The T Implementation of Icon

David Gudeman

ABSTRACT

This report describes a program in the T programming language that implements a large subset of the Icon programming language. The implementation uses a continuation semantics of Icon.

1. Introduction

In most programming languages, an expression returns exactly one result. The result may be a structured type containing many values, but it is still a single result. In the Icon programming language [4], an expression is capable of producing a sequence of zero or more results. These results are produced one at a time, as they are required by the context in which the expression is evaluated. The sequence of results that an expression is capable of producing is called its *result sequence*. For instance, the result sequence of the Icon expression "1 **to** 5" is {1, 2, 3, 4, 5}. This does not mean that "1 **to** 5" *will* produce the given result sequence, only that it *can* produce the sequence, depending on its context. This evaluation mechanism is a completely general part of the Icon language. As a consequence, there is no straightforward way to interpret Icon with another language.

The evaluation mechanism in Icon is said to be *goal-directed*, because all combinations of the results of all subexpressions may eventually be evaluated in an attempt to produce a result. In other words, the evaluation mechanism will try to satisfy the goal of producing a result. For a binary operator, the operation may potentially be performed on all possible pairs of results from the two operands in an attempt to produce a result. If the binary operator is a subexpression of another expression which fails on that result, the binary operator is resumed to try to produce another result. Since all possible pairs of both result sequences may be produced, the evaluation mechanism is often called cross-product evaluation. This report describes a T [3] program that uses a continuation semantics of Icon [5] to implement a large subset of Icon's expression evaluation mechanism, meaning goal-directed, cross-product evaluation of expressions which may produce zero or more results. Other features of Icon, such as expressions that return variables or structured data types are not implemented.

The purpose of this program is to give a small, understandable, implementation of Icon. This intrepreter may be more easily extendible than the C implementation, and the mathematical nature of the implementation, may suggest useful extensions to Icon.

2. Notations and Conventions

The syntax of Icon is given in reference [4]. Icon objects are represented in open-face brackets ($[\![\ ]\!]$), with Icon words and symbols given literally, and syntactic variables given with single capital letters, possibly subscripted. For instance, $[\![\mathbf{E}]\!]$ refers to an arbitrary Icon expression, and $[\![\ \mathbf{if}\ \mathbf{E_1}\ \mathbf{then}\ \mathbf{E_2}]\!]$ refers to an Icon if expression, where $[\![\mathbf{E_1}]\!]$ and $[\![\mathbf{E_2}]\!]$ are arbitrary expressions. There is no possibility of confusing literal objects with syntactic variables because there are no literal objects in Icon that use capital letters, except for program variables. Only lower-case letters are used in this report for program variables.

The open-face brackets can be thought of as quotes, so that literal Icon text is always "quoted". The difference between the open-face brackets used in this report and normal quotes, is that variables may appear inside of the open-face brackets. But these variables always represent text that would be quoted if the variable were not used in its place. Bound variables of a lambda-expression that represent syntactic objects are given in boldface without brackets. For example, a function that takes an expression as its only argument would be written as $\lambda\mathbf{E}.<exp>$ rather than as $\lambda[\![\mathbf{E}]\!].<exp>$.

Domain names are two or three letters long. They are given in bold for syntactic domains and italic for semantic domains. For instance, **Exp** is the domain of Icon expressions, and *Rea* is the domain of real numbers. The syntactic and semantic domains needed for a denotational description of Icon expressions are listed in Appendix A. Listed with the domain names, are single-letter variable names. These variables are always used (possibly with subscripts and primes) to represent a member of the given domain. For example, in the expression $\lambda\mathbf{I}.(s\ \mathbf{I})$, I represents an identifier, and s represents a store.

There is some difficulty in discussing Icon expressions because it is not always clear whether a given expression controls or is controlled by a related expression. For example, in $[\![\mathbf{E}_1 + \mathbf{E}_2]\!]$, $[\![\mathbf{E}_1]\!]$ controls whether $[\![\mathbf{E}_2]\!]$ is evaluated or not, but $[\![\mathbf{E}_2]\!]$ can cause $[\![\mathbf{E}_1]\!]$ to be resumed by failing. To help avoid confusion the following terminology is adopted. An expression $[\![\mathbf{E}_2]\!]$ is defined to be a post-expression of $[\![\mathbf{E}_1]\!]$ if the failure of $[\![\mathbf{E}_2]\!]$ will result in a resumption of $[\![\mathbf{E}_1]\!]$. Likewise, $[\![\mathbf{E}_1]\!]$ is called a pre-expression of $[\![\mathbf{E}_2]\!]$ if $[\![\mathbf{E}_2]\!]$ is a post-expression of $[\![\mathbf{E}_1]\!]$. Usually, a post-expression follows its pre-expression in the syntactic order of an Icon program. For example, in $[\![\mathbf{E}_1 + \mathbf{E}_2]\!]$, $[\![\mathbf{E}_1]\!]$ is a pre-expression of $[\![\mathbf{E}_2]\!]$, and $[\![\mathbf{E}_2]\!]$ is a post-expression of $[\![\mathbf{E}_1]\!]$.

Lambda notation is used as a notation to define functions. Where convenient, functions of multiple parameters are defined as $\lambda x\, y\, z.<expr>$, which can be curried, since it is a short-hand for $\lambda x.\lambda y.\lambda z.<expr>$. In a lambda expression, everything to the right of the dot is part of the function.

Concatenation represents function application, and associates to the left, although parentheses can be used to force a given interpretation. Thus, $f\ g\ x = (f\ g)x$, not $f\ (g\ x)$. Binary mathematical operations (+, ×, etc) are used in their normal infix form. For example, $plus = \lambda x\, y.\ x+y$.

Another notation used to define functions uses the symbol, $\equiv$, which should be read as "is defined to be". This notation is related to lambda notation in that $f \equiv \lambda x\, y.\ <expr>$, where $<expr>$ is some equation in x and y, is the same as $f\ x\ y \equiv <expr>$, and is the same as $f\ x \equiv \lambda y.\ <expr>$. Note that $\equiv$ is not a predicate, it is used only to introduce definitions. If $x \equiv y$ (x is defined to be y) then it is the case that $x = y$, but not conversely

It is often convenient to introduce symbols to represent functions, instead of writting out the lambda expression to describe it. Then it is necessary to define the symbol in a **where** clause [6]. For instance, the function, $f \equiv \lambda x.\ x+(\lambda y.\ x+y)\,5$ would be defined as:

$$f\ x \equiv x + g\ 5$$

$$\textbf{where } g\ y \equiv x+y$$

Variable names that occur at two different levels (such as x above) refer to the same thing at all levels. Notice that the function g had to be defined with the argument y, to avoid confusing it with the x used to define the function f.

Lambda notation and **where** notation may be freely mixed, so the above equation could have been written as

$$f\ x \equiv x + g\ 5$$

$$\textbf{where } g \equiv \lambda y.\ x+y$$

The **where** part may have a nested **where**, for instance

$f\ x \equiv x + g\ 5$

where $g\ y \equiv x + h\ x$

where $h\ z \equiv x{+}y$

defines $f \equiv \lambda x.\ x + (\lambda y.\ x + (\lambda z.\ x{+}y)x)5$. The xs all refer to the same value, no matter how deeply nested.

The **where** may also be followed by **and**, which is used to define a second (or third, etc.) function that is left undefined in the previous equation. For example:

$f\ x \equiv h\ x + g\ 5$

where $h\ y \equiv x{\times}y$

and $g\ z \equiv x$

Notice that both the **where** and the **and** parts can use variables from the equation they are defining.

The **where** notation also allows the definition of recursive functions. The binomial coefficient $\binom{n}{k}$ can be defined by

$Choose\ n\ k \equiv fact\ n\ /\ (fact\ k \times fact(n-k))$

where $fact\ m \equiv$ **if** $m = 0$ **then** 1 **else** $n \times fact(m-1)$

The **if-then-else** notation has the usual meaning. In this report, **where** notation is used almost exclusively except where it is desirable to define an unnamed function, in which case lambda notation is used.

It is important not to confuse syntactic expressions with mathematical expressions. When something is given in open-face brackets, what is being represented is the literal character string inside the brackets (except for syntactic variables, which are used to represent a *set* of literal strings). All other expressions are mathematical notation, which represent mathematical objects. For instance, the symbol ⟦"5"⟧ represents the actual syntactic string “"5"” used in an Icon program to represent an Icon string. The symbol ⟦5⟧ represents the syntactic string “5” used in an Icon program to represent the integer *five*, and the symbol 5 represents the actual integer *five*. In the expression

$F⟦\mathrm{E}+5⟧ \equiv F⟦\mathrm{E}⟧ + 5$

F represents a mathematical function that maps from Icon expressions to numbers. E represents a set of syntactically legal Icon expressions. The symbols + and 5 in the brackets are characters that occur literally in an Icon program, while the right hand + represents a mathematical function and the right hand 5 represents an integer.

A variable that takes its values from domain A is said to be of type A. A function f that maps domain A to domain B is said to be of type $A \rightarrow B$ and the set of all functions of type $A \rightarrow B$ form the domain $[A \rightarrow B]$. A function $g \equiv \lambda x\ y.\langle expr\rangle$ where x and y are of types A and B respectively and $\langle expr\rangle$ is of type C, is of type $A{\times}B \rightarrow C$. Note that the function can be curried, so $g\ a$ is a function of type $B \rightarrow C$. This implies that the type of g could equivalently be written as $(A \rightarrow B) \rightarrow C$. It is natural to allow $\rightarrow$ to associate to the left, so that a function, $\lambda x_1 x_2 \cdots x_n.\langle expr\rangle$ is of type $A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow A_n \rightarrow B$ if x_i is of type A_i and $\langle expr\rangle$ is of type B. Also, the type $A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow A_n \rightarrow B$ is equivalent to the type $A_1{\times}A_2{\times}\cdots{\times}A_n \rightarrow B$. Types are generally given in the first form to emphasize the possibility of currying.

3. The Denotational Method

In denotational semantics, programming language expressions are defined in terms of their syntactic components. That is, they are defined to *denote* mathematical objects, which are the *meanings* of the expressions. So the Icon expression $[\![\mathbf{E}_1 + \mathbf{E}_2]\!]$ is defined in terms of $[\![\mathbf{E}_1]\!]$, $[\![\mathbf{E}_2]\!]$, and the operator $[\![\ +\ ]\!]$.

The meaning of an expression in a programming language is not absolute, of course, but depends on the state of the machine when the expression is evaluated. This is handled by defining expressions as functions that act on a machine state, returning not only a value, but also another machine state for the next expression. The state of the machine is considered here to be the contents of the memory. The mathematical version of a machine memory is a store s, which is a function mapping syntactic identifiers into semantic values. So, if $[\![\mathbf{I}]\!]$ is an Icon identifier, $s[\![\mathbf{I}]\!] = v$ where v is the value stored in the program variable $[\![\mathbf{I}]\!]$. Stores are functions of type $\mathbf{Ide} \rightarrow Sv$, where **Ide** is the domain of identifiers and Sv is the domain of storable values. (**Ide** is written in boldface because it is a syntactic domain.) The set of all possible stores forms the domain, Sto, of stores. If E is a function that maps Icon expressions into their meanings. Then $E[\![\mathbf{I}]\!]$ should be equal to v. This is represented by the equation

$$E[\![\mathbf{I}]\!]s \equiv s[\![\mathbf{I}]\!] \tag{3.1}$$

which could just as well be written as

$$E[\![\mathbf{I}]\!] \equiv \lambda s.\ s[\![\mathbf{I}]\!]$$

Assignment is nearly as easy. Let

$$\mathit{update}\ v\,\mathbf{I}\,s \equiv \lambda\mathbf{I}'.\ \mathbf{if}\ \mathbf{I} = \mathbf{I}'\ \mathbf{then}\ v\ \mathrm{else}\ s\,\mathbf{I}' \tag{3.2}$$

So *update* is a function that returns a function of type $\mathbf{Ide} \rightarrow Sv$; therefore, *update* is of type $Sv \rightarrow \mathbf{Ide} \rightarrow Sto \rightarrow (\mathbf{Ide} \rightarrow Sv)$. Notice that $\mathbf{Ide} \rightarrow Sv$ is the type of Sto, so *update* is of type $Sv \rightarrow \mathbf{Ide} \rightarrow Sto \rightarrow Sto$. Given a storable value, v, an identifier, **I**, and a store, s, *update* returns another store that is the same as s except that **I** has the value v. Then

$$E[\![\mathbf{I} := \mathbf{E}]\!]s \equiv \mathit{update}(E[\![\mathbf{E}]\!]s)[\![\mathbf{I}]\!]s \tag{3.3}$$

The assignment of the value of one variable, $[\![\mathbf{I}_1]\!]$, to another variable, $[\![\mathbf{I}_2]\!]$, is represented by

$$E[\![\mathbf{I}_1 := \mathbf{I}_2]\!]s = \mathit{update}(E[\![\mathbf{I}_2]\!]s)[\![\mathbf{I}_1]\!]s$$

reduce by (3.1)

$$= \mathit{update}(s[\![\mathbf{I}_2]\!])[\![\mathbf{I}_1]\!]s$$

reduce by (3.2)

$$= \lambda\mathbf{I}.\ \mathbf{if}\ \mathbf{I} = [\![\mathbf{I}_1]\!]\ \mathbf{then}\ s[\![\mathbf{I}_2]\!]\ \mathrm{else}\ s\,\mathbf{I}$$

The statement $[\![\mathbf{E}_1 ; \mathbf{E}_2]\!]$ produces the store as changed by the evaluation of $[\![\mathbf{E}_1]\!]$ followed by the changes caused by $[\![\mathbf{E}_2]\!]$. This is written as

$$E[\![\mathbf{E}_1 ; \mathbf{E}_2]\!]s \equiv E[\![\mathbf{E}_2]\!](E[\![\mathbf{E}_1]\!]s) \tag{3.4}$$

In (3.3) and (3.4), E maps one store into another, rather than mapping a store into a value as in (3.1). This corresponds to the difference between a statement (assignment) and an expression (identifier). Since Icon has only expressions, a different form of E is needed, one that can can produce a value and a changed store at the same time.

4. Continuations

Another problem with the form of E given above is that the evaluation of $[\![E]\!]$ may fail. What value would the function $E[\![E]\!]$ return in that case? Mathematics does not have any provision for the failure of a function to produce a result, so some other method is needed. One possibility is to create a special value, *fail*, that would be returned when a function fails. Then

$$E[\![I := E]\!]s \equiv \textbf{if } E[\![E]\!] = \textit{fail} \textbf{ then } \textit{fail} \text{ else } \textit{update}(E[\![E]\!])[\![I]\!]s \qquad (4.1)$$

All functions would then have to know what *fail* means. There are three reasons not to use this approach. First, it adds a great deal of repetitive complication to the semantics. Every denotation of a compound syntactic object would have to have some version of the **if** expression in (4.1), and generators would add even more problems. Second, it is very dissimilar to Icon's expression evaluation mechanism, in which failure specifically corresponds to *not* returning a value. Since one of the major purposes of this semantics is to gain insight into the evaluation mechanism, this is clearly undesirable. Third, there are problems with extending the simple approach outlined in Section 3 to gotos, returns, coroutines, and other constructs found in other languages, that are usually handled by a type of function called a *continuation*. Continuations also serve to solve the problem of Icon's generators.

A continuation is a function that maps one store into another. But unlike $E[\![I := E]\!]$ in (3.3) or $E[\![E_1 ; E_2]\!]$ in (3.4), which mapped the current store into the *next* store, a continuation maps the current store into the *final* store of the entire program. So a continuation must somehow contain the effect of the entire rest of the program. In this sense, a continuation may be thought of as a *context* in which an expression is evaluated. The function $E[\![E]\!]$ applies its continuation to whatever store is produced by evaluation of $[\![E]\!]$.

Let E be a semantic function that takes a statement (not an expression), a continuation c, and a store s, and returns the meaning of executing the statement in the context of c with the store s. Let V be a semantic function that returns the value of an expression. Then

$$E[\![I := E]\!]c\ s \equiv c(\textit{update}\ (V[\![E]\!])\ [\![I]\!]\ s) \qquad (4.2)$$

This definition says that if $[\![I := E]\!]$ is executed in a context in which the rest of the program is c, the result is equivalent to executing c with the store produced by doing the assignment. "Executing" c means applying it to a store. Notice that the continuation is applied to the store by E. *All* $E[\![E]\!]$ are defined to do something like this, so in some sense, $E[\![I := E]\!]$ is actually being passed as an argument to c. This can be made clearer using $[\![E_1 ; E_2]\!]$ as an example.

In $[\![E_1 ; E_2]\!]$ the continuation of $[\![E_1]\!]$ is a function that executes $E[\![E_2]\!]$ in the store produced from the execution of $[\![E_1]\!]$, in the context of $[\![E_1 ; E_2]\!]$.

$$E[\![E_1 ; E_2]\!]c\ s \equiv E[\![E_1]\!]c_1 s$$

$$\text{where } c_1 s_1 \equiv E[\![E_2]\!]c\ s_1$$

or

$$E[\![E_1 ; E_2]\!]c\ s \equiv E[\![E_1]\!](\lambda s_1 . E[\![E_2]\!]c\ s_1)s \qquad (4.3)$$

Compare this to (3.4), in which the order of $[\![E_1]\!]$ and $[\![E_2]\!]$ is reversed. In continuation semantics, the reversal is implicit in the definition of E, since the evaluation of $E[\![E_1]\!]$ will apply c_1 to the store produced from execution of $[\![E_1]\!]$. If $[\![\ ;\ ]\!]$ associates to the left, then

$$E[\![E_1 ; E_2 ; E_3 ; \cdots ; E_n]\!]c\ s = E[\![E_1]\!]c_1 s$$

$$\text{where } c_1 s_1 \equiv E[\![E_2]\!]c_2 s_1$$

where c_2 is the continuation of $[\![E_1 ; E_2]\!]$. The continuation of $[\![E_1 ; E_2]\!]$ is $E[\![E_3 ; \cdots ; E_n]\!]c$, so

$$E[\![E_1;E_2;E_3;\cdots;E_n]\!]c\ s = E[\![E_1]\!]c_1 s$$

$$\text{where } c_1 s_1 \equiv E[\![E_2]\!]c_2 s_1$$

$$\text{where } c_2 s_2 \equiv E[\![E_3]\!]c_3 s_2$$

...

$$\text{where } c_{n-1}s_{n-1} \equiv E[\![E_n]\!]c\ s_{n-1}$$

In order to find the meaning of $[\![E_1;E_2;E_3;\cdots;E_n]\!]$, it is necessary to first find the meaning of $[\![E_n]\!]$, then of $[\![E_{n-1}]\!]$, etc., going backward and "unwinding" the equation. The continuation c_1 being defined in the where clauses is *called* (applied to the original store s) to produce the final store.

Another way to think about continuations is as a list of functions "strung together", where the store is passed from one continuation to the next. Each continuation evaluates the expression it represents, then changes the store accordingly. But even though continuations appear to be chained together, notice that c_2 is in fact defined as a *part* of c_1, so that c_1 *contains* c_2 as part of its definition. In general c_i is defined as a part of c_{i-1} so each continuation *contains* all of the subsequent continuations. Since c_n calls c, and since c is an encoding of the rest of the program, it follows that c_n was the previous encoding of the rest of the program, and the same for each other c_i.

An expression must return a result, so its continuation must "pass on" the result to the rest of the program. To do this, the continuation must take the result as an argument. Let k be a function that takes as arguments an expressible value (a value that may be expressed in the language) and a store, and produces a store. The type of k is $Ev \rightarrow Sto \rightarrow Sto$. As an example of the use of k, the addition operator is defined as

$$E[\![E_1 + E_2]\!]k\ s \equiv E[\![E_1]\!]k_1 s$$

$$\text{where } k_1 v_1 s_1 \equiv E[\![E_2]\!]k_2 s_1$$

$$\text{where } k_2 v_2 s_2 \equiv k\ (v_1 + v_2)\, s_2$$

This says that the continuation of $[\![E_1]\!]$ in the context of $[\![E_1 + E_2]\!]$, is a function (k_1) that takes as arguments the value of $[\![E_1]\!]$ (v_1) and the store produced by the evaluation of $[\![E_1]\!]$ (s_1) and applies the meaning of $[\![E_2]\!]$ (meanings are functions and can therefore be applied to arguments) to a continuation, k_2. The meaning of $[\![E_2]\!]$ applies k_2 to the value and store produced by the evaluation of $[\![E_2]\!]$. The continuation k_2 applies the sum of the two values to the continuation of the outer expression, k. In lambda notation, the definition is

$$E[\![E_1 + E_2]\!]k\ s \equiv E[\![E_1]\!](\lambda v_1 s_1 . E[\![E_2]\!](\lambda v_2 s_2 . k\ (v_1 + v_2)\, s_2) s_1) s$$

5. Denotations in Icon

Tennent [8] described the semantics of SNOBOL4 in terms of a pair of continuations instead of a single continuation. This is necessary because expressions in SNOBOL4 can effect the flow of control of the program by failing or succeeding. So a failure of an expression causes the failure continuation to be called, and a success causes the success continuation to be called. In Icon, E is a semantic function that maps Icon expressions into their denotations. Denotations are functions that map an expression continuation k, a failure continuation c, and a store s into the final store of the program. So E is of type $\mathbf{Exp} \rightarrow Ec \rightarrow Fc \rightarrow Sto \rightarrow Sto$. The remainder of this report is dedicated to defining E for expressions in Icon.

The two kinds of continuations result from the fact that Icon expressions may fail or succeed in a manner similar to SNOBOL4. In Icon, however, success or failure corresponds simply to producing a result or not producing a result. If an expression does not produce a value, then its denotation simply calls the failure continuation. It obviously does not need to pass on a value like the k-continuation described, so the only argument a failure continuation needs is a store. A failure continuation, then, is a function mapping one store into another, and is of type $Sto \rightarrow Sto$. Failure continuations are represented by the variable name, c.

If the expression produces a value, then its denotation calls the expression continuation. An expression continuation needs both a value and a store as arguments, like the k-continuation described above, but it needs something else as well. Suppose the part of the program represented by the continuation fails. Any expression can fail, and the failure must be handled by the expression that called the continuation. This is most easily controlled by passing to the expression another continuation that encodes the "rest of the program" on a failure.

This continuation is referred to as a resumption continuation, because it is called to resume a pre-expression. Note that no results are passed *to* a resumption continuation; it is called by an expression that has failed, so there is no value to be passed to it. Likewise, an expression cannot resume a post-expression (it *can* re-evaluate a post-expression), so a resumption continuation does not have to be passed a resumption continuation of its own. This leaves only the store as an argument, so a resumption continuation is of type $Sto \rightarrow Sto$, the same type as a failure continuation. This makes sense, since it is the continuation called by post-expressions when they fail. Finally an expression continuation is a function that takes a value, a failure continuation, and a store, and produces a store. It is of type $Ev \rightarrow Fc \rightarrow Sto \rightarrow Sto$, and is represented by the variable name, k. Notice the k can be curried, giving a type: $Ev \times Fc \rightarrow (Sto \rightarrow Sto)$ or $Ev \times Fc \rightarrow Fc$.

$E[\![\mathbf{E}]\!] k\ c\ s$ is the meaning of the Icon expression $[\![\mathbf{E}]\!]$ in a context in which the rest of the program is k if $[\![\mathbf{E}]\!]$ produces a value, or c if $[\![\mathbf{E}]\!]$ does not produce a value, and where the current store is s. E can be curried so that it has type: $\mathbf{Exp} \times Ec \times Fc \rightarrow (Sto \rightarrow Sto)$ or $\mathbf{Exp} \times Ec \times Fc \rightarrow Fc$.

6. The T Version of Icon's Continuation Semantics

Rather than a function, T makes use a more convenient association list to represent the store. Since there is no store to pass around, failure continuations are functions of no arguments, which return no useful value. The initial failure continuation is

```
(define (cstart) 'fails)
```

which returns the symbol, FAILS.

The function E is conveniently defined in its curried form, (E exp k c) returns a failure continuation, which must be called to produce the result of exp. Expression continuations are also defined in their curried form: (k v c) produces a failure continuation. This convention allows (E exp k c) to return c, or to return the result of evaluating (k v1 c1) for some v1 and k1. The starting expression continuation is

```
(define (kstart v c) (lambda () (list 'returns v)))
```

All Icon expressions are given in a lisp-like prefix form. Where symbols representing Icon operators and reserved words are written as lisp function names. Sometimes, this requires use of a backslash to prevent T from treating characters as read macros. Unary operators that use the same symbol as some infix operator, are prefixed with a "U". For example, $[\![-\mathrm{a}]\!]$ is written (U- a) and $[\![/\mathrm{a}]\!]$ is written (U/ a) but $[\![!\mathrm{a}]\!]$ is written (! a).

To get the result of an Icon expression, type (icon exp), without quoting exp. (icon ...) is a macro:

```
(define-syntax (icon exp) `(pp ((E ',exp kstart cstart))))
```

There are actually two variables used to make a complete Icon store. Global-store is a simple association list, which is used to hold the values of global variables. Another macro is provided to install variables in this list:

```
(define-syntax (global . varlist)
  '(block (append! global-store
                   (map (lambda (a) (cons a '&null)) ',varlist)) 'DEFINED))
```

The other variable is Icon-store, which is actually a stack of association lists, used to represent local stores. There is one empty list on the stack at the beginning. Local variables are produced by referencing them.

Icon stores procedures as global variables, and the user can create one by using the macro:

```
(define-syntax (procedure name args . body)
  '(block (push global-store '(,name procedure ,args (; ,@body)))
        'DEFINED))
```

Notice that both 'global and 'procedure are defined to return the symbol, 'DEFINED. This is to prevent them from returning (and writting out to the screen) the entire list, global-store.

Three more macros are defined to make it easy to create new expression and failure continuations:

```
(define-syntax (newk v c . exp) '(lambda (,v ,c) ,@exp))

;; exp has to be called in newc to produce an object of the right type
(define-syntax (newc . exp) '(lambda () ((block ,@exp))))

(define-syntax (recurse args . body) '(labels ((,args ,@body)) ,(car args)))
```

Newk returns an expression continuation which produces the evaluation of exp. Newc produces a failure continuation which evaluates exp and then calls the last value produced. This means that exp must produce a failure continuation. Recurse is not actually specific to continuations, it simply produces a recursive function, where args is the form of a function call, and the function variable (car args) represents the same thing at all points within the form.

Several type conversion functions had to be written to make T types work with Icon types, these functions are not discussed further.

The rest of this report describes the function, E, along with auxiliary functions to make E interpret Icon expressions. The following macros are defined to help break expressions into parts within E:

```
(define-syntax (expr-type ls) '(car ,ls))
(define-syntax (arg1 ls) '(cadr ,ls))
(define-syntax (arg2 ls) '(caddr ,ls))
(define-syntax (arg3 ls) '(cadddr ,ls))
(define-syntax (arglist ls) '(cdr ,ls))
```

Notice that the first argument is actually the *second* element of the expression.

7. Elementary Expressions

Elementary expressions are expressions without components. They include [[**&fail**]], [[**&null**]], identifiers, and literals, and are defined in the outer 'cond expression of E. The expression [[**&fail**]] is the simplest expression in Icon, in terms of its denotational description. It does not produce any value, so it ignores the expression continuation. All it does is apply the failure continuation to the store:

```
(define (E exp k c)
  (cond
        ((eq? '&fail exp) c)
```

In this expression, [[&fail]] denotes a function that takes the following arguments: an expression continuation *k*, which is the "rest of the program" after the expression, *if* the expression produces a value (which it does not), and a failure continuation *c*, which is the rest of the program if the expression does *not* produce a value. The result of applying the denotation to the arguments is the failure continuation. This means that the expression executes the part of the program that was supposed to be executed on failure.

The next simplest expression is [[**&null**]], because all it does is return the value 'null. For an expression to return a value, means that the expression's expression continuation is applied to that value:

The resumption continuation that [[**&null**]] passes on to its post-expressions is the failure continuation, *c*, it received from its pre-expression. That is, [[**&null**]] does not produce another value if it is resumed, so a resumption causes a failure to the original failure continuation, *c*.

The value of an identifier is found in the store. To make use of T's association lists, a function is defined to get the value of the symbol as a variable. Since update (as defined earlier) has a somewhat different meaning, this function is given a different name:

```
(define (value id)
  (let ((x (assq id (car icon-store))))
     (if x (cdr x)
        (let ((x (assq id global-store)))
            (if x (cdr x) '&null)))))
```

Value looks in the local store first (the first element of icon-store), and if the variable is not there, it looks in the global store. If the symbol is not found in either location, it is an un-initialized local variable, and its value in '&null. As for [[**&null**]], the value produced is given to k as an argument:

```
((symbol? exp) (k (value exp) c))
```

The similarity between this and [[**&null**]] arises because identifiers and [[**&null**]] are both simple expressions that return exactly one value. The only difference in semantics is the value. [[**&null**]] always returns the value *null*, and an identifier must find its value in the store.

Icon literals are similar enough to T literals to use T's value directly:

```
((atom? exp) (k exp c))
```

All other legal atomic expressions have been tried by this point. There are several other atomic expressions that are related to control stuctures, they will be considered with their corresponding control structures.

8. Unary Operators

If exp is not atomic, it must be a list. The rest of E is based on the value of the first element of exp, its expr-type. The first operator is the unary [[/]] operator which produces *null* if its operand evaluates to *null*, otherwise it fails:

```
(t (case (expr-type exp)
    ((U/)
      (E (arg1 exp)
        (newk v1 c1 (if (eq? v1 '&null) (k v1 c1) c1))
        c))
```

This definition produces c if exp fails, otherwise it produces the result of newk. Newk produces v1, is it is &null, otherwise, it calls v1's expression continuation. C1 may produce another result, or it may simply call c.

Next is the ⟦ \ ⟧ operator, which returns the value of its operand if it is not *null*, and fails if it is *null*. It seems reasonable that the definition should be in some sense the converse of the definition of ⟦ / ⟧:

```
((U\)
  (E (arg1 exp)
    (newk v1 c1 (if (neq? v1 '&null) (k v1 c1) c1))
    c))
```

Notice that 'eq? has been replaced with 'neq?.

9. Monogenic Unary Operators

The unary operators, ⟦+−?~*.⟧ all produce exactly one result for each result produced by their operand. Such operators are called *monogenic*. These are all grouped together into a function, UOP:

```
((U+ U- U? U~ U* U.)
  (E (arg1 exp)
    (newk v1 c1 (k (UOP (expr-type exp) v1) c1))
    c))
```

Exp is evaluated with an expression continuation that applies the appropriate function to its value, then passes the result to k. UOP has a simple case-by-case definition:

```
(define (UOP op v)
  (case op
    ((U+) (->number v))
    ((U-) (- (->number v)))
    ((U?) (error "random operation not handled~%"))
    ((U~) (->cset v))
    ((U*) (string-length (->string v)))
    ((U.) (if (symbol? v) (value v) v))))
```

The conversion functions cause error exits if conversion is not possible.

The ⟦¬E⟧ operator, produces &null if ⟦E⟧ fails, and fails is ⟦E⟧ produces a value. This is done very simply by forcing c to act as the expression continuation, and forcing k to act as the failure continuation to the operand:

```
((not)
  (E (arg1 exp)
    (newk v1 c1 c)
    (newc (k '&null c))))
```

10. Element Generation

The unary operator ⟦ ! ⟧ presents more difficulties than many other operators, because it may produce multiple values. This means that when *E*⟦ !E⟧ calls its expression continuation, it must provide a resumption continuation that will produce the next element in the sequence, calling the expression continuation again.

```
((!)
    (E (arg1 exp) (element-gen k) c))
```

All of the work has been put into the function 'element-gen.

```
(define (element-gen k)
  (recurse (k1 v1 c1)
      (set v1 (->string v1))
      (if (string-empty? v1) c1
        (k (string-slice v1 0 1)
            (newc (k1 (string-tail v1) c1))))))
```

This function returns an expression continuation, k1, that is defined recursively to apply elements of its value to k. First k1 converts its argument to a string. Then if there are characters, it applies k to the first element, passing a resumption continuation which does the same for the tail of the string.

11. Statement Separator

The semicolon in Icon can be thought of as an infix control structure with certain syntactic restrictions on where it may appear (only in braces or the outer level of a function). It evaluates its left operand, then produces the result of evaluating its right operand

```
((;)
    (bounded-eval (arglist exp) k c))
```

Bounded-eval evaluates all of the arguments, making k and c be the continuations of the last arguments:

```
(define (bounded-eval ls k c)
  (case (length ls)
      ((0) (k '&null c))
      ((1) (E (car ls) k c))
      (else
        (E (car ls)
            (newk v1 c1 (bounded-eval (cdr ls) k c))
            (newc (bounded-eval (cdr ls) k c))))))
```

An empty list evaluates to &null. A list with one expression is the same as the expression alone. Otherwise, each element is evaluated with both continuations leading to evaluation of the next element.

12. Alternation

The alternation control structure, infix [[|]], produces the result sequence of its left operand followed by the result sequence of its right operand. In terms of continuations, the expression continuation is applied to the evaluation of the left argument, with a resumption continuation that evaluates the right argument, passing the result to the expression continuation.

```
((|)
   (E (arg1 exp)
     k
     (newc (E (arg2 exp) k c))))
```

13. Infix Operators

A monogenic infix operator is an operator that returns exactly one result for each pair of results produced by cross-product evaluation of its operands. A function BOP, similar to UOP for unary operators, is required to map monogenic infix operators into binary functions. In direct correspondence with unary monogenic operators:

```
((+ - * / % ^ ++ -- ** || & .)
   (E (arg1 exp)
     (newk v1 c1
        (E (arg2 exp)
          (newk v2 c2
             (k (BOP (expr-type exp) v1 v2) c2))
          c1))
     c))
```

Notice that if (arg1 exp) produces a result, it passes it to newk as the argument v1. Likewise, v2 is result of (arg2 exp). BOP produces the result of combining the two operands.

A conditional infix operator produces one or zero results for each pair of results of its operands. COP is a predicate mapping Icon conditional operators and two values into t or nil, to control which of the two continuations is called.

```
((< <= = >= > ~= << <<= == >>= >> ~== === ~===)
   (E (arg1 exp)
     (newk v1 c1
        (E (arg2 exp)
          (newk v2 c2
             (if (COP (expr-type exp) v1 v2)
               (k v2 c2) c2))
          c1))
     c))
```

Just as for monogenic operators, v1 is the result of (arg1 exp) and v2 is the result of (arg2 exp). The difference between this definition and that for monogenic infix operators is in the second newk, k2. In k2, monogenic operators always produce a result (by calling the expression continuation), but conditional operators test some relation between v1 and v2, then call either the expression continuation (with v2) or the failure continuation. As for monogenic operators, the failure continuation attempts to resume the second operand, which supposedly tries to resume the first operand if there are no more values.

14. Integer Generation

The Icon expression ⟦ E_1 to E_2 ⟧ produces the sequence of integers from ⟦ E_1 ⟧ to ⟦ E_2 ⟧. Like infix and conditional operators, this operator needs to evaluate two arguments, so it calls E twice:

```
((to)
   (E (arg1 exp)
      (newk v1 c1
         (set v1 (->int v1))
         (E (arg2 exp)
            (newk v2 c2
               (integer-gen k c2 v1 (->int v2) 1))
            c1))
      c))
```

Again, v1 and v2 are the results of (arg1 exp) and (arg2 exp). All k2 does is call integer-gen.

```
(define (integer-gen k c v1 v2 v3)
  (set v3 (->int v3))
  (let* ((past? (cond ((>0? v3) (set past? >))
                      ((<0? v3) (set past? <))
                      (t (error "value error, by 0~%")))))
     ((recurse (k1 v c1)
        (if (past? v v2) c1
            (k v (newc (k1 (+ v3 v) c1)))))
      v1 c)))
```

Like element-gen, integer-gen returns an expression continuation defined recursively to call k with a resumption continuation that produces further values. 'Past? is used for the predicate, because the sign of v3 determines wether the function is counting up or down.

Extending the definition above to ⟦ E_1 to E_2 by E_3 ⟧ only involves getting the result of ⟦ E_3 ⟧ and incrementing by that value instead of 1.

```
((toby)
   (E (arg1 exp)
      (newk v1 c1
         (set v1 (->int v1))
         (E (arg2 exp)
            (newk v2 c2
               (set v2 (->int v2))
               (E (arg3 exp)
                  (newk v3 c3 (integer-gen k c3 v1 v2 v3))
                  c2))
            c1))
      c))
```

15. Assignment

To simplify assignment to its essentials, assume that all assignments are of the form [[I := E]], where [[I]] is an identifier. Then

```
((:=)
   (E (arg2 exp)
      (newk v1 c1 (k (store-v v1 (arg1 exp)) c1))
      c))
```

Store-v is a function similar to *update* except that it works on T data structures.

```
(define (store-v v id)
  (cond
     ((assq id (car icon-store))
        (push (car icon-store) (cons id v)))
     ((assq id global-store)
        (push global-store (cons id v)))
     (t (push (car icon-store) (cons id v))))
  v)
```

Store-v looks in the same sequence of stores for the variable.

A reversible assignment, [[I <- E]], is the same as an assignment unless it is resumed. If resumed, the expression has to replace the old value of the variable and resume [[E]]

```
((<-)
   (let* ((id (arg1 exp)) (oldv (value id)))
     (E (arg2 exp)
        (newk v1 c1 (k (store-v v1 id)
                       (newc (store-v oldv id) c1)))
        c)))
```

The failure continuation of k is newc, so any attempt to resume the expression causes the old value to be restored.

16. Control Structures

The [[if E_1 then E_2]] control structure has a very simple interpretation: if [[E_1]] produces a value then produce the result sequence of [[E_2]], otherwise fail.

```
((if)
   (E (arg1 exp)
      (newk v1 c1 (E (arg2 exp) k c))
      c))
```

Adding the else-part is as easy as changing the failure continuation of E[[E_1]] to something that evaluates the else-part. Notice that the then-part does not use either argument of its expression continuation, so it is not difficult to adapt the function to a failure continuation by simply dropping those two arguments:

```
((ifelse)
    (E (arg1 exp)
       (newk v1 c1 (E (arg2 exp) k c))
       (newc (E (arg3 exp) k c))))
```

The expression [[every E]] produces every result of [[E]]. In other words, it calls [[E]]s resumption continuation every time [[E]] produces a value

```
((every)
    (let ((c (exit-loop c)))
      (E (arg1 exp)
         (set-up-loop k c (newk v1 c1 c1))
         c)))
```

Set-up-loop pushes three continuations onto the stack, icon-loop-stack for use with [[break]] and [[next]] expressions, then returns its last argument.

The expression [[while E]] evaluates [[E]] once. If it produces a value, then it is re-evaluated (*not* resumed). This implies a recursive function:

```
((while)
    (let ((c (exit-loop c)))
      (E (arg1 exp)
         (set-up-loop k c
            (recurse (k1 v1 c1) (E (arg1 exp) k1 c)))
         c)))
```

The failure continuation c1 passed to k1 is ignored. If k1 were to simply pass on its own resumption continuation, then when an iteration failed, it would resume (arg1 exp) from the previous iteration. This is an interesting idea, but is not Icon.

It is convenient to think of [[do]] as an infix control structure with the syntactic restriction that it may only appear as the immediate sub-expression of a while or every loop. It can then be given a semantics that describes its effect. In both types of loop, the expression [[E_1 do E_2]] evaluates [[E_1]]. If it produces a value, then [[E_2]] is evaluated and the value of [[E_1]] is returned. If [[E_1]] does not produce a value, then the expression fails:

```
((do)
    (E (arg1 exp)
      (newk v1 c1 (E (arg2 exp)
                     (newk v2 c2 (k v1 c1))
                     (newc (k v1 c1))))
      c))
```

The expression [[break]] exits the innermost enclosing loop, producing the value &null. It is an elementary expression:

```
((eq? 'break exp) (k-break '&null c))
```

K-break pops the last set of continuations pushed onto icon-loop-stack, calling the k-break continuation with values v and c passed to it. In this case v is &null, and c doesn't matter because it will not be called.

As a unary operator, [[breakE]] causes the value of [[E]] to take the place of the loop. If [[E]] produces a value, it should pass it to k-break, otherwise, it should call the c continuation on icon-loop-stack:

```
((break)
   (E (arg1 exp) k-break c-break))
```

C-break just calls the c break continuation, it does not pop the stack, because the continuations pushed onto the stack are defined to pop the stack:

```
(define (c-break) (cadar icon-loop-stack))
```

The expressions [[next]] restarts the loop at the top. This currently only works for /fIwhile/fP-loops because there is no clear way to get a hold of the correct continuation to restart an *every*-loop:

```
((eq? 'next exp) (k-next))
```

Where k-next simply calls the next-continuation on the loop stack:

```
(define (k-next) ((caddar icon-loop-stack) nil nil))
```

17. Procedure Calling

The arguments to procedures are called with mutual evaluation of the actual parameters. In the degenerate case, there is no function, and the last value is returned, this is done by the comma operator:

```
((,)
   (mutual-eval
     (arglist exp)
     (newk v1 c1 (k (car v1) c1))
     c
     '(&null)))
```

The newk returns the car of the list, v1. The list '(&null) will be the tail of the list, v1, so if the arglist is empty, the value &null will be returned.

Mutual-eval is similar to bounded-eval, except that that a list is kept of all results, and the failure continuations retry previous expressions:

```
(define (mutual-eval ls k c v)
   (if (null? ls) (k v c)
      (E (car ls)
        (newk v1 c1 (mutual-eval (cdr ls) k c1 (cons v1 v)))
        c)))
```

If ls is nil, the initial value is returned. Otherwise, the first expression in ls is evaluated with an expression continuation which will evaluate the rest of the expressions in ls. The failure continuation resumes the previous expression in ls.

The only implemented built-in function is *write*:

```
((write)
   (mutual-eval (arglist exp)
      (newk v1 c1 (k (or (icon-write (reverse v1)) "") c1))
      c
      nil))
```

Since icon-write knows about T lists, the tail of the list is nil. If the result of icon-write is nil, then the argument list to ⟦ write ⟧ was empty, so an empty string is produced.

```
(define (icon-write ls)
   (if (null? ls)
       (block (format t "~%") nil)
       (let ((v (if (eq? '&null (car ls)) "" (->string (car ls)))))
          (format t "~A" v)
          (or (icon-write (cdr ls)) v))))
```

This function returns the last element of ls, writting out all elements.

If (expr-type exp) is none of the above operators, it must be a user-defined function:

```
(else
   (let ((p (value (expr-type exp))))
      (if (and (list? p) (eq? 'procedure (car p)))
          (mutual-eval
             (arglist exp) (proc-caller p k) c nil)
          (error "Unknown Icon operator~%")))))))
```

P is defined to be the value of (expr-type exp) as a variable. If it is a list, and the first element of the list is the symbol, executed by doing a mutual-eval of the arguments, using (proc-caller p k) as the expression continuation. Proc-caller returns an expression continuation which sets up a stack of return continuations, similar to icon-loop-stack:

```
(lset icon-return-stack ())

(define (proc-caller p k)
   (lambda (v c)
       (push icon-store (bind-formal-parameters (cadr p) (reverse v)))
       (push icon-return-stack (cons k c))
       (E (caddr p) (kfail c) (cfail c))))

(define (kfail c) (lambda (v1 c1) (pop icon-store) (pop icon-return-stack) c))

(define (cfail c) (lambda () (pop icon-store) (pop icon-return-stack) (c)))
```

Kfail and Cfail are continuations to pop icon-return-stack and icon-store, and fail the procedure if it falls off the end. Both call the failure continuation passed to the procedure call.

The icon expressions, ⟦ **return** ⟧ ⟦ **suspend** ⟧ and ⟦ **fail** ⟧ cause a return from a procedure, suspending of a procedure, or failing a procedure respectively. The first two may be unary operators, or elementary expressions, in which case, they produce &null:

```
((eq? 'fail exp) (pop-c-fail))
((eq? 'return exp) ((get-k-return) '&null (pop-c-fail)))
((eq? 'suspend exp) ((pop-k-suspend) '&null c))
...
   ((return)
      (E (arg1 exp) (get-k-return) (pop-c-fail)))
   ((suspend)
      (E (arg1 exp) (pop-k-suspend) c))
```

Pop-c-fail returns the procedure failure continuation, get-k-return and pop-k-suspend return the procedure expression continuation. Pop-c-fail and pop-k-suspend both pop icon-return-stack and icon-store.

```
(define (get-k-return)
  (let ((kc (car icon-return-stack)))
     (if kc
       (lambda (v c) ((car kc) v (cdr kc)))
       kstart)))
(define (pop-c-fail)
  (let ((kc (pop icon-return-stack)))
     (if kc
       (block (pop icon-store) (cdr kc))
       cstart)))
(define (pop-k-suspend)
  (let ((kc (pop icon-return-stack)) (env (car icon-store)))
     (if kc
       (let ((k (car kc)))
          (pop icon-store)
          (lambda (v c)
            (k v
               (lambda ()
                 (push icon-return-stack kc)
                 (push icon-store env)
                 (c)))))
       kstart)))
```

Pop-k-suspend also remembers the top elements of icon-return-stack and icon-store, and supplies a resumption continuation which will restore the two stacks before resumming.

The [[**return**]] expression without an argument does not work properly. Notice the difference between the [[**return**]] and [[**suspend**]] expressions. Return passes on the procedures failure continuation, and suspend passes on its own failure continuation.

18. Conclusions

The program described here elements a large subset of Icon expressions. Those parts of Icon that are not implemented generally straightforward extensions. There are difficulties with the limitation operator and the repeated alternation operator, because they both need some mechanism to count results. It is not clear what the best way is to count results, though one possibility is to add a *count* stack which all expressions would have to know about. It is interesting that these two control operations would require special additions to the T implementation, because they each require a special instruction in the Icon machine (the virtual machine that Icon is "compiled" into for the C implementation) [9].

The idea of implementing a continuation semantics is not new, and the problems with such implementations is well known. It is just very difficult to make such implementations efficient. No effort has been expended to make this implementation efficient, but there are several things that could be done. One possibility is to write the interpreter to produce as much lisp code as possible during an interpretation phase, then the lisp code could be compiled.

References

1. A. De Bruin, *Operational and Denotational Semantics Describing the Matching Process in SNOBOL4*, Afdeling Informatica, Mathematisch Centrum, Amsterdam, 1980.
2. A. C. Fleck and R. S. Limaye, "Formal Semantics and Abstract Properties of String Operations and Extended Formal Language Description Mechanisms", *Siam J. on Computing 12*, 1 (Feb. 1983), .
3. D. P. Friedman, C. T. Haynes, E. Kohlbecker and M. Wand, *Scheme 84 Interim Reference Manual*, Technical Report No. 153, Indiana %C Bloomington, IN, January, 1985.
4. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
5. D. A. Gudeman, *A Continuation Semantics For Icon Expressions*, The Univ. of Arizona Tech. Rep. 86-15, The Univ. of Arizona, Tucson, AZ, Apr. 1986.
6. P. J. Landin, "The Next 700 Programming Languages", *Comm. ACM 9*, 3 (1966), .
7. J. E. Stoy, *Denotational semantics: The Scott-Strachey Approach To Programming Language Theory.*, MIT Press, Cambridge, 1977.
8. R. D. Tennent, *Mathematical Semantics of SNOBOL4*, Technical Report No. 73-16, Queen's University, Kingston, Ontario, 1973.
9. S. B. Wampler, *Control Mechanisms for Generators in Icon*, The Univ. of Arizona Tech. Rep. 81-18, The Univ. of Arizona, Tucson, AZ, Dec. 1981.

Appendix B — T Source Code

```
(herald E)
;;; definitions needed to run ticon
(define-syntax (icon exp) '(pp ((E ',exp kstart cstart))))
(define (kstart v c) (lambda () (list 'returns v)))
(define (cstart) 'fails)
(define-syntax (global . varlist)
   '(block (append! global-store
              (map (lambda (a) (cons a '&null)) ',varlist)) 'DEFINED))
(define-syntax (procedure name args . body)
   '(block (push global-store '(,name procedure ,args (\; ,@body)))
        'DEFINED))

;;; new continuation forms
(define-syntax (newk v c . exp) '(lambda (,v ,c) ,@exp))
;; exp has to be called in newc to produce an object of the right type
(define-syntax (newc . exp) '(lambda () ((block ,@exp))))
(define-syntax (recurse args . body) '(labels ((,args ,@body)) ,(car args)))

;;; definitions for handling an implicit store
;(define icon-store (make-empty-locale 'icon-locale)) doesn't work
(lset global-store '((- . -)))            ; t doesn't do "append!" correctly
(lset icon-store '(()))
(define (value id)
   (let ((x (assq id (car icon-store))))
      (if x (cdr x)
         (let ((x (assq id global-store)))
            (if x (cdr x) '&null)))))
(define (store-v v id)
   (cond
      ((assq id (car icon-store))
         (push (car icon-store) (cons id v)))
      ((assq id global-store)
         (push global-store (cons id v)))
      (t (push (car icon-store) (cons id v))))
   v)


;;; stack of "break" and "next" continuations
(lset icon-loop-stack ())
(define (set-up-loop k-break c-break k-next)
   (push icon-loop-stack (list k-break c-break k-next))
   k-next)
(define (exit-loop c)
   (newc
      (or (pop icon-loop-stack)



          (error "break or next not in a loop~%"))
```

```
(define (k-break v c)
  ((or (car (pop icon-loop-stack))
       (error "break not in loop~%"))
       v c))
;;the c-break continuation is already defined to pop the stack
(define (c-break) (cadar icon-loop-stack))
;;the k-next continuation is actually of type K.  It must be evaluated,
;; and must not pop the stack.
(define (k-next) ((caddar icon-loop-stack) nil nil))

;;; "return", "suspend" and "fail" continuations for procedure calling
(lset icon-return-stack ())
(define (kfail c) (lambda (v1 c1) (pop icon-store) (pop icon-return-stack) c))
(define (cfail c) (lambda () (pop icon-store) (pop icon-return-stack) (c)))
(define (get-k-return)
  (let ((kc (car icon-return-stack)))
     (if kc
        (lambda (v c) ((car kc) v (cdr kc)))
        kstart)))
(define (pop-c-fail)
  (let ((kc (pop icon-return-stack)))
     (if kc
        (block (pop icon-store) (cdr kc))
        cstart)))
(define (pop-k-suspend)
  (let ((kc (pop icon-return-stack)) (env (car icon-store)))
     (if kc
        (let ((k (car kc)))
           (pop icon-store)
           (lambda (v c)
              (k v
                 (lambda ()
                    (push icon-return-stack kc)
                    (push icon-store env)
                    (c)))))
        kstart)))

;;; type conversion functions
(define (->number n)
  (cond
     ((number? n) n)
     ((string? n)
        (let* ((ls (read-objects-from-string n)) (m (car ls)))
           (if (or (cdr ls) (not (number? m)))
              (error "~S not a number~%" n)
              m)))
     (t (error "~S not a number" n))))
(define (->real n) (->float (->number n)))
(define (->int n) (->integer (->number n)))
(define (->string s)
  (cond
     ((string? s) s)
     ((number? s) (format nil "~S" s))
     (t (error "~S not a string~%" s))))
(define (->cset c) (error "csets not implemented~%"))

;;; definitions for "parsing" expressions
(define-syntax (expr-type ls) `(car ,ls))
(define-syntax (arg1 ls) `(cadr ,ls))
(define-syntax (arg2 ls) `(caddr ,ls))
(define-syntax (arg3 ls) `(cadddr ,ls))
(define-syntax (arglist ls) `(cdr ,ls))
```

```
;; the program when exp in is evaluated with continuations k and c
;; k is of type V x C -> C and c is of type C.
(define (E exp k c)
  (cond
     ((eq? '&fail exp) c)
     ((eq? '&null exp) (k '&null c))
     ((eq? 'break exp) (k-break '&null c))
     ((eq? 'next  exp) (k-next))
     ((eq? 'fail  exp) (pop-c-fail))
     ((eq? 'return exp) ((get-k-return) '&null (pop-c-fail)))
     ((eq? 'suspend exp) ((pop-k-suspend) '&null c))
     ((symbol? exp) (k (value exp) c))
     ((atom? exp) (k exp c))
     (t (case (expr-type exp)
          ((U/)
             (E  (arg1 exp)
               (newk v1 c1 (if (eq? v1 '&null) (k v1 c1) c1))
               c))
          ((U\\)
             (E  (arg1 exp)
               (newk v1 c1 (if (neq? v1 '&null) (k v1 c1) c1))
               c))
          ((U+ U- U? U~ U* U.)
             (E  (arg1 exp)
               (newk v1 c1 (k (UOP (expr-type exp) v1) c1))
               c))
          ((not)
             (E  (arg1 exp)
               (newk v1 c1 c)
               (newc (k '&null c))))
          ((!)
             (E  (arg1 exp) (element-gen k) c))
          ((break)
             (E  (arg1 exp) k-break c-break))
          ((\;)
             (bounded-eval (arglist exp) k c))
          ((|)
             (E  (arg1 exp)
               k
               (newc (E (arg2 exp) k c))))
          ((+ - * / % ^ ++ -- ** || & \)
             (E  (arg1 exp)
               (newk v1 c1
                    (E  (arg2 exp)
                      (newk v2 c2
                           (k (BOP (expr-type exp) v1 v2) c2))
                      c1))
               c))
          ((< <= = >= > ~= << <<= == >>= >> ~== === ~===)
             (E (arg1 exp)
               (newk v1 c1
                    (E (arg2 exp)
                      (newk v2 c2
                           (if (COP (expr-type exp) v1 v2)
                              (k v2 c2) c2))
                      c1))
               c))
          ((to)
             (E  (arg1 exp)
               (newk v1 c1
                    (set v1 (->int v1))
                    (E  (arg2 exp)
                      (newk v2 c2
                c1))
          c))
     ((toby)
          (E  (arg1 exp)
            (newk v1 c1
                (set v1 (->int v1))
                (E  (arg2 exp)
                  (newk v2 c2
                      (set v2 (->int v2))
                      (E  (arg3 exp)
                        (newk v3 c3 (integer-gen k c3 v1 v2 v3))
                        c2))
                  c1))
            c))
     ((:=)
          (E  (arg2 exp)
            (newk v1 c1 (k (store-v v1 (arg1 exp)) c1))
            c))
     ((<-)
          (let* ((id (arg1 exp)) (oldv (value id)))
            (E  (arg2 exp)
                (newk v1 c1 (k  (store-v v1 id)
                                (newc (store-v oldv id) c1)))
                c)))
     ((if)
          (E  (arg1 exp)
            (newk v1 c1 (E (arg2 exp) k c))
            c))
     ((ifelse)
          (E  (arg1 exp)
            (newk v1 c1 (E (arg2 exp) k c))
            (newc (E (arg3 exp) k c))))
     ((every)
          (let ((c (exit-loop c)))
            (E  (arg1 exp)
                (set-up-loop k c (newk v1 c1 c1))
                c)))
     ((while)
          (let ((c (exit-loop c)))
            (E  (arg1 exp)
                (set-up-loop k c
                  (recurse (k1 v1 c1) (E (arg1 exp) k1 c)))
                c)))
     ((do)
          (E  (arg1 exp)
            (newk v1 c1 (E  (arg2 exp)
                            (newk v2 c2 (k v1 c1))
                            (newc (k v1 c1))))
            c))
     ((\)
          (mutual-eval
            (arglist exp)
            (newk v1 c1 (k (car v1) c1))
            c
            '(&null)))
     ((return)
          (E  (arg1 exp) (get-k-return) (pop-c-fail)))
     ((suspend)
          (E  (arg1 exp) (pop-k-suspend) c))
     ((write)
          (mutual-eval (arglist exp)
            (newk v1 c1 (k (or (icon-write (reverse v1)) "") c1))
            c
          (else
              (let ((p (value (expr-type exp))))
                 (if (and (list? p) (eq? 'procedure (car p)))
                      (mutual-eval
                         (arglist exp) (proc-caller p k) c nil)
                      (error "Unknown Icon operator~%")))))))

;;; Evaluate unary operation and return the result.
(define (UOP op v)
   (case op
      ((U+) (->number v))
      ((U-) (- (->number v)))
      ((U?) (error "random operation not handled~%"))
      ((U~) (->cset v))
      ((U*) (string-length (->string v)))
      ((U.) (if (symbol? v) (value v) v))))

;;; Evaluate the binary operation and return the result.
(define (BOP op v1 v2)
   (case op
      ((+) (+ (->number v1) (->number v2)))
      ((-) (- (->number v1) (->number v2)))
      ((*) (* (->number v1) (->number v2)))
      ((/)
         (set v1 (->number v1))
         (set v2 (->number v2))
         (if (and (integer? v1) (integer? v2))
              (div v1 v2)
              (/ v1 v2)))
      ((%) (mod (->number v1) (->number v2)))
      ((^) (expt (->number v1) (->number v2)))
      ((++ – **) (->cset v1))
      ((||) (string-append (->string v1) (->string v2)))
      ((&) v2)))

;;; Evaluate the conditional operation and return the result.
(define (COP op v1 v2)
   (case op
      ((<)  (<  (->number v1) (->number v2)))
      ((<=) (<= (->number v1) (->number v2)))
      ((=)  (=  (->number v1) (->number v2)))
      ((>=) (>= (->number v1) (->number v2)))
      ((>)  (>  (->number v1) (->number v2)))
      ((~=) (N= (->number v1) (->number v2)))
      ((==)  (string-equal? (->string v1) (->string v2)))
      ((<< <<= == >>= >> ~==)  (error "lexical comparisons not implemented"))
      ((=== ~===) (error "general comparison not implemented"))))

(define (element-gen k)
   (recurse (k1 v1 c1)
      (set v1 (->string v1))
      (if (string-empty? v1) c1
         (k  (string-slice v1 0 1)
              (newc (k1 (string-tail v1) c1))))))

(define (integer-gen k c v1 v2 v3)
   (set v3 (->int v3))
   (let* ((past? (cond ((>0? v3) (set past? >))
                       ((<0? v3) (set past? <))
                       (t (error "value error, by 0~%"))))
        ((recurse (k1 v c1)
            (if (past? v v2) c1
                 (k v (newc (k1 (+ v3 v) c1)))))
```

```
(define (bounded-eval ls k c)
  (case (length ls)
     ((0) (k '&null c))
     ((1) (E (car ls) k c))
     (else
       (E  (car ls)
           (newk v1 c1 (bounded-eval (cdr ls) k c))
           (newc (bounded-eval (cdr ls) k c))))))

(define (mutual-eval ls k c v)
  (if (null? ls) (k v c)
     (E  (car ls)
       (newk v1 c1 (mutual-eval (cdr ls) k c1 (cons v1 v)))
       c)))

(define (icon-write ls)
  (if (null? ls)
     (block (format t "~%") nil)
     (let ((v (if (eq? '&null (car ls)) "" (->string (car ls)))))
       (format t "~A" v)
       (or (icon-write (cdr ls)) v))))

;;; return an expression continuation that takes a backwards list of
;;;  arguments as v, and calls p with the arguments bound to the formal
;;;  paramaters.  The continuation to return or suspend to is k, and
;;;  the continuation to fail to is the expression continuation's c.
(define (proc-caller p k)
  (lambda (v c)
     (push icon-store (bind-formal-parameters (cadr p) (reverse v)))
     (push icon-return-stack (cons k c))
     (E (caddr p) (kfail c) (cfail c))))

(define (bind-formal-parameters vars vals)
  (cond
     ((null? vars) nil)
     ((null? vals) (cons (cons (car vars) '&null)
                    (bind-formal-parameters (cdr vars) nil)))
     (t (cons (cons (car vars) (car vals))
          (bind-formal-parameters (cdr vars) (cdr vals))))))
```