   r := L1:[L2:[1,2],L2,L2]  (list)
```

Solution: Stay tuned!

Table application: Cross reference

Consider a program that prints a cross reference listing that shows the lines on which each word appears.

```
% xref
to be or
not to be is not
going to be
the question
^D
be.............1 2 3
going..........3
is.............2
not............2 2
or.............1
question.......4
the............4
to.............1 2 3
```

Problem: Sketch out a solution.

Cross reference solution

```
procedure main()
    refs := table()
    line_num := 0

    while line := read() do {
        line_num +:= 1
        every w := !split(line) do {
            /refs[w] := []
            put(refs[w], line_num)
            }
        }

    every pair := !sort(refs) do {
        writes(left(pair[1],15,"."))
        every writes(!pair[2]," ")
        write()
        }
end
```

Question: Are lists really needed in this solution?

Another approach:

```
procedure main()
    refs := table([])  # BE CAREFUL!
    line_num := 0

    while line := read() do {
        line_num +:= 1

        every w := !split(line) do
            refs[w] |||:= [line_num]
        }
    ...
end
```

Tables and generation

When applied to a table, `!` generates the values in the table.

Consider a table `romans` that maps roman numerals to integers:

```
][ write(Image(romans));
T1:[
  "I"->1,
  "V"->5,
  "X"->10]

][ .every !romans;
   10  (integer)
   1  (integer)
   5  (integer)
```

The `key(t)` function generates the keys in table `t`:

```
][ .every key(romans);
   "X"  (string)
   "I"  (string)
   "V"  (string)

][ .every romans[key(romans)];
   10  (integer)
   1  (integer)
   5  (integer)
```

Language design question: What is the Right Thing for `!t` to generate?

Table key types

Any type can be used as a table key.

```
][ t := table();

][ A := [];
][ B := ["b"];

][ t[A] := 10;
][ t[B] := 20;
][ t[t] := t;

][ write(Image(t));
T2:[
  L1:[]->10,
  L2:[
    "b"]->20,
  T2->T2]
```

Table lookup is identical to comparison with the === operator, using value semantics for scalar types and reference semantics for structure types.

```
][ A;
   r := L3:[]  (list)
][ t[A];
   r := 10  (integer)
][ t[[]];
   r := &null  (null)

][ get(B);
   r := "b"  (string)
][ B;
   r := L3:[]  (list)
][ t[B];
   r := 20  (integer)
```

Table application: Cyclic list counter

Consider a procedure `lists(L)` to count the number of unique lists in a potentially cyclic list:

```
][ lists([]);
   r := 1  (integer)

][ lists([[],[]]);
   r := 3  (integer)

][ A := [];
][ put(A,A);
][ put(A,[A]);
][ A;
   r := L1:[L1,L2:[L1]]  (list)

][ lists(A);
   r := 2  (integer)
```

Implementation:

```
procedure lists(L, seen)
    /seen := table()

    if \seen[L] then return 0

    count := 1
    seen[L] := 1   # any non-null value would do

    every e := !L & type(e) == "list" do
        count +:= lists(e, seen)
    return count
end
```

Problems: Write `lcopy(L)` and `lcompare(L1,L2)`, to copy and compare lists.

csets—sets of characters

Icon's cset data type is used to represent sets of characters.

In strings, the order of the characters is important, but in a cset, only membership is significant.

A cset literal is specified using apostrophes. Characters in a cset are shown in collating order:

```
][ 'abcd';
   r := 'abcd'  (cset)

][ 'bcad';
   r := 'abcd'  (cset)

][ 'babccabc';
   r := 'abc'  (cset)

][ 'babccabdbaab';
   r := 'abcd'  (cset)
```

Equality of csets is based only on membership:

```
][ 'abcd' === 'bcad' === 'bcbbbbabcd';
   r := 'abcd'  (cset)
```

(In other words, csets have value semantics.)

If c is a cset, *c produces the number of characters in the set.

For !c, the cset is converted to a string and then characters are generated.

csets, continued

Strings are freely converted to character sets and vice-versa.

The second argument for the `split` procedure is actually a character set, not a string. Because of the automatic conversion, this works:

```
split("...1..3..45,78,,9 10  ", "., ")
```

But more properly it is this:

```
split("...1..3..45,78,,9 10  ", '., ')
```

Curio: Converting a string to a cset and back sorts the characters and removes the letters.

```
][ string(cset("tim korb"));
   r := " bikmort"  (string)
```

csets, continued

A number of keywords provide handy csets:

```
][ write(&digits);
0123456789
   r := &digits  (cset)

][ write(&lcase);
abcdefghijklmnopqrstuvwxyz
   r := &lcase  (cset)

][ write(&ucase);
ABCDEFGHIJKLMNOPQRSTUVWXYZ
   r := &ucase  (cset)
```

Others:

&ascii	The 128 ASCII characters
&cset	All 256 characters in Icon's "world"
&letters	The union of &lcase and &ucase

csets, continued

The operations of union, intersection, difference, and complement (with respect to &cset) are available on csets:

```
][ 'abc' ++ 'cde';        # union
   r := 'abcde'  (cset)

][ 'abc' ** 'cde';       # intersection
   r := 'c'  (cset)

][ 'abc' -- 'cde';       # difference
   r := 'ab'  (cset)

][ *~'abc';              # complement
   r := 253  (integer)
```

Problem: Create csets representing the characters that may occur in:

(a) A real literal

(b) A Java identifier

(c) A UNIX filename

Problem: Print characters in string s1 that are not in string s2.

csets, continued

Problem: Using csets, write a program to read standard input and calculate the number of distinct characters encountered.

Problem: Print the numbers in this string (s).

```
On February 14, 1912, Arizona became the 48th
state.
```

Sets

A `set` can be created with the `set(L)` function, which accepts a list of initial values for the set:

```
][ s := set([1,2,3]);
   r := S1:[2,1,3]  (set)

][ s2 := set(["x", 1, 2, "y", 1, 2, 3, "x"]);
   r := S1:[2,"x",1,3,"y"]  (set)

][ s3 := set(split("to be or not to be"));
   r := S1:["to","or","not","be"]  (set)

][ set([[],[],[]]);
   r := S1:[L1:[],L2:[],L3:[]]  (set)

][ s4 := set();
   r8 := S1:[]  (set)
```

Values in a set are unordered. All values are unique, using the same notion of equality as the === operator.

The unary `*`, `!`, and `?` operators do what you'd expect:

```
][ *s2;
   r := 5  (integer)

][ .every !s;
   2  (integer)
   1  (integer)
   3  (integer)

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

Sets were a late addition to the language.

Sets, continued

The `insert(S, x)` function adds the value `x` to the set `S`, if not already present, and returns `S`. It always succeeds.

The `delete(S, x)` function removes the value x from S and returns S. `It always succeeds.`

The `member(S, x)` function succeeds iff `S` contains `x`.

Examples:

```
][ every insert(s,!"testing");
Failure

][ s;
   r := S1:["s","e","g","t","i","n"]  (set)

][ insert(s, "s");
   r := S1:["s","e","g","t","i","n"]  (set)

][ every delete(s, !"aieou");
Failure

][ s;
   r := S1:["s","g","t","n"]  (set)

][ member(s, "a");
Failure

][ member(s, "t");
   r := "t"  (string)
```

Sets, continued

Set union, intersection, and difference are supported:

```
][ fives := set([5,10,15,20,25]);
   r := S1:[5,10,15,20,25]  (set)

][ tens := set([10,20,30]);
   r := S1:[10,20,30]  (set)

][ fives ** tens;
   r := S1:[10,20]  (set)

][ fives ++ tens;
   r := S1:[5,10,15,20,25,30]  (set)

][ fives -- tens;
   r := S1:[5,15,25]  (set)

][ tens -- fives;
   r := S1:[30]  (set)
```

Problem: Write a program that reads an Icon program on standard input and prints the unique identifiers. Assume that `reserved()` generates a list of reserved words such as "if" and "while", which should not be printed.

Sets and tables—common functions

The `insert`, `delete`, and `member` functions can be applied to tables:

```
][ t := table();
   r := T1:[]  (table)

][ t["x"] := 10;
   r := 10  (integer)

][ insert(t, "v", 5);
   r := T1:["v"->5,"x"->10]  (table)

][ member(t, "i");
Failure

][ delete(t, "v");
   r := T1:["x"->10]  (table)
```

Note that the only way to truly delete a value from a table is with the `delete` function:

```
][ t["x"] := &null;  # the key remains...
   r := &null  (null)

][ t;
   r := T1:["x"->&null]  (table)

][ delete(t, "x");
   r := T1:[]  (table)
```

Records

Icon provides a record data type that is simply an aggregate of named fields.

A record declaration names the record and the fields. Examples:

```
record name(first, middle, last)

record point(x,y)
```

`record` declarations are global and appear at file scope.

A record is created by calling the record constructor.

```
][ p := point(3,4);
   r := R1:point_1(3,4)  (point)

][ type(p);
   r := "point"  (string)

][ p.x;
   r := 3  (integer)

][ p.y;
   r := 4  (integer)

][ p2 := point(,3);
   r := R1:point_3(&null,3)  (point)

][ type(point);
   r1 := "procedure"  (string)

][ image(point);
   r2 := "record constructor point"  (string)
```

Records, continued

A simple example:

```
record point(x,y)
record line(a, b)

procedure main()
    A := point(0,0)
    B := point(3,4)

    AB := line(A,B)
    write("Length: ", length(AB))

    move(A,-3,-4)
    write("New length: ", length(AB))
end

procedure length(ln)
    return sqrt((ln.a.x-ln.b.x)^2 +
                (ln.a.y-ln.b.y)^2)
end

procedure move(p, dx, dy)
    p.x +:= dx
    p.y +:= dy
end
```

Output:

```
Length: 5.0
New length: 10.0
```

Problem: Modify `move()` so that a new point is created, rather than modifying the referenced point.

Records, continued

A routine to produce a string representation of a point:

```
procedure ptos(p)
    return "(" || p.x || "," || p.y || ")"
end
```

Records can be meaningfully sorted with `sortf`:

```
][ pts := [point(0,1), point(2,0), point(-3,4)];

][ every write(ptos(!sortf(pts,1)));
(-3,4)
(0,1)
(2,0)
Failure

][ every write(ptos(!sortf(pts,2)));
(2,0)
(0,1)
(-3,4)
Failure
```

Fields in a record can be accessed with a subscript:

```
][ pt := point(3,4);

][ pt[2];
   r := 4  (integer)
```

String scanning basics

Icon's string scanning facility is used for analysis of strings.

The string scanning facility allows string analysis operations to be intermixed with general computation.

String scanning is initiated with `?`, the scanning operator:

```
expr1 ? expr2
```

The value of `expr1` is established as the subject of the scan (`&subject`) and the scanning position in the subject (`&pos`) is set to 1. `expr2` is then evaluated.

```
][ "testing" ? { write(&subject); write(&pos) };
testing
1
   r := 1  (integer)
```

The result of the scanning expression is the result of `expr2`.

The procedure `snap()` displays `&subject` and `&pos`:

```
][ "testing" ? snap();
&subject =  t e s t i n g
&pos =  1  |
   r := &null  (null)
```

String scanning—`move(n)`

The built-in function `move(n)` advances `&pos` by `n` and returns the substring of `&subject` between the old and new values of `&pos`. `move(n)` fails if `n` is too large.

```
][ "testing" ? {
...     snap()
...     move(1)
...     snap()
...     write(move(2))
...     snap()
...     write(move(2))
...     snap()
...     write(move(10))
...     snap()
...     };
```

```
first snap():
    &subject =  t e s t i n g
    &pos =  1  |

move(1):
    &subject =  t e s t i n g
    &pos =  2    |

write(move(2)):
    es
    &subject =  t e s t i n g
    &pos =  4        |

write(move(2)):
    ti
    &subject =  t e s t i n g
    &pos =  6            |

write(move(10)):
    &subject =  t e s t i n g
    &pos =  6            |
```

String scanning—`move(n)`, continued

`&pos` can be thought of as a scanning "cursor". `move(n)` adjusts `&pos` (the cursor) by `n`, which can be negative.

A scanning expression that iterates:

```
][ "testing" ? while move(1) do {
...          snap()
...          write(move(1))
...          };

&subject =  t e s t i n g
&pos =  2    |
e

&subject =  t e s t i n g
&pos =  4       |
t

&subject =  t e s t i n g
&pos =  6          |
n

&subject =  t e s t i n g
&pos =  8             |
Failure
```

Negative movement:

```
][ "testing" ? { move(5); snap();
...              write(move(-3)); snap()};
&subject =  t e s t i n g
&pos =  6          |
sti
&subject =  t e s t i n g
&pos =  3      |
```

String scanning—`move(n)`, continued

Example: segregation of characters in odd and even positions:

```
][ ochars := echars := "";
   r := ""  (string)

][ "12345678" ? while ochars ||:= move(1) do
...                  echars ||:= move(1);
Failure

][ ochars;
   r := "1357"  (string)

][ echars;
   r := "2468"  (string)
```

Does this work properly with an odd number of characters in the subject string? How about an empty string as the subject?

String scanning—`tab(n)`

The built-in function `tab(n)` sets `&pos` to `n` and returns the substring of `&subject` between the old and new positions. `tab(n)` fails if `n` is too large.

```
][ "a longer example" ? {
...     write(tab(4))
...     snap()
...     write(tab(7))
...     snap()
...     write(tab(10))
...     snap()
...     write(tab(0))
...     snap()
...     write(tab(12))
...     snap()
...     };
a l   (write(tab(4))
&subject =  a   l o n g e r   e x a m p l e
&pos =  4       |

ong   (write(tab(7))
&subject =  a   l o n g e r   e x a m p l e
&pos =  7           |

er    (write(tab(10))
&subject =  a   l o n g e r   e x a m p l e
&pos = 10                 |

example (write(tab(0))
&subject =  a   l o n g e r   e x a m p l e
&pos = 17                                 |

ample (write(tab(12))
&subject =  a   l o n g e r   e x a m p l e
&pos = 12                     |
```

String scanning—`move` vs. `tab`

Be sure to understand the distinction between `tab` and `move`:

Use `tab` for absolute positioning.

Use `move` for relative positioning.

Example:

```
][ &lcase ? { write(tab(3)); write(tab(3));
...           write(move(3)); write(move(3)) };
ab

cde
fgh
```

String scanning—many(cs)

The built-in function many(cs) looks for one or more occurrences of the characters in the character set cs.

many(cs) returns the position of the end of a run of one or more characters in cs, starting at &pos.

For reference:

```
  x   x   y   z   .   .   .
|   |   |   |   |   |   |   |
1   2   3   4   5   6   7   8
```

many in operation:

```
][ "xxyz..." ? many('x');
   r := 3  (integer)

][ "xxyz..." ? many('xyz');
   r := 5  (integer)

][ "xxyz..." ? many('xyz.');
   r1 := 8  (integer)

][ "xxyz..." ? { move(2); many('yz') };
   r2 := 5  (integer)
```

Note that many(cs) fails if the next character is not in cs:

```
][ "xxyz..." ? many('.');
Failure

][ "xxyz..." ? { move(1); many('yz') };
Failure
```

many(cs), continued

many is designed to work with tab—many produces an absolute position in a string and tab sets &pos, the cursor, to an absolute position.

For reference:

```
  x   x   y   z   .   .   .
|   |   |   |   |   |   |   |
1   2   3   4   5   6   7   8
```

many and tab work together:

```
][ "xxyz..." ? { p := many('xyz'); tab(p);
                                   snap() };
&subject =  x x y z . . .
&pos =  5           |

][ "xxyz..." ? { tab(many('xyz')); snap() };
&subject =  x x y z . . .
&pos =  5           |

][ "xxyz..." ? { tab(many('xyz') + 2); snap()};
&subject =  x x y z . . .
&pos =  7               |
```

Sometimes it is better to describe what is not being looked for:

```
][ "xxyz..." ? { tab(many(~'.')); snap() };
&subject =  x x y z . . .
&pos =  5           |
```

String scanning—upto(cs)

The built-in function upto(cs) generates the positions in &subject where a character in the character set cs occurs.

```
][ "bouncer" ? every write(upto('aeiou'));
2
3
6
Failure
```

A loop to print out vowels in a string:

```
][ "bouncer" ? every tab(upto('aeiou')) do
...              write(move(1));
o
u
e
Failure
```

A program to read lines and print vowels:

```
procedure main()
    while line := read() do {
        line ? every tab(upto('aeiou')) do
                    write(move(1))
        }
end
```

When should upto be used with move, rather than tab?

upto vs. many

An attempt at splitting a string into pieces:

```
][ "ab.c.xyz" ? while write(tab(upto('.'))) do
...                       move(1);
ab
c
Failure
```

A solution that works:

```
][ "ab.c.xyz" ? while write(tab(many(~'.'))) do
...                       move(1);
ab
c
xyz
Failure
```

How could we make a list of the pieces?

How could we handle many dots, e.g., "ab...c..xyz"?

How could the upto('.') approach be made to work?

String scanning—`upto(cs)`, continued

Consider a program to divide lines like this:

```
abc=1;xyz=2;pqr=xyz;
```

into pairs of names and values.

```
procedure main()
    while line := read() do {
        line ? while name := tab(upto('=')) do {
          move(1)
          value := tab(upto(';'))
          move(1)
          write("Name: ", name, ", Value: ",
            value)
          }
        write()
        }
end
```

Interaction:

```
abc=1;xyz=2;pqr=xyz;
Name: abc, Value: 1
Name: xyz, Value: 2
Name: pqr, Value: xyz

a=1;b=2
Name: a, Value: 1
Name: b, Value: 1
```

What's wrong?

How can it be fixed?

Pitfall: incomplete scope

A scanning expression with an incomplete scope can produce a baffling bug.

Consider a routine to cut a string into pieces of length `n`, and produce a list of the results:

```
procedure cut(s, n)
    L := []

    s ? while put(L, move(n)) # get next n chars
    put(L, tab(0))            # add leftover

    return L
end
```

Execution:

```
][ cut(&lcase, 10);
   r := L1:["abcdefghij","klmnopqrst",""]
```

Solution:

```
procedure cut(s, n)
    L := []
    s ? {
        while put(L, move(n))
        put(L, tab(0))
      }
    return L
end
```

Underlying mechanism: Scanning expressions can be nested. Exiting a scanning expression restores the previous values of `&pos` and `&subject`. (Initially `1` and `""`, respectively.)

Review

Review of string scanning thus far:

Scanning operator:

```
expr1 ? expr2
```

Sets `&subject` to the value of `expr1` and sets `&pos` to 1. When `expr2` terminates, the previous values of `&subject` and `&pos` are restored.

Functions for changing `&pos`:

`move(n)`	relative adjustment; string result
`tab(n)`	absolute adjustment; string result

Functions typically used in conjunction with `tab(n)`:

`many(cs)` produces position after run of characters in `cs`.

`upto(cs)` generates positions of characters in `cs`

Pitfalls:

`many(cs)` fails if the next character is not in `cs`.

Short scope on scanning expression causes unexpected restoration of prior `&subject` and `&pos` values.

String scanning examples

A procedure to compress a series of dots into a single dot:

```
procedure compress(s)
    r := ""
    s ? {
        while r ||:= tab(upto('.')+1) do
            tab(many('.'))
        r ||:= tab(0)
        }

    return r
end
```

A test program:

```
procedure main()
  while ln := (writes("String? ") & read()) do {
     write(compress(ln))
     write()
     }
end
```

Interaction:

```
String? a..test...right......here
a.test.right.here

String? ..testing.....
.testing.

String? .......
.
```

String scanning examples, continued

Problem: Write a procedure `rmchars(s, c)` that removes all characters in `c` from `s`. Example:

```
][ rmchars("a test here", 'aieou');
   r := " tst hr"  (string)

][ rmchars("a test here", &letters);
   r := "  "  (string)
```

Problem: Write a procedure `keepchars(s, c)` that returns a copy of `s` consisting of only the characters in `c`.

```
][ keepchars("(520) 577-6431", &digits);
   r := "5205776431"  (string)
```

String scanning examples, continued

Problem: Write a routine `expand(s)` that does simple run-length expansion:

```
][ expand("x3y4z");
   r := "xyyyzzzz"  (string)

][ expand("5ab0c");
   r := "aaaaab"  (string)

][ *expand("1000a1000bc");
   r := 2001  (integer)
```

Assume the input is well-formed.

String scanning examples, continued

Problem: Write a procedure `fname(path)` that accepts a UNIX path name such as `/x/y/z.c`, `../a/b/.init`, or `test_net`, and returns the file name component.

Problem: Make up a string scanning problem and solve it.

String scanning examples, continued

Problem: Write a program that reads the output of the `who` command and produces a list of users sorted by originating host.

Once upon a time, `who` output looked like this:

```
whm         pts/1        Feb 21 19:54    (mesquite.CS.Arizona.EDU)
cpilson     pts/228      Feb 21 20:30    (tuc-ts1-8.goodnet.com)
nicko       pts/62       Feb 20 07:44    (raleigh.CS.Arizona.EDU)
deepakl     pts/2        Feb 20 00:17    (italic.CS.Arizona.EDU)
ilwoo       pts/7        Feb 15 04:51    (folio.CS.Arizona.EDU)
siva        pts/135      Feb 21 21:37    (pug.CS.Arizona.EDU)
rajesh      pts/9        Feb 14 14:24    (astra.CS.Arizona.EDU)
muth        pts/8        Feb 19 09:18    (granjon.CS.Arizona.EDU)
butts       pts/111      Feb 21 20:41    (nomi)
ganote      pts/153      Feb 21 20:25    (lectura.CS.Arizona.EDU)
...
```

Desired output format:

```
rajesh          astra.CS.Arizona.EDU
ilwoo           folio.CS.Arizona.EDU
muth            granjon.CS.Arizona.EDU
deepakl         italic.CS.Arizona.EDU
ganote          lectura.CS.Arizona.EDU
whm             mesquite.CS.Arizona.EDU
butts           nomi
siva            pug.CS.Arizona.EDU
nicko           raleigh.CS.Arizona.EDU
cpilson         tuc-ts1-8.goodnet.com
```

Restriction: You can't use `sortf`.

String scanning examples, continued

`who` output format:

```
whm       pts/1        Feb 21 19:54    (mesquite.CS.Arizona.EDU)
```

A solution:

```
procedure main()
    who := open("who", "rp") # open pipe to read

    lines := []
    while line := read(who) do {
        line ? {
            user := tab(many(~' '))
            tab(many(' '))            # (A)
            term := tab(many(~' '))
            tab(many(' '))
            time := move(12)          # (B)
            tab(upto('(') + 1)
            sys := tab(upto(')'))
            }
        put(lines, sys || "\x00" ||
            left(user,15) || sys)
        }

    every line := !sort(lines) do
      line ? {
        tab(upto('\x00')+1)
        write(tab(0))
        }
end
```

Shortcut: Since `term` and `time` aren't used, lines (A) through (B) could be deleted.

String scanning—`find(s)`

The built-in function `find(s)` generates the positions in `&subject` (starting at `&pos`) where the string `s` begins.

```
][ "infringing on infinity" ? {
...   every posn := find("in") do
...     write(posn)
...   };
1
5
8
15
18
Failure
```

A fragment to print lines on standard input that contain "error" at or beyond position 10:

```
while line := read() do {
    line ? if (p := find("error")) >= 10 then
                write("Found at ", p)
           else
                write("Not found")
    }
```

Interaction:

```
1234567890
Not found
an error here
Not found
here is another error
Found at 17
error error error
Found at 13
```

String scanning—`find(s)`, continued

A different approach for the previous example:

```
while line := read() do {
    line ? if tab(10) & p := find("error") then
                write("Found at ", p)
           else
                write("Not found")
    }
```

Problem: Write a program `anyof` to print lines that contain any of the strings named as command line arguments. Example:

```
% anyof read write < (code above)
while line := read() do {
                write("Found at ", p)
                write("Not found")
```

String scanning—`find(s)`, continued

A routine to replace one text string with another:

```
procedure replace(s, from_str, to_str)
  new_str := ""

  s ? {
     while new_str ||:= tab(find(from_str)) do {
        new_str ||:= to_str
        move(*from_str)
        }
     new_str ||:= tab(0)
     }

    return new_str
end
```

Example:

```
replace("to be or not to be", "be", "eat")
```

Standalone use of `find`

`find` actually accepts four arguments:

```
find(s1, s2, i1, i2)
```

It generates the locations of `s2` between positions `i1` and `i2` where `s1` occurs. These defaults are used:

```
s2  &subject
i1  &pos if s2 defaulted, 1 otherwise
i2  0
```

Example:

```
][ every write(find("in", "infinite"));
1
4
Failure
```

Another version of the `anyof` program:

```
procedure main(args)
    while line := read() do {
            if find(!args,line) then
                write(line)
        }
end
```

String scanning—`match(s)`

The built-in function `match(s)` succeeds if the string `s` appears next.

```
][ "infinite" ? match("in");
   r := 3  (integer)

][ "infinite" ? tab(match("in"));
   r := "in"  (string)

][ "finite" ? tab(match("in"));
Failure

][ "finite" ? { move(3); tab(match("it"));
                write(tab(0)) };
e
   r := "e"  (string)
```

The expression `tab(match(s))` is very common; `=s` is a synonym for it:

```
][ "infinite" ? ="in";
   r := "in"  (string)

][ "mat" ? =(!"cmb"||"at");
   r := "mat"  (string)
```

Like `find`, `match` accepts four arguments, with defaults for the last three. It is commonly used to see if a string is a prefix of another:

```
if match("procedure"|"global", line) then ...
```

Problem: Comment stripper

Write a program that strips comments from Java source code. It should handle both forms (`//` and `/* ... */`). Ignore the potential of string literals having the sequences of interest.

String scanning—`pos(n)`

The built-in function `pos(n)` succeeds if `&pos` is equivalent to `n`. Either a right- or left-running position may be specified.

Here is a program that reads standard input and prints non-blank lines:

```
procedure main()
    while line := read() do
       line ?
         if not (tab(many(' \t')) & pos(0)) then
            write(line)
end
```

Question: Is the `pos` function really needed? Why not just compare to `&pos`?

Problem: What are two shorter solutions that don't use scanning?

String scanning—`any(cs)`

The built-in function `any(cs)` succeeds if the next character is in the character set `cs`. `&pos+1` is returned if successful.

A procedure to see if a string consists of a digit followed by a capital letter, followed by a digit:

```
procedure NCN(s)
    s ? {
        *s = 3 &
        tab(any(&digits)) &
        tab(any(&ucase)) &
        tab(any(&digits)) &
        return }
end
```

A driver:

```
while line := (writes("String? ") & read()) do
    if NCN(line) then
        write("ok")
    else
        write("not ok")
```

Interaction:

```
String? 8X1
ok
String? 9x2
not ok
String? 4F22
not ok
```

Question: How could `pos()` be used in this procedure?

Summary of string scanning functions

Functions for changing `&pos`:

`move(n)`	relative adjustment; string result
`tab(n)`	absolute adjustment; string result

Functions typically used in conjunction with `tab(n)`:

`many(cs)`	produces position after run of characters in `cs`
`upto(cs)`	generates positions of characters in `cs`
`find(s)`	generates positions of `s`
`match(s)`	produces position after `s`, if `s` is next
`any(cs)`	produces position after a character in `cs`

Other functions:

`pos(n)` tests if `&pos` is equivalent to `n`

`bal(s, cs1, cs2, cs3)`
similar to `upto(cs)`, but used for working with "balanced" strings. (Not covered; included for completeness.)

The functions `any`, `find`, `many`, `match`, and `upto` each accept four arguments, the last three of which default:

```
<fcn>(s1, s2, i1, i2)
```

Problem: `is_assign(s)`

Problem: Write a procedure `is_assign(s)` that succeeds iff `s` has the form *<identifier>=<integer>*.

```
][ is_assign("x4=10");
   r := 6  (integer)

][ is_assign("4=10");
Failure

][ is_assign("abc=10x");
Failure

][ is_assign("_123=456");
   r := 9  (integer)

][ is_assign("_123 = 456");
Failure
```

split.icn

This is the source for `split`:

```
procedure split(s, dlms, keepall)
   local w, ws, addproc, nullproc

   ws := []
   /dlms := ' \t'

   addproc := put
   if \keepall then
      otherproc := put
   else
      otherproc := 1

   if dlms := (any(dlms, s[1]) & ~dlms) then
      otherproc :=: addproc

   s ? while w := tab(many(dlms := ~dlms)) do {
      addproc(ws, w)
      otherproc :=: addproc
      }

   return ws
end
```

Two test cases:

```
"  just  a   test   right  here  "
```

```
"while w := tab(many(dlms := ~dlms)) do"
```

Backtracking with scanning

Consider this:

```
][ "scan this" ? every i := 1 to 10 do
                    write(tab(i));

s
c
a
n

t
h
i
s
Failure
```

And this:

```
][ "scan this" ? every write(tab(1 to 10));

s
sc
sca
scan
scan
scan t
scan th
scan thi
scan this
Failure
```

What's going on?

Backtracking with scanning, continued

In fact, `tab()` is a generator.

A simple approximation of `tab(n)`:

```
procedure Tab(n)
    oldpos := &pos
    &pos := n
    suspend &subject[oldpos:n]
    &pos := oldpos
end
```

Resumption of `tab` undoes any change to `&pos`.

`move(n)` is also a generator, changing `&pos`, suspending, and restoring the old value if resumed.

In essence, any `tab`'s and `move`'s in a failing expression are undone.

```
tab(upto(...)) & ="..." & move(...) &
s := tab(many(...)) & p1(...)
```

Backtracking with scanning, continued

Note the difference between bounded and unbounded `tab(...)` calls:

```
][ "abc 123" ? {
                 tab(many(&letters))
                 tab(many(&digits))
                 snap()
               };
&subject =  a b c   1 2 3
&pos =  4         |

][ "abc 123" ? {
                 tab(many(&letters)) &
                 tab(many(&digits))
                 snap()
               };
&subject =  a b c   1 2 3
&pos =  1  |
```

Two more cases:

```
][ "abc123" ? { tab(many(&letters)) &
                tab(many(&digits))
                snap() };
&subject =  a b c 1 2 3
&pos =  7               |

][ "123" ? { tab(many(&letters)) &
             tab(many(&digits))
             snap() };
&subject =  1 2 3
&pos =  1  |
```

Backtracking in scanning, continued

Here's a program that recognizes time duration specifications such as "10m" or "50s":

```
procedure main(args)
  while line := (writes("String? "),read()) do
    line ?
      if tab(many(&digits)) & move(1) == !"ms" &
                pos(0) then write("yes")
                       else write("no")
end
```

Interaction:

```
String? 10m
yes
String? 50s
yes
String? 100
no
String? 30x
no
```

Backtracking in scanning, continued

A revision that also recognizes specifications such as "10:43" or "7:18":

```
procedure main()
   while line := (writes("String? "), read()) do
       line ?
          if (Nsecs() | mmss()) & pos(0) then
              write("yes")
          else
              write("no")
end

procedure Nsecs()
    tab(many(&digits)) & move(1) == !"ms" &
    return
end

procedure mmss()
    mins := tab(many(&digits)) & =":" &
    nsecs := tab(many(&digits)) &
    *nsecs = 2 & return
end
```

Interaction:

```
String? 10m
yes
String? 9:41
yes
String? 8:100
no
String? 100x
no
```

Backtracking in scanning, continued

Imagine a program that looks for duration specifications and marks them:

```
% cat mark.1
The May 30 tests showed durations
between 75s and 2m.  Further analysis
revealed the span to be 1:14 to 2:03.
%
%
% mark < mark.1
The May 30 tests showed durations

between 75s and 2m.  Further analysis
        ^^^     ^^
revealed the span to be 1:14 to 2:03.
                        ^^^^    ^^^^
%
```

The code:

```
procedure main()
    while line := read() do {
       write(line)
       markline := repl(" ", *line)
       line ? while skip := tab(upto(&digits)) do {
         start := &pos
         ((Nsecs|mmss)() &
          len := &pos - start &
          markline[start+:len] := repl("^", len)) |
          tab(many(&digits))
          }
       write(markline)
       }
end
```

`Nsecs()` and `mmss()` are unchanged.

Backtracking in scanning, continued

Problem: Write a program that reads `Image()` output and removes the list labels.

Example:

```
% cat samples
r := L1:[1,2,3]  (list)
r := L1:[1,L2:[2],L3:[L4:[3,4]]]  (list)
r := L1:[L2:[],L2,L2,L2,L2]  (list)
%
% cleanlx < samples
r := [1,2,3]  (list)
r := [1,[2],[[3,4]]]  (list)
r := [[],L2,L2,L2,L2]  (list)
%
```

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.