A STUDY OF THE ICON PROGRAMMING LANGUAGE

by

Lopamudra Sarangi lopa@cs.arizona.edu
Parag Sarfare parags@cs.arizona.edu

1. Introduction

Icon is a high level general-purpose programming language with extensive features such as string processing, advanced data and control structures and built in operators. It may be classified under the procedural and imperative paradigms of programming languages. Icon has been widely used for scripting and rapid development tools owing to its strong string manipulation abilities and lucid, compact and yet highly expressive program structure.

2. History of Icon

Icon was developed at the University of Arizona in the late 1970s by Dr. Ralph Griswold and team. It has been influenced by SNOBOL4 and SL5. It supports all major operating system platforms and some lesser known ones such as Acorn Archimedes and VAX/VMS. The Icon project is still ongoing and there have been some interesting extensions to the language such as:

- Graphics features - added during 1990-1994.
- Unicon (Unified Extended Icon) 1997 -1999 has support for object-oriented programming, systems programming, and programming-in-the-large.
- JIcon - A Java based implementation of Icon.
- MT Icon- Multitasking Icon.

3. Applications

Icon was most appropriately used to develop applications which require lesser development time and programming effort. It is especially strong at building software tools, for processing text, and for experimental and research applications where short effective programs are required. Programming tasks that require extensive manipulations of strings and structures such as compilers, word processors, and databases made extensive use of the language. Not only has it been useful for short, single use programs but also in very complicated applications that involve complex data structures. Some of the areas where Icon has been effectively used include artificial intelligence, natural language processing, research applications, expert systems, rapid prototyping, program generation, symbolic mathematics, graphic displays and text processing (for analysis, editing, document formatting and generation).

4. Language Features

Icon has some distinguishing features in terms of control constructs and data structures which make it a powerful language. Some of Icon's prominent features are discussed below.

4.1 Expression Evaluation

Icon differs significantly from other programming languages when it comes to expressions. It has an expression-oriented syntax and the expression evaluation mechanism is unique as it integrates **goal-directed evaluation** and **control backtracking**. An expression in Icon can have one of the following outcomes.
1. Succeed producing a result (Eg: the expression ***4 > 3*** succeeds and the result is 3)
2. Fail producing a null value (Eg: the expression ***1 = 0*** fails)

An operation in Icon is performed only if evaluation of all operands succeed. Goal-directed evaluation refers to this automatic searching among alternative results. For example:

min < (x | 5) first compares min to x. If min is less than x, evaluation succeeds and produces the value x. If not, min is compared to 5, the next value generated by the subexpression x | 5. If min is less than 5, evaluation succeeds and produces 5. Otherwise, evaluation fails, and no value is produced. Thus failure drives control expressions in Icon. This allows the concise formulation of many programming tasks by handling errors implicitly. For example the following code

```
line := read() & write(line)
```

can read consecutive lines from an input file until the end of file and write them to the standard output. However, the control backtracking in Icon is limited to the expression in which it occurs. For example in:

```
max := max < x
write("y=", (x | 5) > y)
```

failure in the second expression does not affect the outcome of the first. Another example of goal-directed evaluation is

```
find("on", line1) = find("and", line2)
```

which succeeds if "or" occurs in line1 at the same position as "and" occurs in line2.

4.2 Generators

In Icon generators are used to iterate through multiple outcomes of expressions which produce more than one result. They are similar to iterator-based control structures found in other languages but different in many aspects. The following language constructs can construct a generator:
- expression that can generate a sequence of values
- *every* expression
- to and to-by operator (generates the integers from one value to another)
- element generation operator ! (***!x***, generates all values of x[i] for 1 to number of elements in x)
- repeated alternation (|1, generates a integer 1, infinite times, expr1 | expr2 generates the results of expr1 followed by the results of expr2.
- *seq(..)* generation (seq(i) - generates i,i+1,i+2,...)

For example:

line1 := "Python is like Icon"
find("on", line1)
Here "on" occurs in line1 at positions 5 and 18 and both are successful outcomes of the expression. Generation is inherited like failure, and thus the following expression.

every write(find("on", line1)) produces the same result as above.
The results that a generator produces depend on context. In a situation where only one result is needed, the first is produced. For example:

i := find("on", line1) assigns the value 5 to i.

If the result produced by a generator does not lead to the success of an enclosing expression, however, the generator is resumed to produce another value as illustrated in the previous section. For example:

if (i := find("on", line1)) > 15 then write(i)
Here the first result produced by the generator, 5, is assigned to i, but this value is not greater than 15 and the comparison operation fails. At this point, the generator is resumed and produces the second position, 18, which is greater than 15. The comparison operation then succeeds and the value 18 is written. Another example of a generator is

every write(find("on", line1 | line2))
which writes the positions of "on" in line1 followed by the positions of "on" in line2. The alternation operator
(i | j | k) = (0 | 1)
succeeds if any of i, j, or k has the value 0 or 1.

4.3 String Operations
Text processing was one of the main design goals of Icon and hence the language has a large repertoire of operations for string analysis. Strings in Icon are true first-class objects, not arrays of characters. The string scanning operation has the form

s ? expr
where s is the subject string to be examined and expr is an expression that performs the examination. A cursor is set at the first position of the string (before the 1st character). The function, move(i), moves the position of the cursor by i and produces the substring of the subject between the previous and new positions. For example,

line ? while write(move(2))
writes successive two-character substrings of line, stopping when there are no more characters. Functions such as tab(), find(), upto(), many() etc can be used in string scanning as well. For example,

line ? while write(tab(find("or")))
do move(2)
writes all the substrings of line prior to occurrences of "or".

Another example

line ? while tab(upto(&letters)) do
write(tab(many(&letters)))

writes all the words in the file. The expression tab(upto(&letters)) advances the position up to the next letter, and 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 line. Apart from these there are string editing and conversion operations which make string processing easy. For example,

s:=map("fg/ij/cd","cdefghij",&date)

converts the string date from one format to the other. Icon also provides support for regular expressions in order to enable pattern matching.

4.4 Structures

Structures in Icon are first class data values and can be heterogeneous; that is, the same structure can contain values of different types. String-processing tasks require structures to organize data-- lists of strings, symbol tables, and so on. Icon provides four kinds of built-in structures: records, lists, sets, and tables.

a) Records are the basic structures in Icon and are similar to the “struct” construct of C and other languages.
b) Lists are sequences of values of arbitrary types.

L := list(i, x) creates a list of i values, each of which has the value x while
list1 = ["language", "icon", 78, 25]

creates list1 with four values, two of which are strings and two of which are integers. A member of a list is referenced by its position.Thus,

list1[3] := 78 and list1[8] fails because it is out of range.

The values in a list L are generated by !L. Thus

every write(!L) writes all the values in L.

Lists can be manipulated like stacks, vectors and queues. The standard push, pop, enqueue and dequeue operations could be done on these data structures.

c) Sets are collections of distinct values. An empty set is created by set(). Alternatively, set(L) produces a set with the values in the list L. For example,

S := set([1, "icon", []])

assigns to S a set that contains the integer 1, the string "abc", and an empty list.

The operations union, intersection, and difference can be performed on sets. The function member(S,x) succeeds if x is a member of the set S but fails otherwise. The function ***insert(S, x)*** adds x to the set S, while ***delete(S, x)*** removes x from S. !S generates the members of S.

d) Tables are sets of key-value pairs and provide a form of associative lookup. The key and its corresponding value may be of any type, and the value for any key can be looked up automatically. An empty table is created by the expression

cars := table(0) which assigns to cars a table with the default value 0.

The default value is used for keys that are not assigned another value. Subsequently, symbols can be referenced by any key, such as
cars["honda"] := 1 which assigns the value 1 to the key "honda" in cars.
Tables grow automatically as new keys are added. The function ***sort(T, i)*** generates a list by taking the ith values from the table T.

4.5 Procedures

An Icon program consists of one or more procedures consisting of expressions separated by newlines or semicolons. Control flow starts from the main procedure. A procedure may take arguments and may return a value of interest. A procedure declaration has the form

```
procedure name (parameter-list)
  local-declarations
  initial-clause
  procedure-body
end
```

The parameter list is optional and consists of identifiers separated by commas:
A procedure in Icon can either

- succeed producing a valid result
- fail returning null value or
- suspend giving control to another procedure

The return and fail expressions cause return from a procedure call and destruction of all dynamic local identifiers for that call while suspend leaves the call in suspension with the values of dynamic local variables intact. The procedure call can be resumed to continue evaluation. For example

```
procedure main()
      every write(Invert(-3, 3))
      every write(Invert(-5))
end

procedure Invert(i, j)
/j := i
while i <= j do {
       suspend -i
       i +:= 1
       }
       fail
end
```

has the main procedure calling Invert() twice. The Invert procedure has a loop which can be suspended. At the end of the loop, the fail keyword is used to return null value from the procedure. The output of this program is 3 2 1 0 -1 -2 -3 5.

4.6 Co-Expressions

A co-expression in Icon is a data object which captures the state of an expression. It is a reference to an expression and can be used to suspend and resume an expression as and when necessary. It gives a finer granularity to the control flow in an Icon program. A co-expression is created by

co-expr_name := create expr

Co-expressions can be used for the generation of results from generators in parallel. For example,

```
procedure parallel(a1, a2)
  local x
  repeat {
    if x := @a1 then suspend x else fail
    if x := @a2 then suspend x else fail
    }
end
```

where a1 and a2 are co-expressions. The results for

parallel(create !&lcase, create !&ucase)

are "a", "A", "b", "B", ... "z", and "Z". parallel(a1, a2) terminates when either a1 or a2 runs out of results.

4.7 Data Types

Icon is a weak dynamically typed language. Types are associated with values themselves and not the variables thereby saving the programmer from the burden of type declarations. Any variable may be assigned a value of any type and later assigned a value of a different type either implicitly (depending on the operation at runtime) or explicitly by using conversion functions. However there are some conversions which Icon does not allow. Built-in data types include numerics(integers,reals,floating points), character sets, strings, sets, lists, associative tables, records, and procedures.

5. Icon: A Programming Language Perspective

In this section we look at Icon from a programming language persperctive i.e. we discuss Icon in the light of the “properties” of a programming language that we have learnt in class.

5.1 Procedures

Icon does not have any block structure, and hence nested procedures are not permitted. Functions and procedures are essentially similar in Icon. Functions are procedures that are provided by the Icon libraries. Both procedures and functions are first-class objects that can be assigned to variables, may be passed as arguments in procedure calls, and so on. Variables in the parameter list are usually called-by-value. Each formal parameter is assigned the value of the corresponding actual parameter. Since mutable objects are accessed using a pointer, the pointer is copied to the formal parameter and both the formal and the actual parameters point to the same object, which resembles call-by-reference. Lists for example are mutable objects while strings are immutable. If there are fewer actual parameters than formal, then the extra formal parameters are assigned the default value *&null*. Procedures in Icon are essentially generators too. They can return a value, no value or a sequence of values, using fail, return and suspend keywords correspondingly.

5.2 Preprocessing

Icon programs are preprocessed before they are compiled or interpreted. During preprocessing, constants can be defined, other files may be included and regions of code can be included or excluded, depending on the definition of constants. Preprocessor directives are indicated by a $ at the beginning of a line, as in

$define PI 3.14

which defines the symbol PI and gives it the value 3.14. Subsequently, whenever PI appears in the source program, it is replaced by 3.14 prior to compilation. $define directive is thus basically a macro definition and is expanded before any syntactic or semantic analysis of the source is done. $include is another directive that may be used to link modules prior to compile time, however all the modules linked together exist in the same namespace. So the programmer has to be careful when linking modules so as to not cause any identifier name conflicts.

5.3 Coroutines

Coroutines can be realised in Icon using the co-expressions feature. A co-expression can transfer control to another co-expression using language contructs (using create and then call using @co-expression_name) or returning control implicitly by producing a result. This kind of switch of control changes the location of program execution and the environment in which the subjected expressions are evaluated. The difference here is that, transfer of control is not hierarchial like in procedure calls as it can switched to any desired location based on what co-expression is called using @.

5.4 Scope Rules

Variables in Icon can be of *local*, *static* or *global* types. Scope determines the identifiers a procedure can access. Icon uses lexical scoping. Locals are created when a procedure is entered and deleted when it returns. The names of these identifiers are known only within the procedure. If an identifier is used without declaration it is assumed to be local. Static variables are created when the program starts executing. The values of static variables are retained across procedure calls. Their scope is within the procedure where they are declared. Globals are also created when the execution of the program starts. These names are visible only inside all the procedures that do not declare the same names for local or static variables. Every global variable has a single copy.

There are some other keywords, whose scope is global, and need to be declared outside any procedure. One such keyword is *record(..,..,..)* which declares a structure with sub-fields given in the declaration. The other is *link name*, which is used to indicate the linker to use procedures, records or global variables declared in the file named *name*.

5.5 Binding

Icon has dynamic binding for variable names. In Icon variable names are not bound to types. Objects are bound to types. Names can be used to point to any type of object at point in the source program and perform related operations on them. As long as the objects that form the operands in an operation are of the required Icon run-time system permits it. Link time binding occurs when various libraries are joined together by the linker when instructed using the *link* keyword.

5.6 Polymorphism

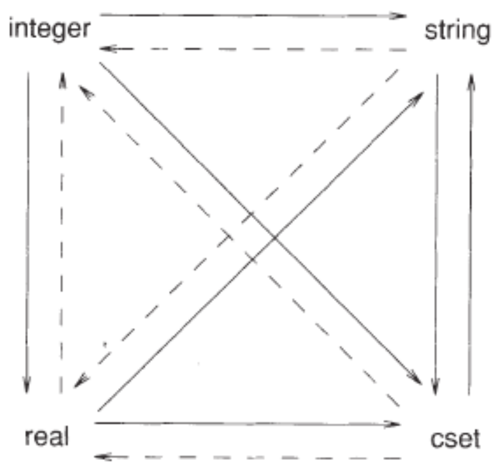
Icon supports polymorphism in terms of polymorphic operations. Although, the meanings of operations cannot be changed during program execution, several operations perform different computations depending upon the types of their operands. Thus, x[i] may subscript a string, list or a table. The meanings of some operations also depend on whether they occur in an assignment or a dereferencing context. Eg. If s is a string, assignment to s[i] occurs in an abbreviation for a concatenation followed by an assignment to s, while if s[i] occurs in a context where its value is needed, it is simply a substring operation. Moreover, the context cannot, in general, be determined at translation time.

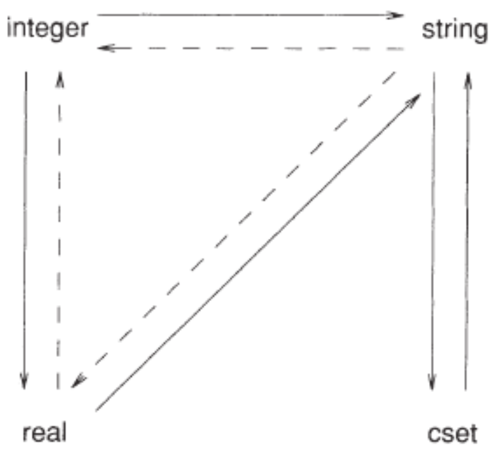
Eg. 1. ***s := "ad"***
s[3:3] := "a" # concatenation followed by assignment, s is now "ada"
write(s[1:3]) # substring operation
2. ***write(*str)*** # gives the size of the object, be it a string, list or table

5.7 Typing

Icon is different from conventional languages when it comes to typing. In Icon variables are not given explicit types. The operations done on the variables decide the type of the variables involved in that operation.

Eg. ***a*** = ***b*** means a compare on binary or on integers.
a == ***b*** means a compare on strings.
a === ***b*** means a compare on lists.





Since operators determine the type of the variable, Icon compensates for this by providing lots of operators. Also, due to this programs in Icon are a bit verbose, as every operator has to effectively encode the required type of the operands. But, at the same time the readability is improved since the presence of a particular operator clearly states the kind of operation that will be performed. The disadvantage is that since the type system is so dynamic, the compiler cannot perform any optimizations since the exact type is not till runtime. There are four types among which mutual type conversion is supported, strings, csets, integers and real numbers. Some conversions can be conditional, such as an integer conversion to real number is done only if the value is in the range of a C long.

Here the dotted lines indicate conditional conversion. Some conversions are natural and occur frequently, such as string to integer and vice versa. But some others are not, like cset to integer. These conversions are then done in a two-step process to reduce the number of conversion routines. cset is converted to a string first and then to an integer.

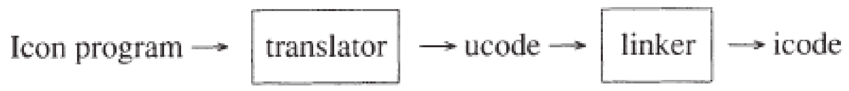
6. **Implementation Details**

6.1 The Icon Virtual Machine

The Icon implementation uses the virtual machine approach. Virtual machines serve as an abstraction for the underlying real machine. Using virtual machines a single common model can be developed that can be ported across various hardware architectures, even when operations of a language do not fit a particular architecture. However, Icon's implementation of the virtual machine doesn't rigidly specify a structure for all aspects of the language such as type checking, storage allocation and garbage collection.

6.2 Components of Implementation

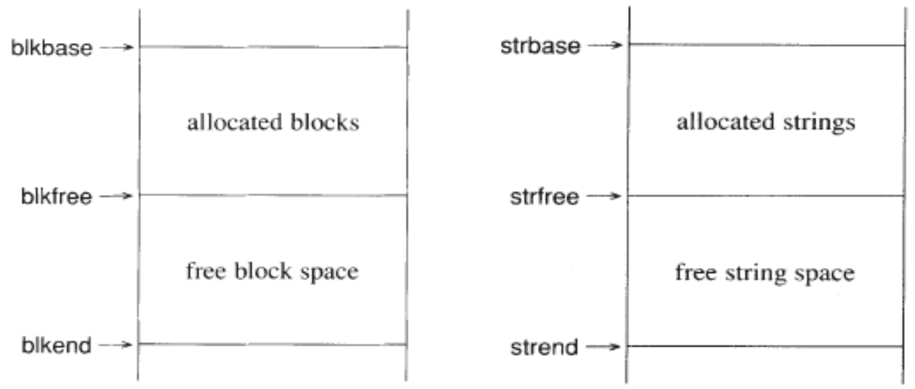
The implementation is comprised of three major components: i) a translator, ii) a linker and iii) a run-time system. The translator is like a compiler, it converts an icon program to virtual machine instructions, known as *ucode*, which is ASCII text. Linker combines one or more ucode files into a single program for the virtual machine. The linker output is called *icode*, which is a binary representation for compactness and ease of processing by the virtual machine.



The run-time system consists of interpreter for icode and a library of support routines to carry out various operations that may occur when an Icon program is executed. The interpreter serves as a software realization of the Icon virtual machine. The creators' note that the difference in execution speeds between a linked executable and interpreted code is insignificant, the reason being the executable generated is for the virtual machine and not a real machine, which still has to make run-time library routine calls to execute the virtual machine instructions.

6.3 Storage Management

Storage Management in Icon accommodates requirements of a wide range of programs. It is designed to be fast and simple and is implicit and automatic. Objects are created as needed during program execution. They grow and shrink automatically and unused space is garbage-collected when necessary. There also is no limit on the size of objects other than the amount of available memory. Data is allocated in separate regions for different types of objects, namely static blocks, non-static blocks and strings. Static blocks are allocated in a static region and are never moved, though they can be reused. Co-expression blocks are allocated in the static region since their C stacks contain internal pointers that are difficult to relocate to another space in memory. Blocks and strings are allocated in separate regions, both using a free-list approach as shown.



Garbage collection in Icon uses the marking and compaction technique. It is divided into two phases, location and compaction. The location phase locates all the roots and their children which are alive. During compaction the live objects are moved from “from” to “to” space and “from” space is cleared.

Icon uses the predictive need mechanism for garbage collection. A predictive need request assures an adequate amount of space before the allocation and no garbage collection occurs during the following allocation request. The advantage of this is that garbage collector can be invoked at a safer time than the actual request allocation time while he disadvantage is that the maximum amount of storage must be determined and care must be taken to make predictive need requests prior to allocation. When garbage collection occurs all potentially accessible data must be reachable from the roots. This is not easy, as pointers to live objects may only exist in registers or on the C stack that the garbage collector cannot locate. Also, objects that are being constructed may temporarily hold invalid data. Hence, assuring that all live data is reachable and valid is difficult and error prone. Instead of garbage collection occurring as a by-product of an allocation request, the amount of space requested is reserved in advance. There are two routines, blkreq and strreq for which check the block and string regions respectively to assure availability of free space.

7. Missing features

The following features are not supported in basic Icon:
* Object Oriented programming
* Exception Handling
* Concurrency and related control structures
* Modules with distinct namespaces

However, extensions like Unicon adds OO features and MT-Icon adds multi-tasking functionality.

8. Summary

Overall, Icon is a powerful language for scripting and rapid development. It has advanced control constructs, data structures and built-in functions which make it a feature rich language. It is easy to learn and program. Though today Icon is a dead language, it was probably ahead of its time and has influenced popular languages of today like Python, Perl & Ruby. On the negative side, Icon code can become increasingly complex to read as a single line of code is capable of a lot of computation and with different control constructs and a wide array of operators, code readability may be hindered considerably.

References

1. http://www.cs.arizona.edu/icon
2. The Icon Programming Language, *Ralph E. Griswold and Madge T. Griswold.*
3. The Implementation of the Icon Programming Language, *Ralph E. Griswold and Madge T. Griswold*
4. http://www.mitchellsoftwareengineering.com/icon
5. The Icon Handbook