12-3-1992

Icon: A Interpreter-Based Approach

Dong Wei
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Programming Languages and Compilers Commons

Recommended Citation
http://lib.dr.iastate.edu/cs_techreports/158

This Article is brought to you for free and open access by the Computer Science at Iowa State University Digital Repository. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Icon: A Interpreter-Based Approach

Abstract
This paper introduces the Icon programming language in a simplified form using a lisp-like syntax to represent an interesting subset of the language. The paper also discusses the semantics and implementation of some of Icon's advanced features with emphasis on generators and goal-directed evaluation. The organization of the paper resembles a chapter in the book "Programming Languages: An Interpreter-Based Approach" by Samuel Kamin. This paper can serve as a supplement to that book, allowing comparative programming language instructors who use the Kamin text to efficiently teach the Icon programming language.

Disciplines
Programming Languages and Compilers
Icon: An Interpreter-Based Approach

TR 92-38
Dong Wei

December 3, 1992

Iowa State University of Science and Technology
Department of Computer Science
226 Atanasoff
Ames, IA 50011
Icon: An Interpreter-Based Approach

Dong Wei

December 3, 1992
Abstract

This paper introduces the Icon programming language in a simplified form using a lisp-like syntax to represent an interesting subset of the language. The paper also discusses the semantics and implementation of some of Icon’s advanced features with emphasis on generators and goal-directed evaluation.

The organization of the paper resembles a chapter in the book “Programming Languages: An Interpreter-Based Approach” by Samuel Kamin. This paper can serve as a supplement to that book, allowing comparative programming language instructors who use the Kamin text to efficiently teach the Icon programming language.
1 Introduction

Icon is a high-level, general-purpose programming language that resembles C in appearance but has many characteristics different from conventional languages such as Pascal and C. It was first developed at the University of Arizona in the late 1970’s following work on the SNOBOL languages, known for their strengths in string analysis and pattern matching. Icon elegantly integrates SNOBOL’s pattern-matching capabilities with conventional control structures to yield a more coherent and powerful programming language. Icon was designed for string and list processing, as well as for general high-level programming tasks. It is suited to situations where a quick solution is needed since it emphasizes ease of programming at the expense of efficiency of execution. In this respect it follows the design philosophy of SNOBOL: the programmer is a more valuable resource than the computer.

Icon is conventional in many respects. Like C and Pascal, it is an imperative, procedural language with variables, operations, procedures, and conventional data types. But Icon places more of an emphasis on manipulation of high-level data structures such as variable-sized strings and lists, and associative tables using a novel control structure based on goal-directed expression evaluation. There are also aspects of logic programming besides those of imperative execution in Icon.

Icon has no type declarations. As in Lisp, every variable can represent any value of any type, and strings and structures are created during program execution, instead of being declared and allocated during compilation. Storage management is automatic and thus garbage collection is performed when necessary.

Icon is an expression-based language with a reserved-word syntax. It has no statements but only expressions. The most significant aspect of Icon is goal-directed evaluation. In Icon, an expression may produce a sequence of results. Such expressions are called generators. Goal-directed evaluation attempts to produce at least one result for each expression. Whenever there are multiple ways to satisfy the subgoals that comprise a particular expression, Icon automatically searches all combinations of alternative subgoal solutions to find a particular combination that satisfies all goals. The combination of generators and goal-directed evaluation is unique to Icon, and it contributes greatly to Icon’s expressive power.

Like many other languages, Icon is still evolving. It gradually and steadily
gains popularity, especially in areas related to list and string processing.

2 The Language

2.1 Syntax

The syntax of our Icon dialect is represented by the following context-free grammar.

input -> expression
expression -> value | variable
   | (if expression expression expression)
   | (while expression expression expression)
   | (:= variable expression)
   | (begin expression+)
   | (procedure variable (variable*) expression)
   | (optr expression*)
   | &fail

optr -> procedure | value-op | function

value-op -> + | - | * | / | = | ~= | < | >
   | <= | >= | << | >>
   | [] | $ |
   | ||
   | every
   | to
   | |

function -> | write
   | find
   | right

value -> integer | quoted-const
quoted-const -> 'integer | 'string
string -> name
procedure -> name
variable -> name
integer -> sequence of digits, possibly preceded by minus sign
name -> any sequence of characters not an integer, and not containing a blank or any of the following characters:

Comments are introduced by the character ‘;’ and continue to the end of the line. A session is started by invoking the interpreter, and is terminated by entering “quit” or Control-D at the interpreter prompt.

2.2 Data Types

Our version of Icon uses two basic data types: integer and string. As mentioned above, any type of value can be assigned to any variable. For example, x might have an integer value at one time and a string value at another. The following code represents two assignments to x:

```
-> (= x 5)
5
-> (= x ’hello)
hello
```

:= is the assignment operator, which assigns the value of its second argument to its first argument. Its first argument must be a variable. Notice that a result is displayed after an assignment expression is evaluated, since the assignment expression (not a statement) produces the value assigned to the variable.

2.3 Success and Failure

Icon programs consist of expressions. Every expression may succeed, i.e., produce a value, or may fail, i.e., produce no value. An expression of a single integer or string always succeeds. Expressions that may either succeed or fail are called conditional expressions. One example is a relational comparison expression.
If an expression that is to produce a result for use as an argument within an enclosing expression fails, evaluation of the enclosing expression fails also. The following assignments are conditional on the successes of the integer comparison operations. Consider the following example, in which the > operation produces the value of its second argument if its first argument is greater than the second, but fails otherwise:

```
-> (:= k 5)
5
-> (:= i (> k 3)) ; (> k 3) produces 3, which is assigned to i
3
-> (:= i (> i 8)) ; (> i 8) fails, no assignment is performed
-> i ; i keeps its old value
3
```

Generally, failure occurs if a relation does not hold or if an operation cannot be performed but is not actually erroneous. Icon has a reserved word &fail, which is an expression that always fails.

```
-> &fail ; produces no result
```

### 2.4 Traditional Arithmetic Operations

There are five arithmetic operations for integers: + (addition), - (subtraction), * (multiplication), / (division) and % (modulo). These operations are illustrated by the following examples.

```
-> (:= i (+ 1 4))
5
-> (- i 3)
```
2
-> (* 2 (+ 1 i))
12
-> (:= k (/ 28 4))
7
-> (% k 2)
1

2.5 Integer and String Operations

The expressive power of Icon derives from its large set of built-in operations. Besides +, -, *, / and % (modulo) operations for integers, our interpreter also has comparison operations for integers and strings. A comparison operation produces the value of its right argument if the specified relation holds, but fails if the relation does not hold. Some examples are shown below,

-> (:= i (= 0 5)) ; not equal to
5
-> (write (= 10 i)) ; equal to (fails, write is not invoked)
-> (< (< 1 i) 10) ; succeeds if i is between 1 and 10
10
-> (> (i % 3) 1)
1

The string comparisons are based on lexical (alphabetical) order.

-> (:= s (== 'OK 'ok)) ; lexically not equal
ok
-> (write (== s 'ok)) ; lexically equal
ok
ok
-> (>> s 'world) ; lexically greater than (fails)
-> (<< (<< 'hi s) 'there) ; lexically less than
there

The last expression succeeds since string ok is strictly between hi and there in lexical order.

Besides the string comparison operations, the interpreter implements the following three string operations:
• Subscripting:
  \[ ([\ ] \text{string} \ n) \]
  which produces the character in \text{string} at position \( n \) as a string of length 1, fails if \( n \) is out of range.

• Size:
  \[ (\$ \text{string}) \]
  which produces the length of \text{string}.

• Concatenation:
  \[ ([|| \text{string1} \text{string2}) \]
  which produces a new string consisting of characters in \text{string1} followed by those in \text{string2}.

Some examples are:

\[-\rightarrow (:= \ s \ 'High)\]
\[\text{High}\]
\[-\rightarrow ([[\ ] \ s \ 1) \quad ; \text{position is } 1\text{-based}\]
\[\text{H}\]
\[-\rightarrow ([[\ ] \ s \ ($ \ s))\]
\[\text{h}\]
\[-\rightarrow ([|| \ s \ '-level)\]
\[\text{High}-level\]

There are three built-in functions as shown below. More details will be presented in the following sections.

• \((\text{write} \ expr)\)
  prints out and returns the value of \( expr \) if the evaluation of \( expr \) succeeds. It fails otherwise.

• \((\text{find} \ \text{string1} \ \text{string2})\)
  produces the sequence of positions at which \text{string1} is found within \text{string2}, but fails if \text{string1} does not occur in \text{string2}. Mechanisms for dealing with multiple results are described in section 2.8.
(right s i)
produces a string of length i in which string s is positioned at the right
and blanks are used to pad out the remaining characters to the left.
For example,

-> (right 'string 8) ; two blanks are inserted to the left
  string
-> (right 'string 4) ; string is truncated at the left
  ring

2.6 Control Structures

Control structures use the success or failure of one expression to govern the
evaluation of others.

We use the same syntax for the if expression as in Lisp. If the first
argument succeeds, the second subexpression is evaluated and its result is
the result of the whole if expression. Otherwise, the third subexpression is
evaluated and its result is returned for the if expression. If evaluation of the
second or third argument fails when being evaluated, the whole expression
also fails.

-> (:= count 1)
  1
-> (if (> count 0)
     'positive
     'nonpositive)
positive

Another control structure is the while expression. It repeatedly eval-
uates its first argument and the success or failure of the evaluation controls
evaluation of the second argument. The loop terminates when the evaluation
of the first argument fails, at which time the while expression fails. A while
expression is evaluated only for its side-effects. For example,

-> (:= i 0)
  0
-> (while (i < 5)
2.7 Procedures

Procedures are the major logical units of an Icon program. The following procedure returns the $n$th Fibonacci number.

```
(begin
  (:= i (+ i 1))
  (write (* i 2))))
```

A procedure call may fail to produce a result in the same way that a built-in operation can fail. For example,

```
-> (procedure fib (n)) ; procedure declaration
  (if (= n 1)
    1
    (if (= n 2)
      1
      (+ (fib (- n 1)) (fib (- n 2)))))

fib
-> (fib 6) ; a procedure call
8
```

This procedure returns the half value of $n$ if $n$ is an even number, or fails otherwise.

```
-> (procedure half (n)
  (if (= (% n 2) 0)
    (/ n 2)
    &fail))

half
-> (half 15)
-> (half 14)
7
```
Similar to user-defined functions in Scheme, procedures are also data values. A procedure declaration constructs a procedure and assigns it to the procedure name. Procedures may be assigned to variables, passed as arguments to other procedures, and so forth.

```
-> (:= x half)
<procedure>
-> (x 10)
5
-> (:= x write)
<procedure>
-> (x 'ok)
ok
ok
-> (:= write half) ; change the meaning of the identifier write
<procedure>
-> (write 8)
4
-> (procedure test (x y z)
  (x y z))
test
-> (test find 'st 'a_string)
3
```

### 2.8 Generators and Goal-Directed Evaluation

Icon generalizes the traditional view of expression evaluation in which expressions evaluate to a single result. A generator in Icon may produce a sequence of results. A simple example is the integer operation `to` which generates the integer sequence ranging from its first integer argument to its second argument. For example, the expression `(to 1 5)` is capable of generating the sequence of integers from 1 to 5. But only the first result is produced in the following context:

```
-> (to 1 5)
1
```
In general, the number of results a generator produces depends on its surrounding context. There are two contexts in which a generator may produce more than one result: iteration and goal-directed evaluation.

An iteration resumes a generator repeatedly to produce all its results. The iteration fails when no more results can be generated. Iteration is designated by the `every` control structure. Consider the following session:

```
  -> (every (write (to 1 5)))
  1
  2
  3
  4
  5
  -> (every (:= i (to 1 5)))
  -> i ; i keeps the last value assigned to it
  5
  -> (:= k 1)
  1
  -> (every (:= k (* k (to 1 i))))
  -> k
  120
```

The last `every` expression computes the factorial of `i`. As seen in the above example, iteration can often replace a traditional loop control structure to simplify programming. The following is another example,

```
  -> (:= s  'abcde)
  abcde
  -> (:= t ([] s 1))
  a
  -> (every (:= t (| ([] s (to 2 ($ s))) t)) ; reverse s
  -> t
  edcba
```

The other two generators used in our version of Icon are

- alternation:

  ```
  ([ expr1 expr2]
  ```

  10
which generates the results of \textit{expr1} followed by the results of \textit{expr2};
and

- \textbf{find:}

\begin{verbatim}
(find \textit{string1} \textit{string2})
\end{verbatim}

which generates all positions at which \textit{string1} is found within \textit{string2},
but fails if \textit{string1} does not occur in \textit{string2}.

Here are some examples:

\begin{verbatim}
-> (every (write (1 0 1)))
0
1
-> (find 'e 'there)
3
-> (every (write (:= i (find 'e 'there))))
3
5
-> i
5
\end{verbatim}

Goal-directed evaluation is the most novel aspect of Icon. It is a strategy
that proceeds forward on the current evaluation path in an attempt to satisfy
a goal, and backtracks to the most recently visited alternative path provided
by generators if the goal cannot be satisfied along the current path. Its
purpose is to achieve success for expression evaluation. The usefulness of
goal-directed evaluation in Icon comes from the capability of generators to
produce more than a single result since the generators provide the source of
alternatives used by goal-directed evaluation. For example,

\begin{verbatim}
-> (:= s1 'is)
is
-> (:= s2 'This_is_a_string)
This_is_a_string
-> (if (= 6 (find s1 s2))
  'yes
  'no)
'yes
yes
\end{verbatim}
The meaning of the last expression is: “If \texttt{s1} occurs as a substring of \texttt{s2} at position 6, then return \texttt{yes}; otherwise return \texttt{no}”. Evaluation of this expression pursues the most deeply nested goals first. The generator \texttt{find} first produces a value 3, but the test for equality with 6 fails. Thus \texttt{find} is resumed to produce the alternative result 6, which satisfies the goal. Finally, \texttt{yes} is returned as the result of this whole expression.

As demonstrated above, goal-directed evaluation results in control backtracking to obtain alternative results from expressions (generators) that have previously produced results when the current operation fails. Consider the following examples:

\[
\begin{align*}
\rightarrow & \ (\= \ i \ (\ast \ 2 \ (< \ 5 \ (\text{to} \ 1 \ 10)))) \\
& 12 \\
\rightarrow & \ (> \ (< \ 3 \ (\= \ i \ (\text{to} \ 1 \ 8))) \ 6) \\
& 6 \\
\rightarrow & \ i \\
& 7 \\
\rightarrow & \ (<<< \ '\text{'one} \ (\text{'an} \ '\text{'two})) \\
& \text{two} \\
\rightarrow & \ (< \ 10 \ (\text{find} \ '\text{'is} \ '\text{'this\_is\_a\_string}); \text{can't satisfy the goal}}
\end{align*}
\]

Using the | generator, the declaration of the procedure \texttt{fib(n)} in the previous section can be simplified as:

\[
\begin{align*}
\rightarrow & \ (\text{procedure fib} \ (n)) \\
& \quad (\text{if} \ (= \ n \ (\text{l} \ 1 \ 2)) \\
& \quad \ 1 \\
& \quad \ (+ \ (\text{fib} \ (- \ n \ 1)) \ (\text{fib} \ (- \ n \ 2))))
\end{align*}
\]

2.9 Bounded Expressions

During goal-directed evaluation, control backtracking is limited by a number of syntactic constructs. In

\[
(\text{if} \ \text{expr1} \ \text{expr2} \ \text{expr3})
\]

if \text{expr1} succeeds, but \text{expr2} fails, \text{expr1} is not resumed for another result. Otherwise, the traditional \textbf{if} semantics would be violated. In this context,
expr1 is said to be bounded. Backtracking may occur within a bounded expression, but once a bounded expression produces a result, it cannot be resumed to obtain another.

Some bounded expressions are listed below.

- The control expression of an if expression is bounded.
- In (procedure variable (args-list) expr), expr is bounded.\(^1\)
- In (while expr1 expr2), both expr1 and expr2 are bounded.
- In (begin expr1 expr2 ... ), all expressions except the last one are bounded.

Some examples are:

\[
\begin{align*}
\rightarrow & (\text{if} \ (\!: i \ (\text{find} \ 'e \ 'hello))) \\
& \quad \quad \text{(write} \ (< \ 3 \ i)) \quad ; \text{fails} \\
& \quad \quad \text{(write} \ '\text{wrong})) \\
\rightarrow & (\text{procedure} \ p \ (i) \ (1 \ (+ \ i \ 1) \ (* \ i \ 2))) \\
p \\
\rightarrow & (< \ 5 \ (p \ 3)) \ ; \text{p is not resumed} \\
\rightarrow & (< \ 4 \ (\text{begin} \\
& \quad \quad \text{(write} \ (1 \ 1 \ 2)) \quad ; \text{bounded, will not write} \ 2 \\
& \quad \quad \text{(to} \ 1 \ 10))) \quad ; \text{resume} \ 4 \ \text{times till} \ 5 \ \text{is produced}
\end{align*}
\]

As shown above, bounding is necessary during goal-directed evaluation. Icon integrates goal-directed evaluation into conventional control structures with the discipline provided by bounding. With generators providing the source of alternatives, this aspect of expression evaluation allows many kinds of computations to be formulated in a natural and concise way.

\(^1\)The procedure block of a real Icon program may include suspend expressions which allow the procedure to generate and return multiple results from the procedure body. This behavior is consistent with our characterization of the procedure body as a bounded expression, because once the complete procedure body terminates, it cannot be resumed.
3 Implementation

The Icon interpreter is written in C++, based on a C++ implementation of the Kamin interpreters by Tim Budd. Since the Icon interpreter reuses considerable code from the Lisp interpreter, this section focuses primarily on the implementation of generators and goal-directed evaluation in the Icon interpreter. See references [1] and [4] for background discussions of the Lisp interpreter implementation.

3.1 Declaration

Generators and goal-directed evaluation pose special difficulties to the implementor. The internal state of a generator must be saved between the time the generator produces a result and the time it is resumed to produce alternative results. For goal-directed evaluation, the previously evaluated parts of an expression and their internal state variables must be maintained.

Like the Lisp interpreter, an expression entered during a read-eval-print loop is parsed into a list. In the Icon interpreter, one more field, oldh, is added to each list node structure as shown below,

```cpp
class ListNode {
    expr h;    // points to the value returned by its subtree
    expr oldh; // points to the original subtree
    expr tail; // points to the next node
}
```

oldh provides the expression that represents the original argument, which is used to generated alternative values for the argument if backtracking is needed.

3.2 Traditional Evaluation

In the absence of backtracking, expressions are evaluated in the same way as in Lisp. When an operation other than a single integer or string is evaluated, the function apply is invoked. This function takes as an argument the target for the evaluation and a list of unevaluated arguments. It first evaluates the arguments, using the simple recursive routine evalArgs, and then invokes
the function `applyWithArgs`, which applies the operation to the arguments, assigning the result to `target`. More details can be found in reference [1].

3.3 Backtracking and the Resume Function

Each class of operation has a corresponding resume function. If the operation is resumed, the resume function for it is called. In our version of Icon, when a comparison operation fails, control backtracking occurs and the resume function for binary operations is invoked. The resume function attempts to produce an alternative result by considering alternative values produced by its argument subexpressions.

Therefore, the pseudo code of the `applyWithArgs` function for the integer comparison operations is as follows (the `applyWithArgs` function for the string comparison operations is similar):

```/*
target is the result produced by the operation;

args is the argument list;
args[0] and args[1] stand for the first and second argument;
args[0].h and args[1].h are the values of the arguments;
args[0].oldh and args[1].oldh point to the unevaluated expressions representing the arguments;

environment maintains the symbol-value pairs, such as the values of arguments passed to a function
*/
BooleanBinaryFunction::applyWithArgs(target, args, environment) {
    target = failure;
    /*
     * failure is a constant pointer to an empty list which indicates a failure. Note that an empty list is not a valid Icon value.
    */
    if (args[0].h or args[1].h is failure) {
        resume(target, args, environment);
```
return;
}
if (either args[0].h or args[1].h is not integer) {
    print error message;
    return;
}
if comparison succeeds {
    // the second argument is the result of the operation
    target = args[1].h;
}
else
    resume(target, args, environment)
}

The iteration operation every may also cause the computation to be resumed until no more results can be produced by the argument of every. The apply function of every is shown as follows:

Every::apply(target, args, environment)
{
    evaluate args[0];
    if args[0] can be resumed
        do {
            resume the expression pointed to by args[0].oldh;
        } while (target value returned is not failure);
    target = failure;     /* every always fails */
}

3.4 Resume Functions for Non-generator Expressions

The key to understanding goal-directed evaluation is to comprehend the implementation of the resume function. The implementation of generators and goal-directed evaluation is mainly the design of the resume functions for all types of expressions. The pseudo code for them is discussed below.

The resume function for a single integer or string simply assigns failure to target. For example,
The `resume` function for unary operations is also straightforward. It tries to resume its only argument to obtain a new value. If successful, it then applies the unary operation to the new argument value.

```cpp
UnaryFunction::resume(target, args, environment)
{
    if args[0] can be resumed /* not an integer or a string */
    {
        resume the expression pointed to by args[0].oldh;
        if a new value returned {
            assign it to args[0].h;
            applyWithArgs(target, args, environment);
            return;
        }
    }
    target = failure;   /* fails */
}
```

The `resume` function for binary operations is more complicated. According to Icon's semantics, the second argument is resumed first, and if it succeeds, the operation is applied. Otherwise, the first argument is resumed. If it succeeds, the whole second argument is reevaluated, and finally the operation is applied to the arguments. The operation fails if none of its two arguments can produce alternative results.

```cpp
BinaryFunction::resume(target, args, environment)
{
    if args[1] can be resumed {
        resume it;
        if a new value returned {
            assign it to args[1].h;
            applyWithArgs(target, args, environment);
            return;
        }
    }
    target = failure;   /* fails */
}
```
if args[0] can be resumed {
    resume it;
    if a new value returned {
        assign it to args[0].h;
        evaluate args[1];
        applyWithArgs(target, args, environment);
        return;
    }
    target = failure; /* fails */
}

3.5 Resume Functions for Generators

The \texttt{resume} functions for generators are tricky, since the expressions representing the arguments of a generator can be generators. First, the generator operation is applied to the initial values of its arguments to obtain the first result. Subsequent results are obtained by the generator operation considering alternative results that are consistent with its original parameterization. If this fails, the generator’s arguments are resumed from right to left, and if alternative argument values are available, the generator operation is applied again. Here, we present the \texttt{applyWithArgs} and \texttt{resume} functions for the \texttt{to} generator. Implementation of \texttt{|} (alternation) and \texttt{find} is left as an exercise for the reader.

\texttt{applyWithArgs}(target, args, environment) {
    if (args[0].h == failure) or (args[1].h == failure) {
        BinaryFunction::resume(target, args, environment);
        return;
    }
    if (args[0].h or args[1].h is not integer) {
        target = error("noninteger args")
        return;
    }
}
if (args[0].h <= args[1].h) {
    create a new node and append it to args list as args[2];
    args[2].h = args[0].h;
    /*
     * The new node is needed in order to correctly handle
     * nested generator expressions, e.g., (to 1 (1 3 5)).
     */
    target = value of args[0].h;
    args[0].h = args[0].h + 1;
} else
    resume(target, args, environment);
}

To::resume(target, args, environment)
{
    if (args[0].h <= args[1].h) {
        target = args[0].h;
        args[0].h = args[0].h + 1;
        return;
    }
    else {
        args[0].h = args[2].h;
        BinaryFunction::resume(target, args, rho);
        /*
         * The argument expressions are resumed. If a new result
         * is produced, To::applyWithArgs is invoked.
         */
    }
}

## 4 Icon as It Really Is

Icon is a very high-level language which has many advanced features to fa-
cilitate programming, especially string and list processing. We will briefly
discuss the syntax of real Icon, and address some of its important features.

4.1 Syntax

As mentioned above, Icon is an imperative, procedural language with a C-like syntax. For example, a program consists of procedure declarations, and it must have a procedure with the name main. Execution of an Icon program begins with the procedure main. Here is a program which outputs the first six Fibonacci numbers in real Icon:

```
procedure fib (i)
    if (i = (1 | 2)) then return 1
    else return fib(i-1) + fib(i-2)
end

procedure main()
    write("The first 6 Fibonacci numbers:")
    every write(fib(1 to 6))
end
```

4.2 Data Types

Icon has ten built-in data types. These include integer, list, procedure, real and string. New types can be added by record declarations in an Icon program.

The built-in function type(x) produces a string representation of the type of its argument. For example,

```
    type(4)    returns "integer"
    type("hello") returns "string"
    type(write) returns "procedure"
```

In many cases, values of one type can be converted to values of another type, either implicitly or explicitly. Implicit conversion occurs in contexts where the type of a value that is expected by an operation is different from the given type, as in:
\[ i := 2 < "50" \]
write(i)

The assignment above assigns the integer 50, not the string “50” to i, because the string “50” is coerced to an integer in order to perform the less-than comparison. Then before the write function is called, i is converted to a string in order to be written. But if an implicit conversion cannot be performed, it causes an error and terminates program execution. For example,

\[ i := "a" + 2 \]

is erroneous, because “a” cannot be converted to an integer.

Explicit conversion is performed by functions whose names correspond to the desired types. For example,

\[ s := \text{string}(210) \quad \text{assigns string } "210" \text{ to } s \]
\[ i := \text{integer}("10") \quad \text{assigns integer 10 to } i \]

If an explicit type conversion cannot be performed, the type conversion function fails. Therefore it can be used to test the convertibility of a value without causing execution errors.

### 4.3 String Scanning

String scanning is an important feature contributing to Icon’s string and list processing ability. It is a high-level facility for string analysis that suppresses the computational details associated with the explicit location of positions and substring specifications.

The string scanning expression

\[ s ? expr \]

specifies, by the value of \( s \), a subject string to which \( expr \) applies. The position within the subject string \( s \) is initially 1. \( expr \) is evaluated in the context of this subject and position.

The following two matching functions are useful in analyzing the subject:

- \( \text{tab}(i) \) sets the position in the subject to \( i \)
- \( \text{move}(i) \) increments the position in the subject by \( i \)
These functions return the substring of the subject between the old and new positions. If the position is out of the range of the subject, the matching function fails and the position is not changed. For example,

```
line ? while write(move(2))
```

writes successive two-character substrings of `line`, stopping when there are not two characters remaining and thus `move(2)` fails.

In string scanning, the trailing arguments of string analysis functions such as `find` are omitted; the functions apply to the subject at the current position. Consider the following program,

```
procedure main()
  while line := read() do
    line ? write(tab(find("//") | 0))
  end
```

`read()` is a function which reads a line from the standard input or fails if the input stream is exhausted. The above string scanning expression writes the initial portion of `line` up to an occurrence of the string “//”, exclusively; or if `find("//")` fails, `tab(0)` moves the position to the end of the subject string, and thus outputs the whole line. Therefore, this simple program can be used to filter out C++-style comments from standard input.

### 4.4 Lists

A list is a linear aggregate of values. One way to create a list is to place brackets around a list of expressions. For example,

```
players := ["Tom", "John", "Tony"]
```

assigns a list of three strings to `players`.

```
city := ["Ames", "Iowa", 50013]
```

assigns a heterogeneous list to `city`.

The function

```
list(i,x)
```
creates a list of values, each of which has the value of \( x \). For example,

\[
\text{vector} := \text{list}(100, 1)
\]

assigns to vector a list of 100 values, each of which is 1.

An empty list can be produced by \([\ ]\) or list(0).

An element of a list is referenced by a subscripting expression, and a list value is a pointer to a structure that contains the elements of the list. For example,

\[
\begin{align*}
\text{states} & := ["Iowa", "Texas", "Michigan"] \\
\text{states}[2] & := "Kansas" \\
\text{alist} & := \text{states} \\
\text{alist}[3] & := "Georgia"
\end{align*}
\]

results in \text{states} and \text{alist} both pointing to the list [“Iowa”, “Kansas”, “Georgia”].

The values in a list may be of any types. For example,

\[
\begin{align*}
\text{x} & := ["\ast", ["a"], ["+", ["b"], [2]]]
\end{align*}
\]

can be thought of as representing a tree in which the first value in the list is associated with the position in the tree and subsequent values represent subtrees. Thus, \( x \) represents the parse tree of expression \( a \ast (b + 2) \).

Lists are not fixed in size. They can be manipulated by stack and queue access functions and grow and shrink automatically using the functions:

- **put**\( (a, x) \): adds \( x \) to the right end of the list \( a \)
- **push**\( (a, x) \): adds \( x \) to the left end of the list \( a \)
- **pop**\( (a) \), **get**\( (a) \): remove an element from the left end of the list \( a \) and return it; but fail if \( a \) is empty.
- **pull**\( (a) \): removes an element from the right end of the list \( a \) and returns it; but fails if \( a \) is empty.

For example,

\[
\begin{align*}
\text{lines} & := [] \\
\text{while} \ & \text{push}(\text{lines}, \text{read}()) \\
\text{while} \ & \text{write}(\text{pop}(\text{lines}))
\end{align*}
\]

writes out the lines of the standard input in reverse order.
4.5 Procedures as Generators

Users may write their own generators by using the keyword `suspend` in the procedure body. For example,

```
suspend expr
```
returns the value of `expr` from the procedure call but leaves the call in suspension with the local identifiers intact. The procedure call can be resumed by iteration or goal-directed evaluation to continue evaluation from where it is suspended. Each suspension produces a result in its result sequence.

The following procedure is a user-defined generator:

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

The result sequence of the `To(i, j)` procedure is the same as the result sequence of the Icon expression `(i to j)`.

The suspend expression is like `every`: it iterates over all the results in the result sequence of its argument, suspending each result to the procedure’s invoker. For example,

```
procedure next (s)
    suspend s[find("t", s)+1]
end

procedure main()
    every write(next("this is a trivial test"))
end
```

outputs
```
here
```
which are each of the characters that follows a \( t \) in the string \( s \). Note that no character follows the last \( t \) in the scanned string, so the string subscripting expression fails.

When a sequence of results is needed, procedure suspension sometimes can be used to avoid redundant computation. For example, the infinite Fibonacci sequence can be generated by the following procedure.

```plaintext
procedure fibseq()
    local i, j, k
    suspend (i := 1) | (j := 1)
    repeat {
        suspend k := i + j
        i := j
        j := k
    }
end
```

The expression `every(write(100 > fibseq()))` writes all the Fibonacci numbers less than 100 in sequence.

### 4.6 Garbage Collection

Icon has no type declarations. Strings, lists and other structures are created during program execution and the memory space for them is allocated as needed at run time. The space for the structures no longer in use is reclaimed by an automatic garbage collection.

### 5 Summary

Superficially, Icon programs resemble those in many other modern programming languages. But Icon integrates traditional control structures and a powerful control abstraction — goal-directed evaluation in a uniform way. It also has a very large set of high-level built-in operations, especially for string and list processing. Therefore, it requires relatively less effort to program in Icon. Not surprisingly, the implementation of Icon is more complicated than the implementations of more traditional languages such as Pascal and C.
Glossary

- **control backtracking.** A mechanism in which control may return to an earlier portion of an expression during goal-directed evaluation.

- **generator.** A procedure or expression that can be resumed to produce more than one result. Generators provide the source of alternatives used by goal-directed evaluation.

- **goal-directed evaluation.** An evaluation mechanism that attempts to produce at least one result for each expression. If the current evaluation path fails to satisfy a goal (produce a result), control backtracks to a previously visited alternative execution path to resume that subexpression (control backtracking). Backtracking continues until available alternatives have been exhausted.

- **list.** A built-in data type. An Icon list consists of a linear aggregate of values. Lists can be used as one-dimensional arrays, as well as stacks and doubly-ended queues.

- **polymorphic operation.** An operation that performs different computations depending on the types of its operands. One example is the subscripting operation: \( x[i] \) may be applied whether \( x \) is a string or a list.

- **procedure (function).** A logical unit of a program of the type procedure. Icon functions are built-in procedures. Procedures and functions are data values which may be assigned to variables.

- **string.** A sequence of characters. Icon strings are represented literally by enclosing within double quotation marks. The size of a string is the number of characters in it.

- **string scanning.** A control structure designed to simplify pattern matching on strings. The computations are performed at a high level that eliminates details such as the explicit computation of indexes to keep track of positions. It has the form of

  \[ s \ ? \ expr \]

  which applies \( expr \) to the subject string \( s \).
success and failure. An expression succeeds if it produces a value; or fails if it produces no value. Failure usually occurs if a relation does not hold or if a certain operation cannot be performed because of domain errors.

6 Exercises

1. Given a positive integer $n$ greater that 2, write an expression that evaluates to 0 if $n$ is not a prime number, or fails otherwise without using loop.

2. Given a positive integer $n$, write a procedure that returns the product of factorials of 1 to $n$.

3. Write a procedure substr2 consisting of a single expression which displays all the substrings of length 2 of a given string argument.

4. Write a procedure str2 that displays all combinations of two characters in the given string. For example,

```
-> (str2 'abcd)
ab
ac
ad
bc
bd
cd
```

5. Many word puzzles depend on the intersection of two words in a common character. Using generators and the operation [], write a procedure (upto string1 string2) which produces a position in string2 of an intersection if there is one, or fails otherwise. For example,

```
-> (upto 'about 'position)
2
```
6. Using **right** and the **upto** procedure of the previous exercise, write a procedure **cross** which displays an intersection of two strings as shown in the following example,

```
-> (cross 'lottery 'boat)
b
lottery
 a
t
```
Acknowledgments

I would like to thank my advisor, Dr. Kelvin Nilsen, for his careful guidance and help throughout my work. Tim Budd provided the interpreters for the languages in Kamin’s book which were written in C++. The coding of the Icon interpreter derived much from his work. I also would like to thank other people, especially Youngtae Kim, who have provided help and encouragement during my graduate studies.

References

1. Tim Budd, the Kamin Interpreters in C++ (report and programs), anonymous ftp from cs.orst.edu, directory pub/budd/kamin.


