What is Icon?

Icon is a high-level, general purpose, imperative language with a traditional appearance, but has 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".

Icon—Dead or Alive?

"I'm worried we're learning some dead language just because it was invented here at the U of A."
—Larry Johnson, CSc 372, Fall 1996

Running Icon

The traditional way of using Icon is to put an entire program into a file, translate it into a bytecode executable, and run it.

In this class we'll start by using an experimental program named `ie`, which evaluates Icon expressions interactively:

```
% /home/cs451/bin/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)
```