The Icon Language

- Icon is a **prototyping language** that traces its ancestry from Pascal and SNOBOL (a [now dead] string manipulation language).
- Icon is dynamically typed. It has generators, string manipulation functions, coroutines, structured data types (lists, tables, and sets), garbage collection, and built-in graphics support.
- Pick up implementations for Unix, Mac, PC, etc from ftp.cs.arizona.edu. The implementations seem very stable.
- With the implementation comes a huge library of useful routines and programs.
- Icon programs are usually interpreted, but there is also a compiler that translates to C.

Slide 8–1

Icon Modules

- An Icon program consists of a number of procedures declared in one or more modules. Modules are separately compiled.
- Each program must have a procedure `main` that will be called first when the program is started.

```
A.icn
procedure X()
   ...
end

procedure Y()
   ...
end
```

```
B.icn
procedure Z()
   ...
end
```

```
M.icn
link A, B
global r, t
procedure main ()
   local s, t
   X()
   Z()
end
```

Slide 8–2

Compiling Icon Programs

- Set these environment variables:
  ```
  setenv IPATH /usr/local/lib/icon/lib
  setenv LPATH /usr/local/lib/icon/include
  setenv FPATH /usr/local/lib/icon/bin
  ```

- To compile an Icon module `M.icn` do `icont -c M.icn`. This generates two files `M.u1` and `M.u2`.

- To link an Icon program (where the `main` procedure is in the module `M.icn`) do `icont M.icn`. This generates an executable file `M`.

- You can pick up additional Icon programs and functions from `/usr/local/lib/icon/lib/bipl` and `/usr/local/lib/icon/lib/gipl`.

Slide 8–3
### Procedure Declarations

There are five parts (some optional):

- The procedure heading, local declarations, initializations, static declarations, and the procedure body.
- A variable that is declared **static** "survives" (its value remains intact) between procedure invocations.
- Statements in an **initial** clause are run **the first time** the procedure is called.

```plaintext
global R, T
procedure name (arguments, extra[])  
  local x, y, z
  static a, b, c
  initial { ... }
  <statements>
end
```

---

### Types and Variables I

- Local variables don't have to be declared, but do it anyway!
- Global variables must be declared.
- A variable that has **not** been declared will automatically be treated as a local variable.
- Icon is dynamically typed. This means that
  - You don’t need to declare the types of variables.
  - A variable may contain different types of data at runtime.

```plaintext
local X
X := "hello"  # String
X := 5  # Integer
X := 6.7  # Real
```

---

### Types and Variables II

- ... You won’t get type errors at compile-time, but you will get them at run-time:
  ```plaintext
  procedure main(args)  
    t := "hello" + 4.5
  end

  Run-time error 102
  File:icon; Line 6
  numeric expected
  offending value: "hello"
  Trace back:
  main()
  {"hello" + 4.5} from line 2
  ```

---

### Types and Variables III

- **type(V)** will return the **name** (a string) of the type of V:

```plaintext
record complex(a,b)
  t := "hello"
  x := type(t)  # x="string"
  t := [5,6,7]
  x := type(t)  # x="list"
  t := complex(4,5)
  x := type(t)  # x="complex"
```

- Some data types are automatically converted to the required type. For example, a string (consisting entirely of digits) can be converted into a number, explicitly or implicitly:

```plaintext
write(5 + "6")  # implicit
write(5+integer("6"))  # explicit
```
Icon Statements I

while e1 do e2 Evaluate e2 until e1 fails.
until e1 do e2 Same as while not e1 do e2.
repeat e Evaluate the expression e repeatedly.
break Jump out of the most closely nested loop.
next Jump to the beginning of the most closely nested loop.
{...} Compound statement.
a := b Assignment. Repeated assignments (a := b := c) are also OK.

Expressions I

- There are fundamental differences in the way Pascal & Icon statements are executed:
  1. Icon statements are expressions that return values.
  2. Icon expression either succeed or fail.
- Failure doesn’t necessarily mean that something has gone wrong, rather, it means that there is no value to return.
numeric("pi") fails because "pi" cannot be converted to number.

Expressions II

Example: numeric
numeric(x) Converts x to a number.
numeric("pi") Fails.

The null value
- All Icon variables have a special null value initially.

Success & Failure

stop(s) Write s and terminate.
expr1 | expr2 Generate the values from expr1, then from expr2.
x | y Generate the variables x and y.
0 | 1 Generate the values 0 and 1.
if i = (0 | 1) then write("ok") Write "ok" if i is 0 or 1.
Expressions III

Success & Failure...

\[ \text{every } i := (0 \mid 1) \text{ do write } (i) \]

First write 0 then 1.

\[ \text{every } (x \mid y) := 0 \quad x := 0; \quad y := 0 \]

If \( p() \) fails, then stop and write "error".

\[ j := i + 10 \quad \text{This fails if } i = \text{null}. \]

\[ \text{\( x \) succeeds (and produces null) if } x = \text{null}. \text{ Fails otherwise.} \]

\[ \text{\( x \) := 0 } \quad \text{Assign 0 to } x \text{ if } x = \text{null}. \]

\[ \text{\( x \) succeeds and produces } x \text{ if } x \neq \text{null}. \text{ Fails otherwise.} \]

\[ \text{\( x := 0 \) if } x \neq \text{null} \text{ then } x := 0. \]

Generators I

- Expressions are **generators**, they can return a sequence of values.

\[ \text{\textbf{\textit{find}}(e1, e2) \text{ returns the positions within the string } e2 \text{ where the string } e1 \text{ occurs.}} \]

\[ \text{\textbf{\textit{find}}("wh", "who, what, when") \text{ has three possible solutions and hence generates three values.}} \]

\[ \text{\textbf{\textit{every}} } e1 \text{ \textbf{\textit{do}} } e2 \text{ evaluates } e2 \text{ for every value generated by } e1. \]

\[ \text{-- } 123456789012345 \]

\[ \text{\textbf{\textit{find}}("wh", "who, what, when") -- 1, 6, 12} \]

\[ \text{\textbf{\textit{every}} } i := \text{\textbf{\textit{\textit{find}}}}("wh", "who, what, when") \text{ \textbf{\textit{do write}}(i)} \]

Generators II

- Every expression has three possibilities: It can generate
  1. no values (≡ failure).
  2. one value
  3. several values

- Expression evaluation in Icon is **goal-directed**; you always try to make every expression succeed and return a value, if at all possible.

- In the example below, \textbf{\textit{find}} first returns 1. This makes \((i := \ldots) > 10\) fail. Next \textbf{\textit{find}} generates 14 which makes \((i := \ldots) > 10\) succeed, and \textbf{\textit{write}} is executed.

\[ \text{if } (i := \text{\textbf{\textit{\textbf{\textit{find}}}}}("wh", "\text{where and at what time?"}) > 10) \]
\[ \text{then write}(i) \]

Generators III

- The last expression can also be written

\[ \text{every write}(10 < \text{\textbf{\textit{\textbf{\textit{find}}}}}("wh", "where and at what time?")) \]

- Icon has many built-in generators, e.g.

\[ \text{\textbf{\textit{i to j by k}}}. \text{ The following two statements are equivalent:} \]

\[ \text{every } i := j \text{ to } k \text{ do } p(i) \]
\[ \text{every } p(j \text{ to } k) \]

- The following statement copies a file \textit{f1} row-by-row to another file \textit{f2}:

\[ \text{while write}(f2, \text{read}(f1)) \]

Slide 8–12

Slide 8–13

Slide 8–14

Slide 8–15
**Built-In Generators**

- **!s** Generates all the characters from the string, set, table, etc. S.
- **?s** Generates random elements from the set, string, table, etc. S.
- **every write(!s)** Write all the characters from the string S, one character per line.
- **upto(C, S)** Generate all the positions in the string S, where the characters in C occur. C is a special construction called a CSet, a set of characters. CSets are written in single quotes, strings in doubles.

```
12345678901234
upto ('xyz', "zebra-ox-young")
genrates {1, 8, 10}
```

Slide 8–16

**Procedures as Generators**

Procedures are really generators; they can return 0, 1, or a sequence of results. There are three cases

- **fail** The procedure fails and generates no value.
- **return e** The procedure generates one value, e.
- **suspend e** The procedure generates the value e, and makes itself ready to possibly generate more values.

```
procedure To(i, j)
while i <= j do {
  suspend i
  i+:= 1
}
end
```

Slide 8–17

**Lists I**

- Lists are a built-in Icon datatype. Lists can be accessed from the beginning (the way you would in LISP, Prolog, etc), the end, or indexed (the way you would access an array in Pascal).
- Lists can be heterogeneous, they can contain elements of different type.

```
x := ["hello", 1, 3.14, "x", "y"]  A list of a string, an integer, a float, and two strings.
y := list(5, "hej")             A list of five strings: ["hej",...,"hej"].
s := *x Number of elements of x.
x || y Concatenate x and y.
```

Slide 8–18

**Lists II**

- **put(x, 67)** Add 67 to the end of the list x.
- **get(x)** Remove and return the last element of x.
- **push(x, 1024)** Add a new element to the beginning of x.
- **pop(x)** Remove and return the first element of x.
- **x[2] := "asp"** Set the second element of x to "asp".
- **!x** Generate all elements of the list, in order.

```
every X := !L do
  write(X)
```

Slide 8–19
Tables I
Tables are associative arrays, they map keys to values. Both values and keys can be of arbitrary type.

```plaintext
x := table(0)  Create a new table x whose default value is 0. This means that if you look up a key which has no corresponding value, 0 is returned.

*x Number of elements in the table.

?q An arbitrary element from the table.

keys(x) Generate all keys in x, one at a time.

!x Generate all values, one at a time.

every X := keys(T) do
    write(X, " ==> ", T[X])
```

Slide 8–20

Tables II – Examples

```plaintext
x["monkey"] := "banana"

x[3.14] := "pi"

x["pi"] := 3.14

x["pi"] += 1  Increment pi by 1

r := x["coconut"]  r will be 0

member(x, 3.14) returns "pi"

member(x, "banana") fails

insert(x, "banana", 5)  x["banana"] := 5

delete(x, "monkey")  remove "monkey"

every m := key(x) do write(m) write keys

every m := !x do write(m) write values
```

Slide 8–21

Sets – Examples

Sets are unordered collections of elements.

```plaintext
x := set([5, 3, "monkey"])  Create a 3-element set from a list.

member(x, 5) returns 5

member(x, "banana") fails

insert(x, "banana") add "banana" to x

delete(x, 5) returns the set {3, "banana", "monkey"}

*x number of elements (3)

?q random element from x

!x generate the elements

S := S1 op S2  set union (op=++), intersection (op=***), difference (op=--).

generate S

while insert(S, read(f)) Read elements from file f into set S
```

Slide 8–22

Records

- Records are the only real declarations in Icon. They must be declared at the outermost (global) level.
- You don’t give the types of the fields, just their names.
- type(X), where X is a record variable, will return the name (a string) of the record type.

```plaintext
record complex (re, im)

procedure P ()
    local x, r, i
    x := complex(5, 4)
    r := x.re  # or r := x[1]
    i := x.im  # or r := x[2]
    t := type(x) # t="complex"
end
```

Slide 8–23
Indirect Procedure Calls

- Procedure names can be constructed at runtime, allowing a powerful form of indirect procedure call.
- Remember to include the directive `invocable all` at the beginning of your module.
- `proc(P)` returns the procedure whose name is the string `P`.

P1 := proc("MyProc1")
P2 := proc("MyProc" || "2")
P3 := proc("find") # Built-ins OK, too.
  # Multiplication has arity 2:
P4 := proc("*", 2)
  # A list of procedures:
L := [P1, P2, P3, P4]
  # Calling MyProc2(45, "X2"):
L[2](45, "X2")

Debugging Icon I

- **Bad news:** There is no Icon debugger.
- **Good news:** You don't need one!
  - When a runtime error occurs, execution terminates, and a traceback (a list of all active procedure calls) is generated:
    
    procedure Q(); x:=x"hello"; end
    procedure P(); Q(); end
    procedure main(); P(); end
    
    Run-time error 102
    File s.icn; Line 7
    numeric expected
    Trace back:
    main()
    P() from line 3 in s.icn
    Q() from line 2 in s.icn
    {&null + "hello"} from line 1

Debugging Icon II

- Since the time for an edit-compile-link is so fast, you can do your debugging using `write` statements.

  - SETENV TRACE=-1 or &trace=-1 will trace function calls.
  - `xdump` will display any variable type:

$$
\text{link ximage} \\
\text{procedure main()} \\
\text{x := table(0); x[5]:="c"} \\
\text{xdump([99, set([3,4]), x])} \\
\text{end} \\
\downarrow \\
L2 := list(3) \\
L2[1] := 99 \\
\phantom{L2[2]} \text{insert}(S1, 3) \\
\phantom{L2[2]} \text{insert}(S1, 4) \\
L2[3] := T1 := table(0) \\
\phantom{L2[3]} T1[6] := "c" \\
\text{end} \\
$$

Binary Trees in Icon I

link ximage
record node (item,left,right)
procedure Preorder (T)
if \ T then {
  write(T.item)
  Preorder(T.left)
  Preorder(T.right)}
end

procedure main()
t := node(1, 
    node(2, &null, &null), 
    node(3, 
        &null, 
        node(4, &null, &null)))
Preorder(t)
xdump(t)
end
**Binary Trees in Icon II**

```icon
> icont b
Translating:
b.icn:
    Preorder
main
> b
1
2
3
4
R_node_A := node()
    R_node_A.item := 1
R_node_A.left := R_node_1 := node()
    R_node_1.item := 2
R_node_A.right := R_node_3 := node()
    R_node_3.item := 3
    R_node_3.right := R_node_2 := node()
        R_node_2.item := 4
```

**Numeric Operations I**

- `abs(N)` absolute value
- `integer(x)` convert to integer
- `and(I1,I2)` bitwise and of two integers
- `icom(I1,I2)` bitwise complement of two integers
- `ior(I1,I2)` bitwise inclusive or of two integers
- `ishift(I1,I2)` shift I1 by I2 positions
- `ixor(I1,I2)` bitwise inclusive or of two integers
- `~N` unary negation
- `?N` random number between 1 and N

- **I1, I2,...** are integers.
- **N1, N2,...** are arbitrary numeric types.

**Numeric Operations II**

- `N1 + N2` addition
- `N1 - N2` subtraction
- `N1 * N2` multiplication
- `N1 / N2` quotient
- `N1 \% N2` remainder
- `N1 ^ N2` N1 to the power of N2
- `N1 > N2` if N1 > N2 then N2 else fail
- `N1 >= N2` if N1 ≥ N2 then N2 else fail
- `N1 < N2` if N1 < N2 then N2 else fail
- `N1 <= N2` if N1 ≤ N2 then N2 else fail
- `N1 = N2` if N1 = N2 then N2 else fail
- `N1 ~= N2` if N1 ≠ N2 then N2 else fail

**Numeric Operations III**

- `N1 op:= N1 := N1 op N2`, where op is any one of the binary operators. Examples:
  - `x +:= y := x + y`
  - `x ||:= y := x || y`
- `seq(I1,I2)` generate the integers I1, I1+I2, I1+2*I2, I1+3*I2, ...
- `I1 to I2` generate the integers between I1 and I2 in increments of I3
- `&time` elapsed time

- **&name** are built-in variables that can be read and (sometimes) modified.
### String Operations I

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char(i)</td>
<td>ASCII character number i</td>
</tr>
<tr>
<td>find(s, p, f, t)</td>
<td>positions in p[f:t] where s occurs.</td>
</tr>
<tr>
<td>map(s1, s2, s3)</td>
<td>map characters in s1 that occur in s2 into the corresponding character in s3</td>
</tr>
<tr>
<td>ord(C)</td>
<td>convert character to ASCII number</td>
</tr>
<tr>
<td>string(X)</td>
<td>convert X to a string</td>
</tr>
<tr>
<td>reverse(S)</td>
<td>return the reverse of S</td>
</tr>
<tr>
<td>type(X)</td>
<td>return the type of X as a string</td>
</tr>
<tr>
<td>*S</td>
<td>length of S</td>
</tr>
<tr>
<td>?S</td>
<td>random character selected from S</td>
</tr>
<tr>
<td>!S</td>
<td>generate characters of S in order</td>
</tr>
</tbody>
</table>

### String Operations II

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>S2</td>
</tr>
<tr>
<td>S1 &gt;&gt; S2</td>
<td>if S1 &gt; S2 then S2 else fail</td>
</tr>
<tr>
<td>S1 &gt;&gt;= S2</td>
<td>if S1 ≥ S2 then S2 else fail</td>
</tr>
<tr>
<td>S1 == S2</td>
<td>if S1 = S2 then S2 else fail</td>
</tr>
<tr>
<td>S1 &lt;&lt;= S2</td>
<td>if S1 ≤ S2 then S2 else fail</td>
</tr>
<tr>
<td>S1 &lt;&lt;= S2</td>
<td>if S1 &lt; S2 then S2 else fail</td>
</tr>
<tr>
<td>S1 ~= S2</td>
<td>if S1 ≠ S2 then S2 else fail</td>
</tr>
<tr>
<td>S[i]</td>
<td>ith character of S</td>
</tr>
<tr>
<td>S[f:t]</td>
<td>substring of S from f to t</td>
</tr>
<tr>
<td>&amp;clock</td>
<td>time of day</td>
</tr>
<tr>
<td>&amp;date</td>
<td>date</td>
</tr>
<tr>
<td>&amp;dateline</td>
<td>date and time of day</td>
</tr>
</tbody>
</table>

### Procedures and Variables

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>args(P)</td>
<td>return number of arguments of procedure P</td>
</tr>
<tr>
<td>exit(I)</td>
<td>exit program with status I</td>
</tr>
<tr>
<td>getenv(S)</td>
<td>return value of environment variable S</td>
</tr>
<tr>
<td>name(X)</td>
<td>return the name of variable X</td>
</tr>
<tr>
<td>proc(S)</td>
<td>return the procedure whose name is S</td>
</tr>
<tr>
<td>variable(S)</td>
<td>return the variable whose name is S</td>
</tr>
<tr>
<td>P!L</td>
<td>call procedure P with arguments from the list L</td>
</tr>
<tr>
<td>stop(I, X1, X2, ...)</td>
<td>exit program with error status I after writing strings X1, X2, etc.</td>
</tr>
</tbody>
</table>

### File Operations I

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>close(F)</td>
<td>close file F</td>
</tr>
<tr>
<td>open(S1, S2)</td>
<td>open and return the file whose name is S1. S2 gives the options: &quot;r&quot;=open for reading, &quot;w&quot;=open for writing, &quot;a&quot;=open for append, &quot;p&quot;=open for read &amp; write, &quot;c&quot;=create.</td>
</tr>
<tr>
<td>read(F)</td>
<td>read the next line from file F</td>
</tr>
<tr>
<td>reads(F,i)</td>
<td>read the next i characters from F</td>
</tr>
<tr>
<td>rename(S1, S2)</td>
<td>rename file S1 to S2</td>
</tr>
<tr>
<td>remove(S)</td>
<td>remove the file whose name is S</td>
</tr>
</tbody>
</table>

- F is a file variable.
### File Operations II

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>where(F)</td>
<td>return current byte position in file F</td>
</tr>
<tr>
<td>seek(F, I)</td>
<td>move to byte position I in file F</td>
</tr>
<tr>
<td>write(F, X1, X2, ...)</td>
<td>write strings X1, X2, ... (followed by a new-line character) to file F. If F is omitted, write to standard output.</td>
</tr>
<tr>
<td>writes(F, X1, X2, ...)</td>
<td>write strings X1, X2, ... to file F.</td>
</tr>
<tr>
<td>!F</td>
<td>generate the lines of F</td>
</tr>
<tr>
<td>&amp;input</td>
<td>standard input</td>
</tr>
<tr>
<td>&amp;errout</td>
<td>standard error</td>
</tr>
<tr>
<td>&amp;output</td>
<td>standard output</td>
</tr>
</tbody>
</table>

### Structure Operations I

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete(X, x)</td>
<td>delete element x from set X; delete element whose key is x from table X.</td>
</tr>
<tr>
<td>get(L)</td>
<td>delete and return the first element from the list L</td>
</tr>
<tr>
<td>pop(L)</td>
<td>delete and return the first element from the list L</td>
</tr>
<tr>
<td>pull(L)</td>
<td>delete and return the last element from the list L</td>
</tr>
<tr>
<td>push(L, X)</td>
<td>add element X to the beginning of list L and return the new list</td>
</tr>
<tr>
<td>put(L, X)</td>
<td>add element X to the end of list L and return the new list</td>
</tr>
<tr>
<td>insert(S,x)</td>
<td>insert element x into set S</td>
</tr>
<tr>
<td>insert(T,K,V)</td>
<td>insert key K with value V into table T. Same as T[K] := V.</td>
</tr>
</tbody>
</table>

### Structure Operations II

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key(T)</td>
<td>generate the keys of the elements of table T</td>
</tr>
<tr>
<td>list(I, X)</td>
<td>produce a list consisting of I copies of X</td>
</tr>
<tr>
<td>set(L)</td>
<td>return the set consisting of the elements of the list L</td>
</tr>
<tr>
<td>sort(X)</td>
<td>return the elements of the set or list X sorted in a list</td>
</tr>
<tr>
<td>sort(T,I)</td>
<td>return the elements of the table T sorted in a list L. If I=1 (sort on keys) or I=2 (sort on values), then L=[[key,val],[key,val],...]. If I=3 (sort on keys) or I=4 (sort on values), then L=[[key,val,key,val],...].</td>
</tr>
</tbody>
</table>

### Structure Operations III

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>table(X)</td>
<td>return a table with default value X.</td>
</tr>
<tr>
<td>*X</td>
<td>number of elements in X</td>
</tr>
<tr>
<td>?X</td>
<td>random element from X</td>
</tr>
<tr>
<td>!X</td>
<td>generate the elements of X (a table or set) in some random order</td>
</tr>
<tr>
<td>![X]</td>
<td>generate the elements of X (a list or record) from beginning to end</td>
</tr>
<tr>
<td>L1</td>
<td></td>
</tr>
<tr>
<td>R.f</td>
<td>field f from record R</td>
</tr>
<tr>
<td>[X1,X2,...]</td>
<td>create a list</td>
</tr>
<tr>
<td>T[X]</td>
<td>value of table T whose key is X</td>
</tr>
<tr>
<td>L[I]</td>
<td>Ith element of list L</td>
</tr>
</tbody>
</table>
Control Structures I

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>break E E</td>
<td>exit loop and return E</td>
</tr>
<tr>
<td>case E of { \ldots }</td>
<td>produce the value of the case clause whose key is E</td>
</tr>
<tr>
<td>every E1 do E2</td>
<td>evaluate E2 for every value generated by E1</td>
</tr>
<tr>
<td>fail</td>
<td>fail the current procedure call</td>
</tr>
<tr>
<td>if E1 then E2 else E3</td>
<td>produce E2 if E1 succeeds, otherwise produce E3</td>
</tr>
<tr>
<td>next</td>
<td>go to the beginning of the enclosing loop</td>
</tr>
<tr>
<td>not E</td>
<td>if E then fail else &amp;null</td>
</tr>
</tbody>
</table>

Control Structures II

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repeat E</td>
<td>evaluate E repeatedly</td>
</tr>
<tr>
<td>until E1</td>
<td>evaluate E2 until E1 succeeds</td>
</tr>
<tr>
<td>do E2</td>
<td></td>
</tr>
<tr>
<td>return E</td>
<td>return E from current procedure</td>
</tr>
<tr>
<td>while E1 do E2</td>
<td>evaluate E2 until E1 fails</td>
</tr>
<tr>
<td>E1</td>
<td>generate the results of E1 followed by the results of E2</td>
</tr>
<tr>
<td>&amp;fail</td>
<td>produces no result</td>
</tr>
<tr>
<td>&amp;null</td>
<td>null value</td>
</tr>
<tr>
<td>&amp;trace</td>
<td>if the &amp;trace is set to a value ( n &gt; 0 ), a message is produced for each procedure call/return/suspend/resume. &amp;trace is decremented for each message produced.</td>
</tr>
</tbody>
</table>

Example 1 (a): Soundex

- When names are communicated by telephone, they are often transcribed incorrectly. Soundex is a system of encoding a name that will mitigate the effects of transcription errors.

```
# Convert all occurrences of A,E,I,O, # U,W,Y in other positions to "."
# Assign the following numbers to the # remaining letters after the first:
# B,F,P,V => 1  L => 4
# C,G,J,K,Q,S,X,Z => 2  M,N => 5
# D,T => 3  R => 6

procedure soundex(name)
local first, c, i
# Convert to uppercase.
name := map(name,
    string(#1case),string(#u-case))
# Retain the first letter of the name
first := name[1]
```

Example 1 (b): Soundex

```
name := map(name,
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
    ",.123.12.22455.12623.1.2.2")

# If two or more letters with the same # code were adjacent in the original name, # omit all but the first
every c := "123456" do
    while i := find(c||c,name) do
        name[i+2] := c
    name[i] := first

# Now delete our placeholder (',')
while i := upto('',name) do name[i] := ""
return left(name,0,"0")
```

Slide 8–40

Slide 8–41

Slide 8–42

Slide 8–43
Example 1 (c): Soundex

\texttt{left(s1, i, s2)} \quad \text{shift s1 to the left,}
\text{append s2:s until position i is reached.}

\texttt{COLBERG} \quad \Rightarrow \text{(code) } "2.441.62" \Rightarrow \text{(remove duplicates) } "2.41.62" \Rightarrow \text{(restore first) } "C.41.62" \Rightarrow \text{(delete ",") } "C4162"

Example

\texttt{COLBERG} \quad \Rightarrow \text{(code) } "2.41.62" \Rightarrow \text{(remove duplicates) } "2.41.62" \Rightarrow \text{(restore first) } "C.41.62" \Rightarrow \text{(delete ",") } "C4162"

Example 2: Crypt

\texttt{procedure main(args) \hspace{1cm} local i, k, ky, l, con}
\text{if *args = 1 then}
\quad ky := get(args)
\text{else}
\quad con := open("/dev/tty", "b")
\quad writes(con, "Enter password: ")
\quad ky := read(con)
\quad close(con)
\text{end}

\begin{itemize}
\item i := 1
\item l := 0
\item k := []
\item every put(k, ord(!ky)) do
\quad l += 1
\text{while writes(char(ixor(ord(reads()), \}
\quad k[i]))) do
\quad i %:= 1 + l
\text{end}
\end{itemize}

Example 3: Pack

\# This programs reads a list of file
\# names from standard input and
\# packages the files into a single file
\# which is written to standard output.

\texttt{procedure main() \hspace{1cm} local in}
\text{while name := read() do {}
\quad close\((\text{in})\)
\quad in := open(name) |
\quad stop("cannot open input file: ",
\quad name)
\quad write("##########")
\quad write(name)
\quad while write(read(in))
\text{}}
\text{end}

Example 4 (a): Tablc

\# Tabulate characters and list each
\# character and the number of times
\# it occurs.
\# \text{-a} Write the summary in alphabetical
\# order of the characters. This is
\# the default.
\# \text{-n} Write the summary in numerical
\# order of the counts.
\# \text{-u} Write only the characters that
\# occur just once.
\text{link options}

\texttt{procedure main(args) \hspace{1cm} local count, unique, order, s, a}
\texttt{local pair, rwidth, opts}
\texttt{# switch to list unique usage only}
\texttt{unique := 0}
\texttt{# alphabetical ordering switch}
\texttt{order := 3}
Example 4 (b): Table

define options(ARGV, "au")
if ARGV[1] then order := 3
if ARGV[2] then order := 4
if ARGV[3] then unique := 1

# table of characters
charcount := table(0)
while charcount[read()] += 1
  a := sort(charcount, order)
  if unique = 1 then
    while s := get(a) do
      if get(a) = 1 then write(s)
    end
  else
    rwidth := 0
    while rwidth <= *!a
      write(left(image(s), 10),
            right(get(a), rwidth))
    end
end

Confused Student Email I–IV

Question I

Hi Dr. Collberg: Is there any expression in ICON similar to "&&" logical "AND" expression in PASCAL? Or should I just use:

If (true) then
  if (true) then
    expr1 & expr2 succeeds (and produces expr2) if both expr1 and expr2 succeed.

Question II

Will there be questions on ICON on the final exam?

No!

Question III

How do I install ICON on my [::] machine?

I have absolutely no idea.

Question IV

Is there a book on ICON?


Confused Student Email V–VI

Question V

Dear Dr. Christian:
I compile and run my program at home on my PC, transfer it to the Unix machine at the department, and then it won't run! What's wrong???

Sincerely,
Confused.

Dear Confused,
The .ui and .u2 files are text files. Be sure to transfer them so that the newline characters are properly converted. Or, transfer the .icn file and recompile.

Question VI

While doesn't this work:
  every write(f2, read(f1))
while this does:
  while write(f2, read(f1))
read is not a generator.

Confused Student Email VII

Question VII

What could cause machcode.icn to lose track of subroutines in other files? My makefile is fine, because at one moment machcode.icn is grabbing external routines correctly then it starts randomly selecting routines to reject (i.e. &null(variables).) It's even rejected YOUR

mcode := mcode_Create()

the second line of the first procedure!!! And then, without changing a single line of code above it, machcode will accept it again and pick some other external routine to complain about!

Icon doesn't have a module system. In other words, all procedures are global. This is why all (most) my procedures are prefixed by the module name. What could have happened is that you've declared a global variable or record or procedure whose name conflicts with one of my procedures, elsewhere in the compiler. So, try to name all your global procedures/variables/records with unique (i.e. long) names.

Also, make sure that you get the case right; mcode_Create() is different from mcode_create().
What were the results of this experiment? Somewhat surprisingly, it worked. We eventually had many competitors, on the order of twenty to thirty of them, but none of their software could compete with ours. We had a wysiwyg online store builder that ran on the server and yet felt like a desktop application. Our competitors had cgi scripts. And we were always far ahead of them in features. Sometimes, in desperation, competitors would try to introduce features that we didn’t have. But with Lisp our development cycle was so fast that we could sometimes duplicate a new feature within a day or two of a competitor announcing it in a press release. By the time journalists covering the press release got round to calling us, we would have the new feature too.

 [...]by word of mouth mostly, we got more and more users. By the end of 1996 we had about 70 stores online. At the end of 1997 we had 500. Six months later, when Yahoo bought us, we had 1070 users. Today, as Yahoo Store, this software continues to dominate its market. […]

I’ll begin with a shockingly controversial statement: programming languages vary in power.

Few would dispute, at least, that high level languages are more powerful than machine language. Most programmers today would agree that you do not, ordinarily, want to program in machine language. Instead, you should program in a high-level language, and have a compiler translate it into machine language for you. This idea is even built into the hardware now: since the 1980s, instruction sets have been designed for compilers rather than human programmers. […]

During the years we worked on Viaweb I read a lot of job descriptions. A new competitor seemed to emerge out of the woodwork every month or so. The first thing I would do, after checking to see if they had a live online demo, was look at their job listings. After a couple years of this I could tell which companies to worry about and which not to.

Beating the Averages

Paul Graham

(This article is based on a talk given at the Franz Developer Symposium in Cambridge, MA, on March 25, 2001.)

http://www.paulgraham.com/lib/paulgraham/sec.txt

In the summer of 1995, my friend Robert Morris and I started a startup called Viaweb. Our plan was to write software that would let end users build online stores. What was novel about this software, at the time, was that it ran on our server, using ordinary Web pages as the interface. […]

Another unusual thing about this software was that it was written primarily in a programming language called Lisp. It was one of the first big end-user applications to be written in Lisp, which up till then had been used mostly in universities and research labs. Lisp gave us a great advantage over competitors using less powerful languages.

A company that gets software written faster and better will, all other things being equal, put its competitors out of business. And when you’re starting a startup, you feel this very keenly. Startups tend to be an all or nothing proposition. You either get rich, or you get nothing. […]

Robert and I both knew Lisp well, and we couldn’t see any reason not to trust our instincts and go with Lisp. We knew that everyone else was writing their software in C++ or Perl. But we also knew that that didn’t mean anything. If you chose technology that way, you’d be running Windows. […]

So you could say that using Lisp was an experiment. Our hypothesis was that if we wrote our software in Lisp, we’d be able to get features done faster than our competitors, and also to do things in our software that they couldn’t do. And because Lisp was so high-level, we wouldn’t need a big development team, so our costs would be lower. […]
The more of an IT flavor the job descriptions had, the less dangerous the company was. The safest kind were the ones that wanted Oracle experience. You never had to worry about those. You were also safe if they said they wanted C++ or Java developers. If they wanted Perl or Python programmers, that would be a bit frightening— that's starting to sound like a company where the technical side, at least, is run by real hackers. If I had ever seen a job posting looking for Lisp hackers, I would have been really worried. [\ldots]

Exercise

```
sed 's/Lisp/Icon/g'
```