Revised!

Icon

CSC 372, Spring 2018
The University of Arizona
William H. Mitchell
whm@cs

A little history

SL5 (SNOBOL Language 5) was developed at UA in the early 1970s.

- "SL5 had everything"
- An example of the *second-system effect*
- Never released
- Ralph thought, "There must be something simpler."

The Icon programming language

- That "simpler" thing
- Designed in mid/late 1970s by a team led by Ralph
- An example of expansion followed by contraction
- Two research focuses with Icon:
    - High level programming language facilities
    - Portable software

Icon implementation:

- First implemented in Ratfor (rational Fortran), to facilitate "porting"
- Later reimplemented in C
- Virtual-machine based

History, continued

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

- Your parents and grandparents paid for Icon.
  (Thanks!)
- Icon was placed in the public domain
  (Open source before it was cool!)
- Ralph himself mailed thousands of Icon tapes ("download"?)

Today:

- Classic Icon is alive and well

- Unicon (Unified Extended Icon) has support for object-oriented programming, systems programming, and programming-in-the-large. Clint Jeffery leads Unicon development.

- Todd Proebsting and Gregg Townsend developed Goaldi ("goal-directed") in 2015.

Design philosophy

Ralph wrote the following about SNOBOL4, but I see the same view manifested in Icon (perhaps minus the second point).

"A main philosophical view emerged in the early design efforts: *ease of use*. To us, this implied several design criteria:

1. Conciseness: the language should have a small vocabulary and express high-level operations with a minimum of verbiage.

2. Simplicity: the language should be easy to learn and use

3. Problem orientation: the language facilities should be phrased in terms of the operations needed to solve the problem, not the idiosyncrasies of computers.

4. Flexibility: users should be able to specify desired operations even if these operations are difficult to implement. [...]"

Design philosophy, continued

Continuing...

> "These objectives had several consequences, most of which can be characterized as <u>a disregard for implementation problems and efficiency</u>. This was a conscious decision and was justified by our contention that human resources were more precious than machine resources, especially in research applications where SNOBOL was expected to be used."

How much more expensive were machine resources then (the 1960s) vs. now?

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:

- about 1.5 MIPS
- maybe a megabyte or two of virtual address space
- campus-wide timesharing system

The C implementation of Icon was developed on a PDP-11/70 owned by the CS department:

- perhaps 1 MIP
- 64k bytes for program code / 64k bytes for data ("split I/D")

Due to these limits Icon's implementation was small and efficient by nature.

A little Icon by observation

```
% /cs/www/classes/cs372/spring18/bin/ie -nn
Icon Evaluator, Version 1.1, ? for help
][ 3+4
  r := 7  (integer)

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

][ "3" + "4.5"
  r := 7.5  (real)

][ 3 || 4.5
  r := "34.5"  (string)

][ 23^21
  r := 39471584120695485887249589623  (integer)

][ 17^99
  r := integer(~10^122)  (integer)
```

Icon by observation, continued

```
][ 3 || 4.5
  r := "34.5"  (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)
```

Icon by observation, continued

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

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

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

][ *(x ||| x)
  r := 8  (integer)
```

Icon by observation, continued

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

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

][ split("Monday, 4/30/18", &digits)
  r := L1:["Monday, ","/","/"]  (list)

][ split("Monday, 4/30/18", ~&digits)
  r := L1:["4","30","18"]  (list)

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

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

String indexing

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[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 <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
```

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, <u>failure propagates.</u>

```
][ 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

][ 4 > 3
  r := 3  (integer)
```

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)
```

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

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

What causes termination of this more compact version?

1. **read()** fails at end of file.
2. That failure propagates outward, causing the **write()** to fail.
3. The **while** terminates because its control expression, **write(...)**, failed.

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}`.

Two more generators:

```
    !"abc"
```

Result sequence: {`"a"`, `"b"`, `"c"`}

```
    10 | 2 | 4
```

Result sequence: {`10`, `2`, `4`}

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))
```

"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
```

Multiple generators

An expression may contain any number of generators:

```
][ every write(10 to 15 by 5, " ", "=="|"---", " ", !"ab")
10 == a
10 == b
10 --- a
10 --- b
15 == a
15 == b
15 --- a
15 --- b
Failure
```

Generators are resumed in a LIFO manner: the generator that most recently produced a result is the first one resumed.

Multiple generators, continued

Here's a program that counts vowels on standard input:

```
$ echo just testing | ./vowcount
3 vowels
```

Implementation, with multiple generators:

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

Key point:

In Icon, any expression in any context can be a generator.

Contrast:

Some languages provide "generators" but they can be only be used in certain contexts, such as a "for" statement.

String scanning

SNOBOL4 is really two languages in one:

- A general purpose programming language
- A pattern matching language

Languages with support for regular expressions are similarly divided:

- A general purpose programming language
- A regular expression facility

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:

- A handful of *string scanning* functions that can be used in conjunction with Icon's other facilities to achieve a seamless interleaving of string pattern matching with regular computation.
- Unrestricted languages ("type(0)") can be recognized with scanning.

sumnums.icn

Imagine a procedure that sums the integers it finds in a string:

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

A solution using Icon's *string scanning*:

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

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'
```

Summary of string scanning in Icon

There is a set of functions that produce positions to be 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**
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, **move(n)**, the **?** scanning operator, and the **&pos** and **&subject** keywords. Nothing more.

Disappointment

Ultimately, Icon's string scanning facility was a disappointment.

- Small and powerful
- Implementation of scanning itself is nearly trivial
- Did achieve interleaving of matching and computation
- But non-trivial techniques and idioms must be learned
- Some pitfalls
- First version often not correct (at least for me)

Is there a sweet spot with primitives and techniques?

Regular expressions:

- Lots of primitives
- Relatively few techniques

Icon's string scanning:

- Very few primitives
- Many techniques

SNOBOL4 patterns:

- A few primitives
- A few techniques

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 randomly draws points: (**blackout.icn**)

```
link graphics

$define Height 500    # symbolic constants
$define Width 500     #  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 # targetgame.icn
```

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.

`cs.arizona.edu/~whm/451` has the materials from a full-semester course I taught on Icon in 2003.

On the Icon 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 the implementation of dynamic languages, 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.

Icon takeaways

I'd like you to know:

- A three-word description of Icon: "Ruby meets Prolog"
- Icon strings are mutable, but references aren't shared.
- Icon expressions can fail and produce no result. Failure propagates.
- Icon has generators, which can produce more than one result.
- A generator can appear anywhere, not just in particular constructs.
- Icon's string scanning facility interleaves string analysis operations with regular computation.