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. The first implementation was in Ratfor, to facilitate porting Icon to a variety of machines. It was later reimplemented in C.

The 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".

The development of Icon was supported by about a decade of funding by the National Science Foundation.

Efficiency by virtue of limited resources

Compared to today computing resources were very limited when Icon was developed.

The FORTRAN implementation of Icon was developed on PDP-10 mainframe with perhaps 1.5 MIPS and maybe a megabyte or two of virtual address space. However, that was a timesharing system that supported users campus-wide and was quite slow at times.

The UNIX implementation of Icon was developed on a PDP-11/70 owned by the CS department. It limited programs to 64k bytes of program code and 64k bytes of data. Its speed was perhaps 1 MIP.

Due to these limits Icon's implementation was required to be small and efficient.

A little Icon by observation

```
% /home/cs372/fall06/ie
Icon Evaluator, Version 1.1, ? for help
][ 3+4
   r := 7  (integer)

][ "abc" || (3 + 4 * 5.6) || center("test", 10, "-")
   r := "abc25.4---test---"  (string)

][ types := [type(center), type(type(center)), *type("type")]
   r := L1:["procedure","string",6]  (list)

][ x := [1,["two"], 'three'] ||| [repl(&digits, 3 > 2)]
   r := L1:[1,L2:["two"],'ehrt',"01234567890123456789"]  (list)

][ x := ""; every(x ||:= !reverse(&lcase))
Failure

][ x[3:-3]
   r := "xwvutsrqponmlkjihgfed"  (string)
```

High-altitude view of Icon

Icon is a high-level, general-purpose imperative language with all the characteristics generally associated with a scripting language.

Icon has...

> Dynamic typing with automatic conversion between types in some cases.
>
> A large set of built-in types: integer, real, string, cset, file, procedure, list, table, set, record, and co-expression.
>
> A large set of operators with an emphasis on polymorphism.
>
> A unique expression evaluation mechanism that provides for expressions producing zero, one, or many results.
>
> A string analysis mechanism fully integrated with the rest of the language.
>
> A small "mental footprint".

Icon itself is not object-oriented but Unicon is.

Icon vs. Ruby: Strings

A string literal in Icon is specified by enclosing characters in double quotes. Unlike Ruby, that is the only way to create a string literal.

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

Because Icon is not object-oriented there are no string "methods". String operations are provided via operators and functions. Examples:

```
][ *s
   r := 7  (integer)

][ ?s
   r := "o"  (string)

][ reverse(s)
   r := "tikloot"  (string)

][ find("it", "test it!")
   r := 6  (integer)
```

Strings, continued

In Icon, positions in a string are <u>between</u> characters and run in both directions:

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

Several forms of subscripting are provided:

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

][ s[1+:4]
   r := "tool"  (string)

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

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

Icon has no "character" type. Single characters are represented by strings of length one.

Strings, continued

Assignment of string values does not cause sharing of data:

```
][ s1 := "Knuckles"
   r := "Knuckles"  (string)

][ s2 := s1
   r := "Knuckles"  (string)

][ s1[1:1] := "Fish "
   r := "Fish "  (string)

][ s1
   r := "Fish Knuckles"  (string)

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

In other words, strings use value semantics.

Any substring can be the target of an assignment.

Icon vs. Ruby: lists and tables

Icon's `list` type corresponds very closely to Ruby's `Array` class but there are some differences. Two examples:

Sublists can't be assigned to. Something like `L[2:4] := [1,2,3]` is not permitted in Icon.

The core language does not support comparison of lists. (In contrast, Ruby does support comparison of lists but blows up when comparing cyclic lists.)

Icon's `table` type is very similar to Ruby's `Hash` class but again there are differences. A couple of them:

Icon doesn't provide anything like `h = {"a", 1, "b", 2}` for initialization of a table.

Any value can be used as a table key in Icon but that's not the case with Ruby.

Keywords

Icon provides "keywords" to reference a number of commonly-used values. For example, `&null` represents the null value, which is the initial value of every variable. Here are examples of several keywords:

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

    ][ &date
       r := "2006/11/29"  (string)

    ][ &lcase || &ucase
       r := "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
    (string)

    ][ &input
       r := &input  (file)

    ][ &phi
       r := 1.618033988749895  (real)
```

Do keywords provide an advantage versus functions of the same names? (E.g., `date()`, `lcase()`, etc.)

Failure

A key design feature of Icon is that <u>an expression can fail to produce a result</u>. 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 due to failure 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]);  # s is still "testing"
Failure
```

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)
```

Failure, continued

What do these expressions do?

```
max := max < n

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

write(a < b)

f(a < b, x = y, 0 ~= *s)
```

How do Java exceptions compare to Icon's failure mechanism?

Failure, continued

From the mid-term:

> *Write a Ruby method `extract(s, m, n)` that extracts a portion of a string 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/`*

The solution in Icon:

```
procedure extract(s,m,n)
    return split(split(s, '/')[m], ':')[n]
end
```

It takes advantage of the propagation of failure and doesn't bother to check whether the first `split` succeeds.

A little I/O

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

```
][ 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)
```

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)
```

When end of file is reached, `read()` fails.

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 causes termination of the loop immediately above?

`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)
```

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 <u>expression</u>:

```
    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
```

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 `&` 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:

```
    r > 3 & write("r = ", r)

    while line := read() & line[1] ~== "." do write(line)
```

Here is pseudo-code for the implementation of &:

```
    Value andOp(Value expr1, Value expr2)
    {
        return expr2
    }
```

How does it work?

Procedures

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
```

Procedures, 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([1,2]);
Failure
```

Procedures—call tracing

One of Icon's debugging facilities is call tracing. Tracing is activated by setting the keyword `&trace` or the `TRACE` environment variable.

```
% cat -n sum.icn
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
```

```
% 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
%
```

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: 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: Out of gas...
```

The *result sequence* of `Gen` is `{3, 7}`.

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      :   13  | Gen()
Gen: Starting up...
    gen.icn      :    3  | Gen suspended 3
Result = 3
    gen.icn      :   14  | Gen resumed
Gen: More computing...
    gen.icn      :    6  | Gen suspended 7
Result = 7
    gen.icn      :   14  | Gen resumed
Gen: Out of gas...
    gen.icn      :    9  | Gen failed
    gen.icn      :   15  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: Out of gas...
```

Note: The preferred way to cause failure in an expression is to use the `&fail` keyword:

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

Generator basics, continued

If an expression involving a suspended generator fails the generator is resumed in hopes of it producing a value that will cause the expression to succeed.

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

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

```
Value? 3
Gen: Starting up...
Found 3
Value? 10
Gen: Starting up...
Gen: More computing...
Gen: Out of gas...
10 not found
Value? 7
Gen: Starting up...
Gen: More computing...
Found 7
```

This is an example of *goal directed evaluation* (GDE). In this case the goal is to see if the value entered by the user is one of the results of `Gen`.

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

][ write(15 < Gen() + 10)
Gen: Starting up...
Gen: More computing...
17

][ every write(repl("abc",Gen()))
Gen: Starting up...
abcabcabc
Gen: More computing...
abcabcabcabcabcabcabc
Gen: Out of gas...
Failure
```

The ability for a generator to appear in any expression is a hallmark of Icon.

The generator `to-by`

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

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

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

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

Bounded expressions

Here is one way to print the odd integers between 1 and 100:

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

This expression exhibits backtracking, just like Prolog.

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.

Bounded expressions, continued

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

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

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

Consider

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

and

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

The latter is an infinite loop!

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

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

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 write(!"abc");
a
b
c
Failure
```

The result sequence of `!"abc"` is `{"a", "b", "c"}`.

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

Speculate: What does the following program do?

```
procedure main()

    lines := []

    every push(lines, !&input)

    every write(!lines)

end
```

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.

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

Multiple generators, continued

Recall this vowel counter:

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

Here is a more concise version, using multiple generators:

```
procedure main()
    vowels := 0
    every !!&input == !"aeiouAEIOU" do
        vowels +:= 1
    write(vowels, " vowels")
end
```

Ruby: `print(STDIN.read.count("aeiouAEIOU"), " vowels\n")`

Which is better?

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
```

Problem: Generalize the program to any number of dice.

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

Alternation

The alternation <u>control structure</u> looks like an operator:

```
    expr1 | expr2
```

This creates a generator whose *result sequence* is 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 same as the Gen procedure shown earlier.

The expression

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

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

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
```

What is the result sequence of `write((3 | (7 to 11) | 13) > 10)`?

Example: Word tallying

Here is an Icon version of our word tallying example:

```
procedure main()
    counts := table(0)

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

    pairs := sort(counts, 1)
    every pair := !pairs do
        write(left(pair[1], 12) , pair[2])
end
```

Execution:

```
% echo "to be or not to be" | tally
be          2
not         1
or          1
to          2
```

Example: `picklines`

Imagine 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

```
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
```

String scanning

The SNOBOL4 programming language has a very powerful string pattern matching facility but it shares a problem with regular expressions in Ruby: you're either doing regular computation or you're matching a pattern—the operations can't be interleaved smoothly, like they can be in Prolog.

A design goal for Icon was to integrate string pattern matching with regular computation—match a little, compute a little, match a little, compute a little, etc.

The end result was a handful of *string scanning* functions that can be used in conjunction with Icon's other facilities to achieve the desired full integration of string pattern matching with regular computation.

In the end, Icon's string scanning facility turned to be a disappointment. It is small and powerful but the techniques involved are non-trivial. Too often, the first version of code using string scanning is not correct. Ditto for the second version.

The following slides provide a very brief look at Icon's string scanning facility. (About 50-60 slides are required for an in-depth study of the facility.)

String scanning, continued

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.

Here is a trivial example:

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

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

String scanning functions

There are two string scanning functions that change `&pos`—the current position in `&subject`:

`move(n)`	Move forwards or backwards by n characters. (`&pos +:= n`)
`tab(n)`	Move to position `n`. (`&pos := n`)

Both `move` and `tab` return the string between the old and new values of `&pos`.

There is a group of functions that produce positions to be used in conjunction with `tab`:

`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`
`bal(s, cs1, cs2, cs3)`	similar to `upto(cs)`, but used with "balanced" strings.

There is one other string scanning function:

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

The string scanning facility consists of only the above functions, the `?` operator, and the `&pos` and `&subject` keywords. Nothing more.

move(n)

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)
```

`upto(cs)`

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

A program to read lines and print vowels:

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

Usage:

```
% echo "just testing upto" | upto1
u
e
i
u
o
```

`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
```

Sit back and think: Is there a simpler way to perform this computation?

Example: $a^N b^N c^N$

Here is a program that recognizes lines of the form $a^N b^N c^N$:

```
procedure main()
    while writes("Line? ") & line := read() do {
        writes(line)
        line ? {
            as := tab(many('a')) &
            bs := tab(many('b')) &
            cs := tab(many('c')) &
            *as = *bs = *cs & pos(0) & write(": Yes")
            } | write(": No")
        }
end
```

Usage:

```
Line? aaabbbccc
aaabbbccc: Yes
Line? aabbc
aabbc: No
Line? abcx
abcx: No
```

Example: `listsum`

Below is a program that sums the integers in lines like this: [1,20,[30,[[40]],6,7],[]]

```
procedure main()
    while writes("List? ") & line := read() do
        line ?
            if sum := list(line) & pos(0) then
               write(line, ": sum is ", sum)
            else write(line, ": invalid")
end

procedure list()
    ="[]" & suspend 0
    ="[" & sum := values() & ="]" & suspend sum
end

procedure values()
    num := value() & ="," & sum := values() & suspend num + sum
    suspend value()
end

procedure value()
    suspend (list() | tab(many(&digits))) # doesn't handle negatives!
end
```

It is necessary to consistently use `suspend` to produce results from each of the procedures in order for backtracking to work properly.

Graphics in Icon

Facilities for graphical programming in Icon evolved in the period 1990-1994.

A philosophy of Icon is to insulate the programmer from details and place the burden on the language implementation. The graphics facilities were designed with same philosophy.

Icon's graphical facilities are built on the X Window System on UNIX machines. On Microsoft Windows platforms the facilities are built on the Windows API.

Graphics, continued

Here is a program that draws a "crosshair" of dots in a window:

```
link graphics
procedure main() # g1
    WOpen("size=300,200")

    every x := 0 to 300 by 3 do
        DrawPoint(x, 100)   # horizontal

    every y := 0 to 200 by 7 do
        DrawPoint(150, y)  # vertical

    WDone()  # wait for a "q" to be typed
end
```

Graphics, continued

Here is a program that randomly draws points.

```
link graphics

$define Height 300  # symbolic constants
$define Width 500  #  via preprocessor

procedure main() # g2
    WOpen("size=" || Width ||","||Height)

    repeat {
        DrawPoint(?Width-1, ?Height-1)
        }
end
```

Speculate: How long will it take it to black out every single point?

Larger example: target game

This program draws a circular target. If the player clicks inside the target within 800ms, the radius shrinks by 10%. If not, the radius grows by 10%.

```
procedure main() # g3
    WOpen("size="||Width||","||Height, "drawop=reverse")
    x := ?Width; y := ?Height; r := 50
    repeat {
        DrawCircle(x, y, r); hit := &null
        every 1 to 80 do { # poll for input every 10ms for 800ms
            WDelay(10)
            while *Pending() > 0 do {
                if Event()=== &lpress then {
                    if sqrt((x-&x)^2+(y-&y)^2) < r then {
                        FillCircle(x,y, r)
                        WDelay(500)
                        FillCircle(x,y,r)
                        hit := 1
                        break break
                        }
                    }
                }
            }
        DrawCircle(x,y,r)
        if \hit then r *:= .9 else r *:= 1.10
        x := ?Width; y := ?Height
        }
end
```

Smaller example: curve editor

Steve Kobes wrote this very elegant curve editor in 2003:

```
procedure main()
   WOpen("height=500", "width=700", "label=Curve Editor")
   pts := []
   repeat case Event() of {
      &lpress:
        if not(i := nearpt(&x, &y, pts)) then {
            pts |||:= [&x, &y]; draw(pts)}
      &ldrag: if \i then {
            pts[i] := &x; pts[i + 1] := &y; draw(pts)}
      !"Qq": break
   }
end
procedure draw(pts)
   EraseArea()
   DrawCurve!(pts ||| [pts[1], pts[2]])
   every i := 1 to *pts by 2 do
      FillCircle(pts[i], pts[i + 1], 3)
end
procedure nearpt(x, y, pts)
   every i := 1 to *pts by 2 do
     if abs(x - pts[i]) < 4 & abs(y - pts[i + 1]) < 4 then return i
end
```