Icon

CSC 372, Spring 2015
The University of Arizona
William H. Mitchell
whm@cs

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 (rational FORTRAN), 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 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 Ratfor 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

```
% /cs/www/classes/cs372/spring15/bin/ie -nn
Icon Evaluator, Version 1.1, ? for help
][ 3+4
  r := 7  (integer)

][ "abc" || (3 + 4.5)
  r := "abc7.5"  (string)

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

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

][ *r
  r := 9  (integer)
```

Icon by observation, continued

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

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

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

][ s * 3
Run-time error 102, numeric expected
offending value: "testing"
{"testing" * 3} from line 40 in ._ie_tmp.icn

][ repl(s,3)
   r := "testingtestingtesting"  (string)
```

Icon by observation, continued

```
][ 'testing this'
   r := ' eghinst'  (cset)

][ &digits
   r := &digits  (cset)

][ split("Thursday, 4/29/2015", &digits)
   r := L1:["Thursday, ","/","/"]  (list)

][ split("Thursday, 4/29/2015", ~&digits)
   r := L1:["4","29","2015"]  (list)

][ *(&letters ++ &digits)
   r := 62  (integer)
```

Icon by observation, continued

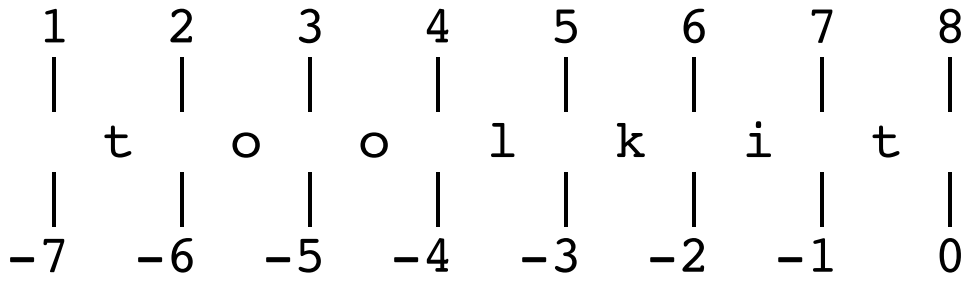
```
][ line := read()
here's some input!
   r := "here's some input!"  (string)

][ write("just",2,"test")
just2test
   r := "test"  (string)

][ x := [1, [2], "three"]
   r := L1:[1,L2:[2],"three"]  (list)

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

][ *(x ||| x)
   r := 6  (integer)
```

Icon by observation, continued

```
][ t := table("Go fish!")
   r := T1:[]  (table)

][ t["one"] := 1
   r := 1  (integer)

][ t['two'] := 2
   r := 2  (integer)

][ t
   r := T1:["one"->1,'otw'->2]  (table)

][ t["three"]
   r := "Go fish!"  (string)

][ table()[1]
   r := &null  (null)
```

String indexing

In Icon, positions in a string are between characters and run in both directions.



Several forms of subscripting are provided.

```
][ s[3:-1]
  r := "olki"  (string)

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

s[i] is a shorthand for **s[i:i+1]**

```
][ s[5]
  r := "k"  (string)
```

What problem does between-based positioning avoid?

It avoids the "to" vs. "through" problem.

Strings use "value semantics"

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)
```

Any substring can be the target of an assignment.

Failure

A key design feature of Icon is that an expression 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
```

We say, "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. And, failure propagates.

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

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

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

][ s := reverse("x" || s[50])       # s is unchanged
Failure
```

When working in Icon, unexpected failure is the root of madness.

Failure, continued

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

```
][ 3 = 0
Failure

][ 4 < 3
Failure
```

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

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

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

Failure, continued

What do these expressions do?

```
write(a < b)

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

max := max < n

max <:= 30
```

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

Failure, continued

Here's a string that represents a hierarchical data structure:

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

Major elements are delimited by slashes; minor elements are delimited by colons.

Imagine an Icon procedure to access an element given a major and minor:

```
][ extract("/a:b/apple:orange/10:2:4/xyz/", 2, 1)
  r := "apple"  (string)

][ extract("/a:b/apple:orange/10:2:4/xyz/", 3, 4)
Failure
```

Implementation:

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

How does **extract** make use of failure?

The **while** expression

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

Here's the general form of the while expression:

```
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)
```

The **while** expression

At hand:

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

If no body is needed, the **do** clause can be omitted.

Here's a more concise way to write the loop above.

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

What causes termination of this more compact version?

read() fails at end of file.

That failure propagates outward, causing the **write()** to fail.

The **while** terminates because its control expression, **write(...)**, failed.

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.

Example:

```
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?

andOp only gets called if evaluation of both operands succeeded, so all it needs to do is to return the value of the right-hand operand!

Procedures

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

Execution of an Icon program 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
```

Use **icont** to compile and run. **-s** suppresses some chatty stuff. **-x** says to execute; without it, **icont** would only produce the executable **double**.

```
% icont –s double.icn –x
```

Procedures, continued

A procedure may produce a result or it may fail. Here's a more flexible version of **double**:

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

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

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

][ double([1,2])
Failure
```

Procedures, continued

Does **double** exemplify duck typing?

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

Here is the Ruby counterpart:

```
def double x
   x * 2
end
```

Is Icon duck-challenged? If so, why?

What are tradeoffs in having different operators for addition and concatenation?

```
][ s := "abc"; n := 123
][ s || n
  r := "abc123"  (string)
```

```
>> s = "abc"; n = 123
>> s + n
TypeError: ...
>> s + n.to_s
=> "abc123"
```

Call tracing in procedures

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

```
% TRACE=-1 icont –s sum.icn -x
        :   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
%
```

```
% 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
```

Generator basics

In most languages, evaluation of an expression produces either a result or an exception.

We've seen that Icon expressions can fail, producing no result.

Some expressions in Icon are *generators*, and can produce many results.

Here's a generator:

```
1 to 3
```

`1 to 3` has the *result sequence* {1, 2, 3}.

The `.every` *directive* of `ie` can be used to show the result sequence of a generator:

```
][ .every 1 to 3
   1  (integer)
   2  (integer)
   3  (integer)
```

Generator basics, continued

Some languages allow generative constructs in particular contexts, like a "for" control structure but an Icon generator can appear at any place in any expression.

```
][ .every repl("*", 1 to 3)
  "*"  (string)
  "**"  (string)
  "***"  (string)

][ s := "abcd"
][ .every write(reverse(s[1:2 to *s]))
a
  "a"  (string)
ba
  "ba"  (string)
cba
  "cba"  (string)
```

Generator basics, continued

If an expression fails to produce a result, Icon resumes the last generator to produce a result.

```
][ i := 1 to 10 & i % 2 = 0 & write(i) & 1 = 2
2
4
6
8
10
Failure
```

Icon backtracks through previous expressions to find an active generator. If one is found, it starts evaluating the following expressions again.

What does this back and forth movement remind you of?

The above is an example of *goal-directed evaluation*.

Generator basics, continued

The **every** control structure drives a generator to failure, making it produce all its results. Example:

```
every i := 1 to 5 do
    write(repl("*", i))
```

Output:

```
*
**
***
****
*****
```

Here's a more concise version:

```
every write(repl("*", 1 to 5))
```

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
```

Note: using ***every*** *control structure*

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

For lists, ! generates the elements:

```
][ every write(![&lcase,&ucase,&digits])
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
Failure
```

"bang", continued

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
```

Execution:

```
% echo "testing" | icont -s vowels.icn -x
2 vowels
```

"bang", continued

Speculate: What does the following program do?

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

Execution:

```
% seq 3 | icont -s tac.icn -x
3
2
1
```

Alternation

The alternation control structure 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 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

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
```

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|0)])
end
```

Would it work with `line[1:21]` instead?

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.

What does **`every write(!x == !y)`** do?

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
```

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
```

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

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.)

The scanning operator

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

```
expr1 ? expr2
```

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

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**.

String scanning functions, continued

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 (including **move** and **tab**), the **?** operator, and the **&pos** and **&subject** keywords. Nothing more.

upto, many, and tab

Here's a procedure that sums the integers it finds in a string:

```
procedure sumnums(s)
   sum := 0
   s ? while tab(upto(&digits)) do
         sum +:= integer(tab(many(&digits)))
     return sum
end
```

upto(&digits) produces the position of the next digit after &pos, the current position. The wrapping tab(...) advances &pos to that position.

tab(many(&digits)) advances over the digits and returns them as a string.

```
][ sumnums("values: 10, 20 and 30")
  r := 60  (integer)
```

A goal of string scanning was to be able to interleave scanning operations with ordinary computation. Does sumnums exemplify that?

Here's a procedure that generates matches for strings of the form $a^N b^N c^N$:

```
procedure aNbNcN()
   tab(upto('a')) &
   start := &pos &
   as := tab(many('a')) &
   bs := tab(many('b')) &
   cs := tab(many('c')) &
   *as = *bs = *cs &
   suspend [start, as || bs || cs]
end
```

The &s are needed to produce procedure-wide backtracking.

A main to test with:

```
procedure main()
   while writes("Line? ") & line := read() do {
      line ? every m := aNbNcN() do
         printf("At %d: '%s'\n", m[1], m[2])
      }
end
```

```
Line? aabbcc abbc aaabbbccc ab abc
At 1: 'aabbcc'
At 13: 'aaabbbccc'
At 26: 'abc'
```

Graphics in Icon

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

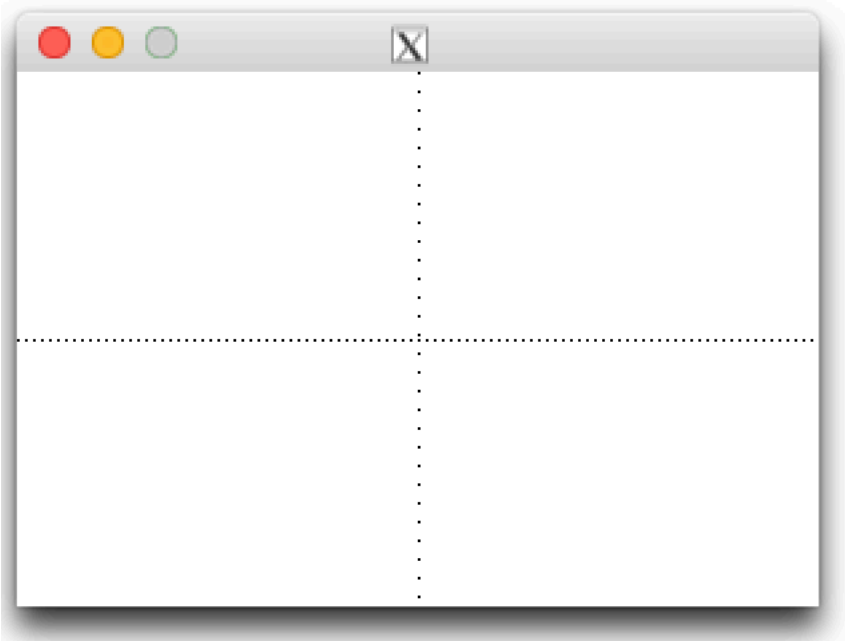
A philosophy of Icon is to insulate the programmer from details and place the burden on the language implementation. The graphics facilities were designed with same philosophy.

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.icn
   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 700    # symbolic constants
$define Width 900     #  via preprocessor

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

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

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

Simple game

```
$define Width 500
$define Height 500
procedure main() # g3.icn
   WOpen("size="||Width||","||Height, "drawop=reverse")

   x := ?Width; y := ?Height; r := 50
   repeat {
      DrawCircle(x, y, r)
      hit := &null
      every 1 to 80 do {
         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
```

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

Kobes' 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
```

Icon resources

cs.arizona.edu/icon is the Icon home page.

On the home page, under "Books About Icon", I recommend three:

The Icon Programming Language, 3rd edition

A comprehensive treatment of the language, with numerous examples of non-numerical applications.

The Implementation of the Icon Programming Language

For a time Ralph taught a course that covered the implementation of Icon's run-time system. This book rose out of that course. If you're interested in how dynamic languages are implemented, this book is definitely worth a look.

Graphics Programming in Icon

Some parts are dated but lots of interesting stuff, like Lindenmayer systems and a caricature algorithm.

unicon.org is the home page for Unicon, a derivative of Icon that supports object-oriented programming, systems programming, and programming-in-the-large.