The Icon Language

- Icon is a prototyping language that traces its ancestry from Pascal and SNOBOL.
- 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.
- 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.

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
```

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`. 
**Procedure Declarations**

- A procedure has five parts: The heading, local declarations, initializations, static declarations, and the procedure body.
- A variable that is declared \textit{static survives} between procedure invocations.
- Statements in an initial clause are run \textit{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**

- Local variables don’t have to be declared, but do it anyway!
- Global variables must be declared.
- An variable that has \textit{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...**

- 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 t.icn; Line 6
numeric expected
offending value: "hello"
Trace back:
main:
  {"hello" + 4.5} from line 2

**Types and Variables...**

- \texttt{type(V)} will return the \textit{name} (a string) of the type of \texttt{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**

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.
brake 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**

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.

**Success & Failure**

i + j Succeeds and returns the value i + j.
i < j Succeeds if i < j, in which case j is returned. Fails otherwise.

**Icon Statements...**

if e1 then e2 else e3 If-expression. If e1 succeeds then evaluate and return the value of e2, otherwise evaluate and return e3. The else-part is optional.
case e of {
e1 : s1
e2 : s2
...
default : s3
}
Similar to repeated if-expression: if e1 then s1 else if e2 then s2 else... else s3. The default-part is optional. e1, e2, ... can be arbitrary expressions of arbitrary type, not just scalar constants as in Pascal.

**Expressions...**

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

All Icon variables have a special null value initially.
Expressions... 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.
every i := (0 | 1) do write (i) First write 0 then 1.

Expressions... Success & Failure...
every (x | y) := 0 x := 0; y := 0
x := p() | stop("error") If p() fails, then stop and write "error".
j := i + 10 This fails if i = null.
/x Succeeds (and produces null) if x = null. Fails otherwise.
/y Succeeds and produces x if x ≠ null. Fails otherwise.
y := 0 if x ≠ null then x := 0.

Generators

Expressions are generators, they can return a sequence of values.
find(e1, e2) returns the positions within the string e2 where the string e1 occurs.
find("wh", "who, what, when") has three possible solutions and hence generates three values.
every e1 do e2 evaluates e2 for every value generated by e1.

-- 123456789012345
find("wh","who, what, when") -- 1, 6, 12

every i:=find("wh","who, what, when") do write(i)

Generators... 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, find first returns 1. This makes ((i :=...) > 10) fail. Next find generates 14 which makes ((i :=...) > 10) succeed, and write is executed.

if (i := find("wh", "where and at what time?")) write(i)
Generators...

- The last expression can also be written

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

- Icon has many built-in generators, e.g. \(i \text{ to } j \text{ by } k\). 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 \(f1\) row-by-row to another file \(f2\):

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

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. \(S\).

\[
\text{every write}(!S) \text{ Write all the characters from the string } S, \text{ one character per line.}
\]

\[
\text{upto}(C, S) \text{ Generate all the positions in the string } S, \text{ where the characters in } C \text{ occur. } C \text{ is a special construction called a } C\text{Set, a set of characters. } C\text{Sets are written in single quotes, strings in doubles.}
\]

- \(12345678901234\)

- \(\text{upto ('xyz', "zebra-ox-young") generates } \{1, 8, 10\}\)

Procedures as Generators

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

- \(\text{fail}\) The procedure fails and generates no value.

- \(\text{return } e\) The procedure generates one value, \(e\).

- \(\text{suspend } e\) The procedure generates the value \(e\), and makes itself ready to possibly generate more values.

\[
\text{procedure To}(i,j) \\
\text{ while } i <= j \text{ do } \{ \\
\text{ suspend } i \\
\text{ i+:= 1 }
\}
\]

Lists

- 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.

\[
x := ["hello",1,3.14,"x","y"] \text{ A list of a string, an integer, a float, and two strings.}
\]

\[
y := \text{list}(5, "hej") \text{ A list of five strings: } ["hej",...,"hej"].
\]

\[
x[2:4] \text{ The list consisting of the second, third, and fourth element of } x.
\]
Lists...

s := *x Number of elements of x.
x ||| y Concatenate x and y.
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)


Tables

Tables are associative arrays, they map keys to values. Both values and keys can be of arbitrary type.
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.
?x 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])

Tables II – Examples

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

Sets

Sets are unordered collections of elements.
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"}
**Sets...**

*\(x\) number of elements (3)

?\(x\) random element from \(x\)

!\(x\) generate the elements

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

while insert \((S, \ \text{read}(f))\) Read elements from file \(f\) into set \(S\)

**Records**

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

```icon
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
```

**Indirect Procedure Calls**

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

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

**Debugging Icon**

- **Bad news:** There is no Icon debugger. **Good news:** You don’t need one!
- Since the time for an edit-compile-link is so fast, you can do your debugging using \(\text{write}\) statements.
- \(\text{SETENV TRACE=-1 or &trace:=-1}\) will trace function calls.
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

xdump will display any variable type:

```
link ximage
procedure main()
  x := table(0); x[5]:="c"
  xdump([99,set([3,4]),x])
end
```

```
L2 := list(3)
  insert(S1,3)
  insert(S1,4)
L2[3] := T1 := table(0)
  T1[5] := "c"
```

```
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
```

```
> icont b
> b
1
2
3
4
R_node_4 := node()
  R_node_4.item := 1
  R_node_4.left := R_node_1 := node()
    R_node_1.item := 2
  R_node_4.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

abs(N)  absolute value
integer(x)  convert to integer
iand(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.

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...  

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

&name are built-in variables that can be read and (sometimes) modified.

String Operations

char(i)  ASCII character number i
find(s, p, f, t)  positions in p[f:t] where s occurs.
map(s1, s2, s3)  map characters in s1 that occur in s2 into the corresponding character in s3
ord(C)  convert character to ASCII number
string(X)  convert X to a string
reverse(S)  return the reverse of S
### String Operations

- `type(X)` return the type of `X` as a string
- `*S` length of `S`
- `?S` random character selected from `S`
- `!S` generate characters of `S` in order
- `S1 || S2` string concatenation
- `S1 >> S2` if `S1 > S2` then `S2` else fail
- `S1 >>= S2` if `S1 = S2` then `S2` else fail
- `S1 <= S2` if `S1 <= S2` then `S2` else fail
- `S1 < S2` if `S1 < S2` then `S2` else fail
- `S1 ~== S2` if `S1 ≠ S2` then `S2` else fail
- `S[i]` `i`th character of `S`
- `S[f:t]` substring of `S` from `f` to `t`
- `&clock` time of day
- `&date` date
- `&dateline` date and time of day

### Procedures and Variables

- `args(P)` return number of arguments of procedure `P`
- `exit(I)` exit program with status `I`
- `getenv(S)` return value of environment variable `S`
- `name(X)` return the name of variable `X`
- `proc(S)` return the procedure whose name is `S`
- `variable(S)` return the variable whose name is `S`
- `P!L` call procedure `P` with arguments from the list `L`
- `stop(I,X1,X2,...)` exit program with error status `I` after writing strings `X1, X2, etc.`

### File Operations

- `close(F)` close file `F`
- `open(S1, S2)` open and return the file whose name is `S1`. `S2` gives the options: "r"=open for reading, "w"=open for writing, "a"=open for append, "b"=open for read & write, "c"=create.
- `read(F)` read the next line from file `F`
- `reads(F,i)` read the next `i` characters from `F`
- `rename(S1,S2)` rename file `S1` to `S2`
- `remove(S)` remove the file whose name is `S`

* `F` is a file variable.
**File Operations...**

where(F) return current byte position in file F

seek(F, I) move to byte position I in file F

write(F, X1, X2, ...) write strings X1, X2, ...(followed by a newline character) to file F. If F is omitted, write to standard output.

writes(F, X1, X2, ...) write strings X1, X2, ...to file F.

!F generate the lines of F

&input standard input

&errout standard error

&output standard output

---

**Structure Operations**

delete(X, x) delete element x from set X; delete element whose key is x from table X.

get(L) delete and return the first element from the list L

pop(L) delete and return the first element from the list L

pull(L) delete and return the last element from the list L

push(L, X) add element X to the beginning of list L and return the new list

---

**Structure Operations...**

put(L, X) add element X to the end of list L and return the new list

insert(S, x) insert element x into set S

insert(T,K,V) insert key K with value V into table T. Same as T[K] := V.

key(T) generate the keys of the elements of table T

list(I, X) produce a list consisting of I copies of X

set(L) return the set consisting of the elements of the list L

---

**Structure Operations...**

sort(X) return the elements of the set or list X sorted in a list

sort(T,I) 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,...].

table(X) return a table with default value X.
**Structure Operations...**

*\(X\)  
number of elements in \(X\)

?\(X\)  
random element from \(X\)

!\(X\)  
generate the elements of \(X\) (a table or set) in some random order

!\(X\)  
generate the elements of \(X\) (a list or record) from beginning to end

L1 ||| L2  
concatenate lists

R.f  
field \(f\) from record \(R\)

[X1, X2, ...]  
create a list

T[X]  
value of table \(T\) whose key is \(X\)

L[I]  
\(i^{th}\) element of list \(L\)

---

**Control Structures**

break \(E\)  
extit loop and return \(E\)

case \(E\) of \{  
produce the value of the case clause whose key is \(E\)

...\}  

every  
evaluate \(E2\) for every value generated by \(E1\)

E1 do  
E2  
produce \(E2\) if \(E1\) succeeds, otherwise produce \(E3\)

e else  
\(E3\)

next  
go to the beginning of the enclosing loop

not \(E\)  
if \(E\) then fail else \&null

---

**Control Structures...**

repeat  
evaluate \(E\) repeatedly

E  

until  
evaluate \(E2\) until \(E1\) succeeds

E1 do  
E2  

return  
return \(E\) from current procedure

E  

while  
evaluate \(E2\) until \(E1\) fails

E1 do  
E2  

E1 | E2  
generate the results of \(E1\) followed by the results of \(E2\)

---

\&fail  
produces no result

\&null  
null value

\&trace  
if the \&trace is set to a value \(n > 0\), a message is produced for each procedure call/return/suspend/resume.
String Scanning

- The expression $s \ ? \ e$ makes $s$ the subject to which string processing operations in $e$ apply.
- The program below prints 3, 13, and 23:

```plaintext
line := "a fish is a fish is a fish"
every line ? write(find("fish"))
```

String Scanning

- `move(i)` advances the position by `i` characters.
- `move` returns the substring of the subject that is matched as a result of changing the position.
- The program below sets $t$ to a string containing the characters of `line` followed by periods:

```plaintext
t := ""
line ? while t := t || move(1) || "."
```

String Scanning

- `tab(i)` moves to position $i$ in the subject and returns the substring between the old and new positions.
- `upto(s)` returns the position of any of the characters in $s$.
- `many(s)` returns the position following the longest possible substring containing only characters in $s$ starting at the current position.

```plaintext
many (&\{\text{letters}\}, "2857435") # fails
many (&\{\text{letters}\}, "abc43543") # succeeds and # returns 4
```

String Scanning

- `any(c)` succeeds if the first character in the subject string is in the cset $c$.
- `match(t)` succeeds if $t$ matches the initial characters of the subject string and returns the position after the matched part.

```plaintext
match("foo", "frukost") # fails
match("foo", "foosball") # succeeds and # returns 4
```
procedure getword(str)
    str ? while tab(upto(&letters)) do {
        word := tab(many(&letters))
        suspend word
    }
end

&letters contains all upper- and lowercase letters.

tab(upto(&letters)) advances the position up to the next letter.

tab(many(&letters)) matches the word and assigns it to word.

The while terminates when tab(upto(&letters)) fails because there are no more words in str.

A cset is a basic Icon type that describes sets of characters.

Csets are written as a string of characters between single quotes.

Predefined csets:
   &digits: digits between 0 to 9.
   &letters: all letters.
   &lcase: lower case letters.
   &ucase: upper case letters.

The normal set operations can be performed using ++ (union), ** (intersection), -- (set difference), and ~ (complement).

The program below lists the most commonly used words in its input and their frequencies of occurrence.

procedure main(args)
    k := integer(args[1]) | 10
    words := table(0)
    while line := read() do
        every words[getword(line)] += 1
    end
    words := sort(words, 4)
    every 1 to k do
        write(pull(words), "\n", pull(words))
end

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,H,I,O,
# U,W,Y in other positions to "."
# Assign the following numbers to the
# remaining letters after the first:
Example 1 (a): Soundex

# B,F,P,V => 1
# C,G,J,K,Q,S,X,Z => 2
# D,T => 3
# L => 4
# M,N => 5
# R => 6

procedure soundex(name)
  local first, c, i
  # Convert to uppercase.
  name := map(name, string(&lcase), string(&ucase))
  # Retain the first letter of the name
  first := name[1]
  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[1] := first
  # Now delete our place holder ('.')
  while i := upto('.', name) do name[i] := ""
  return left(name, 4, "0")
end

left(s1, i, s2) shift s1 to the left, append s2:s until position i is reached.

Example

COLBERG \( \Rightarrow \) (code) "2.441.62" \( \Rightarrow \) (remove duplicates) "2.41.62" \( \Rightarrow \) (restore first) "C.41.62" \( \Rightarrow \) (delete ".") "C4162"

Example 2: Crypt

procedure main(args)
  if *args = 1 then
    ky := get(args)
  else {con := open("/dev/tty", "b")
    writes(con, "Enter password: ")
    ky := read(con)
    close(con)
  }
  i := 1; l := 0; k := []
  every put(k, ord(!ky)) do l += 1
  while writes(char(ixor(ord(reads()), k[i]))) do
    i %:= l + 1
end
Example 3: Pack

# This program reads a list of file names from
# standard input and packages the files into a
# single file which is written to standard output.
procedure main()
    while name := read() do {
        close(in)
        in := open(name) |
        stop("cannot open input file: ", name)
        write("##########")
        write(name)
        while write(read(in))
    }
end

Example 4: Tablec

# Tabulate characters and list each character and
# the number of times it occurs.
# -a Write the summary in alphabetical order of
#   the characters. This is the default.
# -n Write the summary in numerical order
# -u Write the characters that occur just once.
link options
procedure main(args)
    local ccount, unique, order, s, a
    unique := 0 # switch to list unique usage only
    order := 3 # alphabetical ordering switch
    ccount := table(0) # table of characters
    while ccount[reads()] += 1
    a := sort(ccount,order)
    if unique = 1 then
        while s := get(a) do if get(a) = 1 then write(s)
        else {
            rwidth := 0; every rwidth <:= *!a
            while s := get(a) do
                write(left(image(s),10),right(get(a),rwidth))
        }
    end

Confused Student Email

Question 1

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.
Confused Student Email...

**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?*

**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 .u1 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*

```plaintext
while write(f2, read(f1))
```

*while this does:*

```plaintext
while write(f2, read(f1))
```

*read is not a generator.*

*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!*
Confused Student Email...

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().

Readings and References

- http://www.cs.arizona.edu/icon/
- http://dmoz.org/Computers/Programming/Languages/Icon
- The string-scanning examples were taken from http://www.cs.arizona.edu/icon/intro.htm and http://www.nmt.edu/tcc/help/lang/icon.

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.

Beating the Averages

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. [⋯] 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.

[⋯] 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. [⋯]

```
Exercise

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