FLECS: A Tool for Rapid Prototyping of Mechanisms in Success/Failure Based Languages

Mark W. Bailey
Janalee O'Bagy

Computer Science Report No. CS-90-35
July 1990

FLECS: A Tool for Rapid Prototyping of Mechanisms in Success/Failure Based Languages

Mark W. Bailey
mark@virginia.edu

Janalee O'Bagy

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract

Our goal is to provide a prototyping tool that facilitates the addition of new language mechanisms in success/failure based languages. By this, we mean languages that have success and failure of expressions, generators, and backtracking. We have designed an interpreter, called FLECS, for a subset of Icon, a language which exhibits the above features. FLECS is written in Scheme using a technique called continuation-passing-style (CPS). This approach allows us to extend, modify, and experiment with new language mechanisms which range from datatyping issues to general control structures.

In this report, we provide an overview of the base semantics that FLECS implements, followed by a brief description of the CPS implementation technique. FLECS's capability is then demonstrated by augmenting the base language with a general control abstraction that has been shown to be powerful in the context of traditional procedural languages. In conclusion, the semantics of this control abstraction in the presence of success and failure is discussed.

1 Introduction

Since the design of the Icon programming language in the early 1980's, newer languages have incorporated a number of language features that have become well established. These include, among others, first-class functions and a variety of scoping mechanisms. Therefore, we are interested in extending, or changing the Icon language [Gri82][GG83] to reflect these innovations in language design.

Icon is distributed with a flexible interpreter and run-time system. This flexibility is seen most notably in the ease with which one may to add new types and run-time operations [GG86] and change syntax [GW90]. However, using the Icon system to experiment with language extensions that have pervasive semantic implications, such as scoping or adding first-class procedures, requires extensive modifications to the run-time system. The Icon implementation is large, consisting of over 40,000 lines of **C**. Such modifications are therefore more easily accomplished in the context of a very high-level implementation.

Our goal is to provide a high-level implementation that facilitates rapid prototyping of new language features. In an effort to achieve this goal, we have designed an interpreter for a subset of Icon. An interpreter is a natural choice since we would like the language implementation to be quickly modifiable. Unlike other interpreters for Icon [Gri82] [OG87], our's is written in a form called continuation-passing-style [Ste76]. This describes Icon's goal-directed evaluation in terms of continuations, which in many ways is closer to its formal denotational description [Gud86]. Our

source language for the interpreter is Scheme, which provides many desirable features for our interpreter implementation such as ease of parsing, a built-in garbage collector and especially first class functions.

The implemented language is a subset of the Icon Version 8 distribution [Gri90]. Features not implemented in the language include many of the types and most of the run-time routines. In addition, for ease of implementation, the syntax of the language has been moderately altered. Appendix A provides the grammar for the syntax of the supported features.

The purpose of this report is to describe the implementation of our interpreter. Section 2 provides an overview of the Icon language. Section 3 introduces continuations, which are used in the implementation. Section 4 presents the implementation details of the interpreter. Section 5 details the addition of a very general control abstraction as a demonstration of the interpreter's flexibility. While an overview of Icon has been provided, it is assumed that the reader has a working knowledge of Scheme [Ree86].

2 An Overview of Icon

Icon is a general purpose expression-based programming language. Its predecessor, SNOBOL4, provided a backtracking mechanism during pattern matching for matching strings only. Icon has further incorporated goal-directed evaluation into the entire language with the introduction of success and failure of expressions. This integrates the traditional view of control flow, based on sequencing and iteration, with a backtracking mechanism. The language also supports user-defined generators as an extension of the usual procedural abstraction. In the next few sections, we will present a brief overview of the language. For a more complete description, see [GG83].

2.1 Success/Failure

In Icon, the evaluation of an expression may or may not produce a value. An expression that does not produce a value is said to *fail*, while an expression that produces a value is said to *succeed*. An example is

```
i < j
```

which will succeed or fail, based on the values of i and j. If the condition (i < j) is satisfied, the expression will evaluate to j. Otherwise, the expression will fail. The keyword &fail is an expression that always fails.

The use of success and failure in Icon subsumes the role of boolean values in other languages [Gri82]. As such, Icon does not support a built-in boolean type. Where boolean expressions are used to decide the direction of execution in other languages, Icon uses success and failure. Thus, expressions that may succeed or fail are known as *conditionals*, and expressions that always succeed produce a value are called *monogenic*. We may use our previous example in the context of a simple if expression:

```
if i < j then write(i) else write(j)
```

When evaluated, the lesser of the two values is written. In this case, the if control structure chooses which branch, the then or the else, should be evaluated based on the success or failure of the control clause. This example illustrates why it is said that success and failure "direct" the evaluation of expressions in Icon [Gri82].

Success and failure based evaluation is concise and expressive. For example, in other languages, exceptions are frequently signalled by returning a special value outside the "semantic" range of the function. Common examples occur with I/O where such conditions as end-of-file need to be signalled. For example, the **C** character-valued function `getchar()` returns `-1` to indicate end-of-file. In this example, the "semantic" type of the function (`unsigned char`) must be extended (to `signed int`) to provide the special value. In contrast, the Icon expression

```
read()
```

simply fails when end-of-file is encountered. Thus the success/failure scheme allows the programmer to detect an exception where no value has been computed and the control needs to be modified.

2.2 Generators

In addition to conditional and monogenic expressions, there are expressions that produce more than one result. Such expressions are called *generators*. Generators produce their values one-at-a-time, when they are required to by the surrounding context. An example generative expression is

find(exp_1, exp_2)

which attempts to find the position of the string exp_1 within the string exp_2. Since there may be multiple instances of exp_1 within exp_2, there may be multiple results for the expression. Upon evaluation of this expression, the first result is produced and the generator is *suspended* (allowing it to pick up where it left off when it is *resumed*). The remaining results are not produced until the surrounding context needs them. This need is created by the failure of a subsequent expression. Failure causes the last suspended generator to be resumed to produce its next value. Specifically, we could have

```
if(5 < find("p", "peter piper picked")) then write("yes") else write("no")
```

The result of evaluating the find expression is the value 1. This value is then used in the surrounding expression (the comparison), which in turn fails. This failure resumes find to produce its next value (7). This second value is again used in the surrounding expression; since (5 < 7) succeeds, the then clause is executed.

In a different context, it is possible for the find expression to run out of values, or fail. Such an example is

```
if(15 < find("p", "peter piper picked")) then write("yes") else write("no")
```

where the find expression is *exhausted* by producing the values 1, 7, 9, and 13. All of these cause the surrounding expression to fail. In this case, the control clause fails and the else clause is chosen.

2.3 Procedures

Icon provides a typical procedural abstraction for the programmer. Arguments are passed by value, and values may be returned to the calling procedure. Icon provides both non-generative and generative procedures.

Upon invocation, formal parameters are bound to actuals and evaluation begins at the top of the procedure definition. Evaluation continues within the procedure until it succeeds or fails. The procedure succeeds if a value is explicitly produced for the caller, while failure occurs when no value is produced upon completion of the procedure body.

2.3.1 Non-generative Procedures

Non-generative procedures may produce zero or one value upon invocation. A return value is indicted with the return keyword. An example procedure definition is:

```
procedure addndouble(x,y)
   z := x + y
   return 2 * z
end
```

An example invocation of this procedure is:

```
write(addndouble(4,5))
```

A procedure will succeed upon evaluation of a return expression that produces a value. A procedure may fail in three different ways: upon evaluation of a fail expression, upon reaching the end keyword for the current procedure, or upon failure of a return expression.

2.4 Generative Procedures

Icon only provides a few built-in generators. Therefore, a method for creating user-defined generators is available. Generators are simply procedures that use the suspend keyword rather than the return keyword to return a value to the caller. However, if the invocation of the generator is resumed by the caller, control returns to the point where the generator last suspended. The following example implements a generator that produces the result sequence `0`, `1`, `0`, `1`, ...

```
procedure flipflop()
   while(1) do {
      suspend 0
      suspend 1
   }
end
```

In this example, the procedure will return a value each time the invoking procedure resumes it. However, it is possible to create a generator that fails in the same way that a non-generative procedure fails. In the example below, the procedure will fail if it is resumed enough times to complete the while loop.

```
procedure to(start, stop)
   count := start
   while (count <= stop) do {
      suspend count
      count := count + 1
   }
end
```

2.5 Expression Evaluation

Order of expression evaluation in Icon is simple. Icon uses a default evaluation strategy in which the evaluation order is deterministic, even in the presence of backtracking. This strategy is used when procedure invocation and simple expressions are encountered. Icon also provides a number of special forms [Ree86][Dyb87] that may modify the default strategy. These special forms include if and while and are listed in appendix A of this report.

In the absence of any special forms, expressions are evaluated left-to-right. Failure causes resumption of suspended generators in LIFO (right-to-left) order. The last suspended generator *available* (see Section 2.5) is resumed upon failure of its subsequent expression. An example is the evaluation of a function's arguments

f(exp_1, exp_2, ..., exp_n)

where exp_1 will be evaluated first, followed by exp_2 and so on. Upon failure of any of the expressions, the previous expression is resumed and execution starts forward from that point. In the expression

```
f(pos := find("ells", "she sells sea shells"), 10, 15 <pos)
```

the two left-most arguments are evaluated, producing `6` and `10` respectively. The third argument is then evaluated, which fails. This causes the first argument (containing find) to be resumed since the second is a constant, not a generator. Now, the second and third arguments, in turn, are re-evaluated. This produces the values `10` and `17` respectively. Finally, the function call is made with `17`, `10`, and `17` as its arguments. It should be noted that since everything is an expression, the assignment operator (`:=`) produces the variable as its value.

The resumption of generators upon failure, as described above, provides a backtracking mechanism for Icon. This is what gives Icon its goal-directed evaluation: the goal is to succeed—produce a value—for every expression.

2.6 Generator Lifetime

During evaluation, generators compute values, suspend, and are available for future resumption. However, at points during evaluation, generators become unavailable for resumption. In other words, generators have a limited lifetime defined by the control structures that contain them. The limited lifetime of generators provides a *bounding* for the backtracking mechanism. To understand the motivation for bounding, let us look at the evaluation of the if control structure. An if expression has the usual form:

if exp_1 then exp_2 else $exp_3 exp_3$

If exp_1 succeeds, exp_2 is selected, otherwise exp_3 is selected. However, once the exp_1 succeeds or fails, none of the generators contained within exp_1 may be resumed. The motivation for bounding is to provide the "natural" semantics of traditional control structures in the presence of generators. If exp_1 in the if expression were not bounded, failure of exp_2 could resume a generator contained in exp_1, which in turn might fail, causing selection of exp_3. This obviously would not be the natural semantics of if.

Icon defines implicit "barriers" for backtracking that, once crossed, prevent the resumption of suspended generators that were evaluated before the barrier. Each of Icon's control structures defines these implicit barriers (or bounds) for the expressions that they contain. The if control structure prevents the resumption of any generator in the control clause (exp_1 above) from being resumed after the appropriate branch is chosen. Another example is

while exp_1 do exp_2

which is similar to a traditional while. exp_2 is only evaluated if exp_1 produces a result. However, once the decision of whether to evaluate exp_2 has been made, no generators in exp_1 may be resumed. Upon completion of exp_2, exp_1 is *re-evaluated*. Again, the bounding of exp_1 preserves the natural semantics of while.

In the case that there is no suspended generator to resume, an expression fails to its outer context. This, in many cases, is a compound expression. A compound expression has the form:

{exp_1; exp_2; ... ; exp_n}

In Icon, the semicolons are optional provided each expression occupies its own source line. In a compound expression, both success and failure of the i^{th} expression causes the next expression in the sequence to be evaluated. In addition, all but the final expression (exp_n) are bounded, thus limiting the control backtracking to within a single expression. Upon failure of a subsequent expression, the final expression may be resumed by the compound expression's outer context.

This control gives Icon the sequential nature of traditional languages. While function invocation follows the default evaluation strategy, control structures define backtracking boundaries, the order in which their subexpressions are evaluated, and the resulting value of the control structure. These three characteristics are what define the flow of execution.

2.7 Generative Control Structures

Icon provides *generative* control structures in addition to many of the standard control structures found in Algol-like languages. A generative control structure attempts to provide some control specific to the multi-result nature of generators. For instance

every exp_1 do exp_2

will *always* exhaust all the generators within exp_1. Furthermore, each time exp_1 produces a result, exp_2 is evaluated. Another generative control structure is

$exp_1 \backslash exp_2$

which is called limitation. This is the only Icon control structure where expressions are evaluated in right-to-left order. exp_2 is evaluated first, which must produce a numeric value. This value limits the number of results that exp_2 is allowed to produce. Used in conjunction with every, limitation can be used to produce no more than *n* results. This is particularly useful when the generator is capable of producing an infinite number of results. For example:

```
every (fibnum := fib() \ 20) do write(fibnum)
```

will write no more than the first `20` values produced by the fib generator.

3 Implementation Technique

In an effort to make the implementation of our interpreter capable of modeling a variety of control mechanisms, we have used a well-known implementation technique [All88][HF86][KH89][Ste78] known as continuation-passing-style. This technique has been shown to be flexible enough to implement a large spectrum of mechanisms from simple if expressions to catch/throw and coroutine mechanisms. This style of programming builds functions during language interpretation and so requires first-class functions in the interpreter source language. Before we see the implementation, an overview of the technique is necessary.

3.1 Continuations

While evaluating a program, an interpreter must maintain the program's execution state. Typically, this is accomplished with a program counter and program stack to store state information. This state includes control information and is known as the control context. This context may be represented as a procedure that embodies the state of the program at a given point. The procedure may subsequently be called to continue at the point previously saved. Thus the control context is known as the computation's *continuation* [Ste78].

Every expression has a continuation that represents the remaining computation of the program. A continuation may be viewed as a function of one argument. The argument is required because the current expression's value may affect the control of the remaining computation. Further information on continuations may be found in [Ste76] [Ste78][Dyb87][Hay87].

3.2 Continuation-passing Style

The interpreter described in this report is written in CPS—a style of programming in which continuations are given explicitly to direct control flow. This implementation technique is known as the continuation-passing style (CPS) [Ste76]. The use of CPS for this implementation is what will make the modeling of complex and varied control flow simpler and thereby meet the goal of this work.

CPS is not specific to interpreters and can be demonstrated by transforming a simple function to CPS. Figure 1 shows a typical recursive definition of the factorial function in Scheme. In the basis case ($n = 0$) a value is returned,

```
(define (fact n)
  (if (eq? 0 n) 1
      (* n (fact (- n 1))) ))
```

Figure 1: A standard factorial definition.

while for *n > 1* a value is computed by first calling `fact` with *n - 1*, completing the computation, and then returning the value. Figure 2shows the `fact` function after conversion to CPS. Note that the basis case is no longer a *return*, but, instead a *call* to the function `k`, which represents the continuation for `fact`. In the non-basis case, `fact` is called recursively and a new continuation—one that incorporates the computation for this call to `fact`—augments the one passed to `fact`. An interesting consequence of converting the definition of `fact` is that the new definition is tail-recursive, even though the original definition was not. This is true of all functions written in CPS [Ste76]. Further, the only computation that is actually done in `fact` is the calculation of the new actuals for the recursive call to `fact`. All other computations are done in the newly constructed continuation after the basis case calls its continuation.

```
(define (fact n k)
  (if (eq? 0 n) (k 1)
      (fact (- n 1) (lambda (v) (k (* v n)))) ))
```

Figure 2: A CPS factorial definition.

Any procedure or entire program can be converted to CPS. For example, typical meta-circular interpreters for Scheme have a very simple format. The evaluation function looks for the syntactic constructs of the language being implemented and dispatches the evaluation of them to helper-functions. These functions, in turn, may call the evaluator recursively. Such an interpreter can be converted to CPS in the same way that our factorial example has been. Figure 3 shows a portion of such an interpreter implementing Scheme presented in [Haynes86]. Note that the author named the main evaluation function `meaning` rather than the usual `eval`.

```
(define meaning
  (lambda (e r k); e = expression, r = environment, k = continuation
    (if (symbol? e)
       (k (deref (r identifier)))
       (match e
         [('quote literal) literal]
         [('if predicate-exp then-exp else-exp)
            (meaning predicate-exp r
              (lambda (v) (if v
                              (meaning then-exp r k)
                              (meaning else-exp r k))))]
         .
         .
         .
       ))))
```

Figure 3: A portion of a CPS-circular interpreter for Scheme.

4 Implementation Description

This section describes the implementation of our interpreter. As described above, this interpreter is a recursive CPS interpreter based on the [HF86] interpreter for Scheme. The description incrementally improves the design of the interpreter until the final version is reached. The complete implementation may be found in appendix B.

4.1 Success and Failure

The simplest expressions in Icon are conditional and monogenic expressions. Examples of such include strings, numbers, and variables. For our interpreter, these expressions represent the basis case for its recursive definition. [HF86]

develops a recursive CPS interpreter that uses a single continuation to represent the computation yet to be completed. This reflects the underlying semantics of the language. Scheme expressions produce a single result, which is always passed to the same context (continuation) no matter what the value is. Control decisions in Scheme are based on values of control expressions.

An interpreter for Icon, however, must provide a different semantics. Expressions may succeed and continue forward in the computation, or fail and cause control to backtrack. Thus, an expression has two possible control paths: one for success and one for failure. We choose to represent these two paths by two different continuations[1]. The choice of which continuation to call is based on outcome (success or failure), not on values as with Scheme.

Upon success of an expression, the result is passed to the *success continuation*. Failure causes the failure continuation to be called—with no parameter, since no value was produced. Below is the form of the interpreter that incorporates these continuations.

```
(define (eval exp env sk fk)
  (cond ((number? exp) (sk exp))
        ((string? exp) (sk exp))
        ((&fail? exp) (fk))
        ((symbol? exp) (sk (lookup exp env)))
        ((if? exp) (eval-if exp env sk fk))
     .
     .
     .
        ))
```

Figure 4: Interpreter incorporating failure.

Strings and numbers evaluate to themselves, while variables require a simple lookup for their values. &fail simply calls its failure continuation directly; note it passes no parameters.

As our first example of how control is modelled in the interpreter, we present the implementation of if. To better understand its implementation, let us first look at the definition of if for Scheme given in [Haynes86]:

```
(define meaning
         .
         .
         .
         [('if predicate-exp then-exp else-exp)
            (meaning predicate-exp r
              (lambda (v) (if v
                              (meaning then-exp r k)
                              (meaning else-exp r k))))]
         .
         .
         .
       ))))
```

Figure 5: Typical implementation of if.

---

[1] For a formal description of Icon in terms of denotational semantics, see [Gud86].

This definition recursively calls `meaning` to evaluate the control clause. The control clause's value is then passed to the newly constructed continuation, where the selection of the expression to be evaluated, the then or else, is made.

In contrast, a control structure in Icon will always provide both success and failure continuations. Figure 6 shows

```
(define (eval-if exp env sk fk)
  (let ((condition (cadr exp))
        (then-exp (caddr exp))
        (else-exp (cadddr exp)) )
    (eval condition env
          (lambda (v) (eval then-exp env sk fk))
          (lambda () (eval else-exp env sk fk)) )))
```

Figure 6: Our implementation of if.

our definition of Icon's if.

Since the evaluation of the control clause may fail, the selection, based on the outcome of the control clause, must be made by the evaluator itself. Therefore, the call to the evaluator must generate two continuations for the evaluator to select after evaluation. As you will see, the use of success and failure continuations is universal throughout all of the control structure evaluation functions.

4.2 Generators

Icon expressions can generate more than a single value. As with all values, the value produced by an expression is passed to a continuation. A generative expression may be requested to produce other results upon failure of a subsequent expression. A generative expression must make available, in some way, a method by which such a request can be made. In our interpreter, this information is encapsulated by the expression in a *resumption continuation*. This continuation may be called to produce the next value of a generator when it is needed. It is passed along with the expression's result to the surrounding context. Therefore, in the interpreter, every expression that produces a value represents the value by a pair: the actual value, and the resumption continuation for the expression that produced the value.

Figure 7 shows the interpreter that incorporates the resumption continuation. For integers, strings, and variables, the resumption continuation is passed as a pair with the value. Note that for numbers, strings, and variables the resumption continuation is simply the failure continuation for this evaluation. This is because these expressions are monogenic and can't produce other results later. In contrast, generators build a resumption continuation that truly will resume the

```
(define (eval exp env sk fk)
  (cond ((number? exp) (sk (cons exp fk)))
        ((string? exp) (sk (cons exp fk)))
        ((&fail? exp) (fk))
        ((symbol? exp) (sk (cons (lookup exp env) fk)))
        ((if? exp) (eval-if exp env sk fk))
     .
     .
     .
        ))
```

Figure 7: Interpreter incorporating resumption continuations.

expression. However, if the generator is exhausted, it must supply the failure continuation to allow further backtracking to take place. The following code implements a generator of two parameters that produces the two parameters in order. The return success continuation (`sk`) is passed the first parameter and a resumption continuation. .The resumption con-

```
(define (ret-args args sk fk)
  (let ((first (car args))
        (second (cadr args))
    (sk (cons first (lambda ()
                      (sk (cons second fk)))))))
```

Figure 8: A simple generator.

tinuation, if called, will pass the second parameter and the return failure continuation (`fk`) for the generator. Thus, if a third result is needed, the generator will fail. Figure 9 shows the implementation of the to generator. Since the to generator produces a variable number of results based on its arguments, the resumption continuation is recursive. The basis case (no more results) calls the generator's return failure continuation.

```
(define (toby-builtin args sk fk)
  (letrec ((lower (car args))
           (upper (cadr args))
           (by (if (null? (cddr args)) 1
                   (caddr args)))
           (resume (lambda ()
                     (let ((tmp-lower lower))
                       (set! lower (+ lower by))
                       (if (< upper tmp-lower) (fk)
                           (sk `(,tmp-lower . ,resume))))))))
    (resume)))
```

Figure 9: Implementation of the to generator.

4.3 Bounding of Expressions

Previously, we saw that the evaluation of an expression generated not only a value, but a continuation that could be called to resume the evaluation. We have also seen that these resumption continuations are passed along and bound to failure continuations for the evaluation of subsequent expressions. However, as described in Section 2.5, the backtracking mechanism is "bounded" by certain control structures. The bounding of expressions is "passive" in the interpreter. That is, when a backtracking boundary is reached, the resumption continuation is simply *not* passed. The implementation of if (see Figure 6) is a good example of this. The resumption continuation passed from the evaluation of the control clause is not bound to either continuation in either recursive call to `eval`. Therefore, the continuation is lost and can never be called to resume the evaluation of the control clause.

As another example of bounding, the subexpressions of a compound expression are also bounded as described in Section 2.5. Figure 10 shows the implementation of compound expressions:
For a given subexpression, we bind the success and failure continuations to the evaluation of the next subexpression in the list. Therefore, upon success *or* failure, evaluation continues forward rather than resuming the previous subexpression.

So far, we have seen that control is represented in the interpreter by both success and failure continuations. To support generators, a third continuation, called the resumption continuation, is used. The result of evaluating an ex-

```
(define (eval-begin exp env sk fk)
  (if (null? (cdr exp)) (fk)
      (letrec ((eval-list-helper
                (lambda (exp)
                  (if (null? (cdr exp)) (eval (car exp) env sk fk)
                      (let* ((fk (lambda ()
                                   (eval-list-helper (cdr exp)))))
                        (eval (car exp) env (lambda (v) (fk)) fk)))))))
        (eval-list-helper (cdr exp)))))
```

Figure 10: Implementation of compound expression.

pression is therefore a pair consisting of a value and a resumption continuation. Furthermore, in the absence of bounding, resumption continuations become the subsequent expression's failure continuation. Bounding is accomplished by substituting a continuation from the outer context for the resumption continuation. With this in mind, we may introduce the procedure call mechanics.

4.4 Procedure Calls

The procedure call is the most complicated detail of the implementation. The call is composed of the evaluation of the actual parameters, binding of the formals to the actuals, and the invocation of the procedure.

4.4.1 Parameter Evaluation

Now that we have some understanding about the way that values are represented and simple control constructs are implemented, we are ready to examine argument evaluation. This is complicated by the introduction of generators as actual parameters. We delay the discussion of generators until Section 4.3. Figure 11 shows the definition of `eval-actuals` which handles the evaluation of arguments. `eval-actuals` calls the evaluator on the first expression

```
(define (eval-actuals exp env sk fk)
  (eval (car exp) env
        (lambda (parm)
          (if (null? (cdr exp))
              (sk (cons (list (car parm)) (cdr parm)))
              (eval-actuals (cdr exp) env
                            (lambda (parm-list)
                              (sk (cons (cons (car parm)
                                              (car parm-list))
                                        (cdr parm-list))))
                            (cdr parm) ))) fk))
```

Figure 11: Evaluation of actual parameters.

(actual parameter) in the list passed to it and bundles a call to itself as the success continuation. Calls to `eval` and `eval-actuals` recursively alternate, allowing `eval` to be called on each actual, and `eval-actuals` to build a

continuation chain[1] that will generate the parameter list upon completion of parameter evaluation. The parameter list has the form:

((*arg*$_1$ *arg*$_2$ ... *arg*$_n$) *k*)

where *k* represents the resumption continuation for the right-most parameter. This technique is also used by [HF86].

The recursive call to `eval-actuals` is made by the success continuation to `eval`. `eval` passes a list (`parm`) containing the value of the evaluated parameter and its resumption continuation. During any particular invocation of `eval-actuals`, the interpreter dispatches the evaluation of a single parameter. Recall that if the evaluation fails, the previous parameter must be resumed for another result. This is accomplished by passing the current resumption continuation from `eval` (`(cdr parm)`) as the failure continuation for the next `eval-actuals` invocation. In fact, whenever an expression is not bounded this binding of the of the last resumption continuation to the current failure continuation occurs.

Upon success, the current success continuation is augmented by code that prepends the value of the current parameter to the result list (creating the continuation chain discussed above). Note that the order of invocation of the success continuations passed to `eval-actuals` is opposite of the order of their corresponding `eval-actuals` invocations. Therefore, the continuation chain is not called until the entire parameter list has been successfully evaluated.

4.4.2 The Procedure Call

The entire procedure calling sequence is controlled by `eval-app`. The application of the function to the arguments is done in the success continuation for the initial invocation of `eval-actuals`. The value that is passed to the continuation is a pair that contains a list and a continuation, as described in Section 4.2. The first element in the parameter list is the function to be applied, and the continuation is used to resume the evaluation of the function's parameters. It is interesting to note that the actual binding of the formals to the actuals and the resumption continuation to the failure continuation of the function is done by the function itself. The definition of each function is prepended by the code necessary to perform these bindings. This method is also used by [HF86].

```
(define (eval-app exp env sk fk)
  (eval-actuals exp env
                (lambda (val-list)
                  (let ((function (deref (caar val-list)))
                        (actuals (cdar val-list))
                        (back-k (cdr val-list)) )
                    (function actuals sk back-k)))
                fk ))
```

Figure 12: Function application.

4.5 Return, Suspend and Fail

During the evaluation of an expression, the interpreter must choose where program execution (evaluation) will continue. These choices are restricted to evaluating a subexpression or choosing to call one of the continuations that the

[1] A continuation chain is a list of continuations that are called in sequence that build a structure. Each continuation in the chain contributes a value to the structure built by the chain. In this case, the structure is a parameter list.

interpreter has been passed. So far, the only available continuations represent the rest of the computation based on whether the current expression fails or produces a value.

During evaluation of a subexpression, however, return, suspend, or fail may be encountered as in

```
procedure proc
    e0
    if e1 then
        return e2
    e3
end
```

The compound expression has dictated that e_3 should be evaluated whether the if expression succeeds or fails. However, return causes control to "break" from the current procedure, and continue back at the caller. Execution may continue in the caller upon one of two different paths: a success path if e_2 succeeds and a failure path if it fails. Since as we have seen the compound expression already binds the two continuations available to the evaluation of the next expression, we require two more continuations to represent the two return control paths. This brings us to the formulation for return (Figure 13) and suspend (Figure 14). We have added two continuations, `r-sk` and `r-fk`, that represent control paths for return success and return failure respectively. fail simply calls r-fk to return with failure.

```
(define (eval-return exp env sk fk r-sk r-fk)
  (if (null? (cdr exp)) (r-sk (cons &null r-fk))
      (let ((new-sk (lambda (v) (r-sk (cons (car v) r-fk)))))
        (eval (cadr exp) env new-sk fk new-sk fk))))
```

Figure 13: Implementation of return.

```
(define (eval-suspend exp env sk fk r-sk r-fk)
  (eval (cadr exp) env r-sk fk r-sk fk))
```

Figure 14: Implementation of suspend.

The addition of these two continuations brings the total to four. This accurately models the control flow for an Icon program: during evaluation of an expression, the expression can succeed, fail, return a value, or return failure. Therefore, every helper-function must handle all four of these properly. Because control only returns to the calling procedure when a return, suspend, fail or the end of a procedure definition is encountered, these are the only times that the return continuations are called. For all other cases, these are simply passed on to the next evaluation.

Control structures that cause control to break out of normal order of evaluation, as return, suspend, and fail do, require the addition of continuations as seen above.

4.6 Generative Control Structures

The implementation of generative control structures establishes the usefulness of this implementation technique. Many of them were implemented quickly with ease. However, the complexity of several of the control structures in Icon is best illustrated by their implementation.

First, let us look at the definition of every:

```
(define (eval-every exp env sk fk r-sk r-fk)
  (eval (cadr exp) env
        (lambda (e-value)
          (let ((resume (cdr e-value)))
            (if (null? (cddr exp)) (resume)
                (eval (caddr exp) env (lambda (v) (resume)) resume
                      r-sk r-fk) )))
        fk r-sk r-fk))
```

Recall that every evaluates its do clause for *each* value produced by an expression until that expression is exhausted. This is done by simply binding the resumption continuation from the evaluation of the every clause to the failure continuation for the evaluation of the do clause. Notice that the success continuation for the resumption of the every clause remains unchanged, while, after each value is produced, a new resumption continuation is generated and bound to the failure continuation for the do expression.

Finally, Figure 15 shows the implementation for limitation. It is rather involved due to the complexity of the control structure itself. Unlike other control structures in Icon, limitation does more than just dispatch evaluation of expressions based on success and failure. It must count the number of results produced by a generator and force failure upon reaching the upper bound.

```
(define (eval-limit exp env sk fk r-sk r-fk)
  (eval (caddr exp) env                  ; evaluate count expression
        (lambda (value)
          (let ((limit (car value)))
            (if (<= (set! limit (- limit 1)) 0)      ; if count <= 0
                (fk)                      ; then, limit reached, fail
                (eval (cadr exp) env                 ; else, evaluate
                      (lambda (value1)
                        (sk (cons (car value1)
                                  (lambda ()

                                    ; check if limit reached

                                    (if (> (set! limit (- limit 1)) 0)
                                        ((cdr value1)); then resume

                                        ; else limit reached, fail

                                        (fk))))))
                      fk r-sk r-fk))))
        fk r-sk r-fk))
```

Figure 15: limit implementation.

First, the second argument is evaluated to determine the maximum number of results that the first argument may produce. If this number is greater than zero, the generator (first argument) is permitted to produce the first result, otherwise, failure is forced. If a result is produced, the control structure decrements the count each time the generator is

resumed. This is accomplished by augmenting the resumption continuation produced by the second eval. If the counter reaches zero, failure is forced.

5 Addition of a New Control Structure

Now that the interpreter framework has been described and several of the control structure implementations have been shown, let us turn to the addition of a new control structure. It is important that the control model be flexible enough to facilitate the addition of control structures not currently available in Icon. Furthermore, it is interesting to design a control structure that is generally not available in languages having goal-directed evaluation. Note that our modified Icon syntax is used in example Icon programs in this section.

In a CPS interpreter, we see that control is managed by the creation and application of continuations. A given control structure uses continuations to implement its specific behavior. We can add a powerful control structure to such an interpreter by allowing the programmer to access these (already built) continuations. Scheme provides a function, called call-with-current-continuation (call/cc), that provides a general control abstraction to the programmer by giving access to internal continuations. [HF86] succinctly describes the operation of call/cc in Scheme:

> "The function [**call/cc**] must be passed a function of one argument. This argument is in turn passed the current continuation, which is the continuation of the **call/cc** application, represented as a functional object of one argument. Informally, this continuation represents the remainder of the computation from the **call/cc** application point. At any future time this continuation may be invoked with any value, with the effect that this value is taken as the value of the **call/cc** application."

An example use of Scheme's call/cc is shown below. `k` is bound to call/cc's continuation and is immediately called with `12` as its parameter. The effect is that the entire expression evaluates to `12` and the addition is never performed.

```
(call/cc (lambda (k) (+ 4 (k 12))))
```

[HF86][Hay87][HF87] describe a variety of control mechanisms, including coroutines and non-blind backtracking, that may be implemented by the programmer using the call/cc mechanism. So, let's integrate Scheme's call/cc with our interpreter. First, we recognize that the call/cc argument must be a function of two parameters rather than one: a success and a failure continuation. The choice of their bindings is obvious: the success and failure continuations for call/cc. However, recall that success continuations expect a pair: the result and the resumption continuation. The resumption continuation could be supplied by either the programmer or the interpreter; we have chosen the latter. What then, should the interpreter supply as the resumption continuation for the expressions on lines `5` and `11` in Figure 16 There are three plausible choices:

1) The failure continuation of the subsequent call to the success continuation.

2) The failure continuation for call/cc.

3) The continuation to resume invocation of call/cc's functional argument.

Figure 16 illustrates these points for choices 1 and 2. Choice three would cause `phred` to be resumed upon failure of the call to `sk` or `keep`.

Choice three is not a valid one since the resumption continuation is not created until `phred` returns or suspends. A call to the success continuation within `phred` (as is the case) would never return, causing `phred` to never return, thus the resumption continuation for call/cc would not be defined. Choice two violates the model—that continuations never return (presumably even upon failure). Choice one is the definition we choose.

```
1     (global keep)
2     (procedure phred (sk fk)
3           (begin
4                 (:= keep sk)
5                 (sk expr)[1]
6           ))
7                 .
8                 .
9                 .
10                (call/cc phred)[2]
11                (keep expr)[1]
```

Figure 16: Form of call/cc use

Figures 17, 18, and 19 show the modifications to the interpreter. Note that unlike most control structures, the implementation of call/cc must include modifications to application (see Figure 18) since part of its mechanism relies on application. In addition a new datatype has been added—the continuation. Although the modifications may appear to be significant for call/cc, `eval-k` and modifications to `eval-app` are only needed for the support of continuations as a datatype.

```
(define (eval-call/cc exp env sk fk r-sk r-fk)
  (eval (cadr exp) env
        (lambda (arg)
          (let ((new-fk (lambda () (sk `(,&null ,fk))))
                (new-sk (lambda (v) (sk `(,v ,fk))))
                (proc (deref (car arg))))
            (proc `((&k ,sk ,fk) (&k ,fk)) new-sk new-fk))) fk r-sk r-fk))
```

Figure 17: Dispatch routine for call/cc.

```
(define (eval-app exp env sk fk r-sk r-fk)
  (eval-actuals exp env
                (lambda (val-list)
                  (if (starts-with? '&k (deref (caar val-list)))
                      (eval-k (deref (caar val-list)) exp env sk fk r-
sk r-fk)
                      .
                      .
                      .
                  ))))
```

Figure 18: Modifications to eval-app for continuations.

```
(define (eval-k k exp env sk fk r-sk r-fk)
  (if (null? (cddr k))
      (if (= (length (cdr exp)) 0)
          ((cadr k)) (error "No arguments expected"))
      (eval (cadr exp) env
            (lambda (arg)
              ((cadr k) `(,(car arg) ,(caddr k))))
            fk r-sk r-fk)))
```

Figure 19: Dispatch routine for continuation calls.

Figure 20 shows the formulation of a repeat loop using the call/cc construct. The loop body and condition are arbitrary expressions. save-k saves the entry point to the loop (line 8) as a procedural object named loop (line 4). A jump to the top of the loop is accomplished by applying loop (line 10).

```
 1    (global loop)

 2    (procedure save-k (sk fk)
 3         (begin
 4              (:= loop fk)
 5         ))

 6    (procedure main ()
 7         (begin
 8              (call/cc save-k)
 9              loop body
10              (if condition (loop))
11         ))
```

Figure 20: A repeat loop formulation using call/cc.

Although we have chosen this definition for the control of call/cc, it may not be the only appropriate choice. Such a decision can only be made after experimentation with other designs. Since implementation of the control is rather trivial once it has been designed (as shown above), this interpreter framework provides precisely the environment for such experimentation.

6 Concluding Remarks

We have described an interpreter that implements a language based on success and failure of expressions, generators, and backtracking. This interpreter provides the framework for implementing a variety of language mechanisms. It also demonstrates the technique of CPS for a success/failure based language.

In addition to the implementation of a number of Icon control structures, a control abstraction not found in Icon (call/cc) was integrated into the language. This illustrated the ease with which new non-trivial language features may be implemented in the interpreter. Thus, during the language design process, less time may be spent implementing the prototype interpreter and more time may be spent experimenting with the design and interaction of new language mechanisms.

Other constructs that may be interesting within this language domain include non-blind backtracking, coroutines, and pattern matching over procedure arguments. The addition of these constructs would be straightforward.

References

[All88] Lloyd Allison. *Continuations Implement Generators and Streams*. Technical Report 88/112, Department of Computer Science, Monash University, August 1988.

[Dyb87] R. Kent Dybvig. *The SCHEME Programming Language*. Prentice-Hall, 1987.

[GG83] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Prentice-Hall, 1983.

[GG86] Ralph E. Griswold and Madge T. Griswold. *The Implementation of the Icon Programming Language*. Princeton University Press, 1986.

[Gri82] Ralph E. Griswold. The evaluation of expressions in Icon. *ACM Transactions on Programming Languages and Systems*, 4(4): 563-584, October 1982.

[Gri90] Ralph E. Griswold. *Version 8 of Icon*, Technical Report 90-1d, University of Arizona, January 1990.

[GW90] Ralph E. Griswold, and Kenneth Walker. *Building Variant Translators for Version 8 of Icon*, Technical Report 90-4a, University of Arizona, January 1990.

[Gud86] David Gudeman, *A Continuation Semantics For Icon Expressions*. Technical Report 86-15, Department of Computer Science, University of Arizona, April 1986.

[Hay87] Christopher T. Haynes. Logic continuations. *The Journal of Logic Programming*, 4(2):157-176, June 1987.

[HF86] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Obtaining coroutines with continuations. *Computer Languages*, 11(3/4):143-153, 1986.

[HF87] Christopher T. Haynes and Daniel P. Friedman. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems*, 9(4):582-598, October 1987.

[KH89] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *Principles of Programming Languages*. pages 281-292, 1989.

[OG87] Janalee O'Bagy and Ralph E. Griswold. A recursive interpreter for the icon programming language. In *SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 138-149, June 1987.

[Ree86] Jonathan A. Rees and William Clinger, eds. The revised^3 report on the algorithmic language scheme. *SIGPLAN Notices 21*, 12, December 1986.

[Ste76] Guy Lewis Steele Jr. *LAMBDA: The Ultimate Declarative*. AI Memo 379, MIT AI Lab, November 1976.

[Ste78] Guy Lewis Steele Jr. *RABBIT: a compiler for Scheme*. AI Memo 452, MIT, January 1978.

Appendix A

The following grammar describes the language implemented by the interpreter.

program	::= decl-list	
decl-list	::= decl | decl decl-list	
decl	::= **(global** symbol expr **)**	*Global identifier declaration*
decl	::= **(procedure** symbol **(** symbol-list **)** expr **)**	*Procedure declaration*
expr-list	::= expr | expr expr-list	
symbol-list	::= symbol | symbol-list	
expr	::= number | symbol | string	*Basic datatypes*
expr	::= **&fail**	
expr	::= **(** expr expr-list **)**	*Invocation*
expr	::= special-form	
special-form	::= **(fail)**	*Procedure failure*
special-form	::= **(while** expr expr **)**	*While loop*
special-form	::= **(every** expr expr **)**	*Every loop*
special-form	::= **(:=** symbol expr **)**	*Identifier assignment*
special-form	::= **(repalt** expr **)**	*Repeated alternation*
special-form	::= **(return** expr **)**	*Procedure return*
special-form	::= **(suspend** expr **)**	*Generator suspension*
special-form	::= **(lambdafy** (symbol-list) expr **)**	*Anonymous procedure*
special-form	::= **(limit** expr expr **)**	*Limit generator*
special-form	::= **(not** expr **)**	*Not expression*
special-form	::= **(if** expr expr expr **)**	*Conditional*
special-form	::= **(mut** expr-list **)**	*Mutual expression evaluation*
special-form	::= **(?** expr expr **)**	*String scanning expression*
special-form	::= **(alt** expr expr **)**	*Alternation expression*
special-form	::= **(begin** expr-list **)**	*Compound expression*

number, string, symbol correspond to the definitions used for the Scheme language.

Appendix B

This appendix continas a complete listing of the FLECS interpreter. The dialect of Scheme is MIT version 7.0 beta release.

```
(display "Loading evaluator...")
(load "interp.scm")
(display "done") (newline)

(display "Loading variable functions...")
(load "var.scm")
(display "done") (newline)

(display "Loading utilities...")
(load "util.scm")
(display "done") (newline)

(display "Loading type functions...")
(load "types.scm")
(display "done") (newline)

(display "Loading control constructs...")
(load "ctrl-structs.scm")
(display "done") (newline)

(display "Loading builtin functions...")
(load "built-ins.scm")
(display "done") (newline)

(display "FLECS Loaded")
```


File: interp.scm

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  INTERP.SCM  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


; This is an interpreter for the Icon language. It is implemented in Scheme
; using a continuation-passing style.

; Universally throughout this implementation, these abbreviations have the
; following meaning:
;
;       sk   - success continuation (function of 1 parameter)
;       fk   - failure continuation (function of no parameters)
;       r-sk - return-success continuation (function of 1 parameter)
;       r-fk - return-failure continuation (function of no parameters)
;       exp  - expression being evaluated
;       env  - environment to evaluate expression in

; eval is the actual interpreter. It dispatches functions based on the syntax
; of the current expression. This is every bit the same as any other
; Scheme-based interpreter.

(define current-eval-exp '())

(define (eval exp env sk fk r-sk r-fk)
  (set! current-eval-exp exp)
  (cond ((number? exp) (sk (cons exp fk)))      ; succeed with value
        ((string? exp) (sk (cons exp fk)))      ; success with value
        ((&fail? exp) (fk))                     ; expression fails

        ; is a global or local variable or a function/procedure name

        ((symbol? exp) (sk (cons (lookup exp env) fk)))

        ; if it isn't a list, we've run out of possibilities...

        ((not (pair? exp)) (icon-error "invalid Icon expression" exp))

        ; fail keyword. procedure fails

        ((fail? exp) (r-fk))

        ((list? exp) (eval-list exp env sk fk r-sk r-fk))

        ((cset? exp) (sk (cons (string->cset (cadr exp)) fk)))

        ; must be a function/procedure invocation

        (else (let ((e-func (find-cs exp)))
                (if e-func (e-func exp env sk fk r-sk r-fk)
                    (eval-app exp env sk fk r-sk r-fk))))
        ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;                       DETAILS OF INTERPRETER STARTUP
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


; must define it so we can set! it later...

(define global-env '())

; &null needs a value, this is a useful value for the user.

(define &null "Icon &null")

; define the "empty" success and failure continuations. They simply return
; "<< no value >>". The only difference between the two is that failure
; continuations have no parameters.

(define (snull-k v) "<< no value >>")
(define (fnull-k) "<< no value >>")

; define the "error" success and failure continuations. They are used when
; a function requires a continuation, but the continuation should never
; be called.
;
; A good example of this is when the main procedure is initially interpreted.
; The 'eval' function requires return continuations, but return is invalid
; for the first invocation of main.
```

```
(define (serror-k v)
  (error "Success continuation that shouldn't have been called" (car v)))
(define (ferror-k)
  (error "Failure continuation that shouldn't have been called"))

; define the continuation that will be called last. That is, this continuation
; is passed to 'eval' as the failure continuation for the first invocation
; of main.

(define (term-k) (newline) (display "Icon program terminated.") (newline))

; icon is the function that is actually called to invoke the interpreter.
; It initializes the global environment, loads the Icon functions (that
; are defined IN Scheme Icon), and starts 'eval' on procedure main.

(define (icon filename)
  (newline)
  (set! global-env (make-global-env))
  (each-object "built-ins.sicn" (lambda (object) (load-icon object global-env
                                                            #f)))
  (each-object filename (lambda (object) (load-icon object global-env #t)))
  (newline)
  (display "Execution starting...")
  (newline)
  (eval '(main) global-env (lambda (v) (display "Value: ") (display (car v))
(term-k)) term-k serror-k ferror-k))
```

File: ctrl-structs.scm

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  CTRL-STRUCTS.SCM  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


; This section contains the functions that implement the Scheme Icon control
; structures. These are all automajically dispatched by the eval function.
; Thus, each is required to accept the same arguments.

; not:  Swaps the success and failure continuations. Note that to do this,
; the old success continuation no longer can take an argument, and the
; old failure continuation must now take an argument. For the old success,
; we simply throw away the argument, for the old failure, we pass &null
; as the argument.
;
; Syntax: (not exp)

(define (eval-not exp env sk fk r-sk r-fk)
  (eval (cadr exp) env (lambda (value) (fk))
        (lambda () (sk (cons &null fk))) r-sk r-fk))

; '\':  Limitation is the most complex of the control structures. First,
; exp2 is evaluated, if it is greater than 0, exp1 is evaluated, and returned.
; If the outer environment attempts to resume the limitation, the value of
; exp2 is decremented. Again if exp2 is greater than 0, exp1 is resumed.
; This process of decrement, compare, resume is continued until no more values
; are requested, exp2 becomes 0, or exp1 is exhausted of values.
;

; The use of 'limit' is to disambiguate from the `\` operator (return value
; if not null), since control structures look like functions, procedures and
; operations.
;
; Syntax: (limit exp1 exp2)

(define (eval-limit exp env sk fk r-sk r-fk)
  (eval (caddr exp) env
        (lambda (value)
          (let ((limit (type-convert (deref (car value)) 'number
                                     (lambda () (icon-error
                                                  "Expected integer for lim-
it"
                                                  (deref (car value)))))))
            (if (<= (set! limit (- limit 1)) 0) (fk)
                (eval (cadr exp) env
                      (lambda (value1)
                        (sk (cons (car value1)
                                  (lambda ()
                                    (if (> (set! limit (- limit 1)) 0)
                                        ((cdr value1))
                                        (fk))))))
                      fk r-sk r-fk))))
        fk r-sk r-fk))

; lambdafy: This is a new control structure introduced by Scheme Icon. It
; simply builds a new procedure from the supplied formals and body. Its value
; can then be invoked, or assigned to a variable by the enclosing body.
;
; Syntax: (lambdafy formals e1 ... en)

(define (eval-procedure exp env sk fk r-sk r-fk)
  (sk (cons (build-procedure (cdr exp) env) fk)))

; suspend: evaluates the expression, and returns the value to the current
; procedure's caller. This is done by passing eval the current procedure's
; return-success continuation as the new success continuation. That way, if
; the expression produces a value, it is passed to the caller. Note that
; suspend fails if the expression fails.
;
; Syntax: (suspend expr)

(define (eval-suspend exp env sk fk r-sk r-fk)
  (eval (cadr exp) env r-sk fk r-sk fk))

; return: works like suspend, except that the procedure is restricted to
; returning a single result.
;
; Syntax: (return expr)

(define (eval-return exp env sk fk r-sk r-fk)
  (if (null? (cdr exp)) (r-sk (cons &null r-fk))
      (let ((new-sk (lambda (v) (r-sk (cons (car v) r-fk)))))
        (eval (cadr exp) env new-sk fk new-sk fk))))

; '|' (repeated alternation): Produce results by evaluating and resuming
; exp. If exp fails after the first evaluation, it is evaluated again. If
; exp fails on the first evaluation, repeated alternation fails.
;
; Syntax: (repalt exp)

(define (eval-repalt exp env sk fk r-sk r-fk)
  (eval (cadr exp) env (lambda (v) (eval-repalt exp env sk fk r-sk r-fk))
```

```
        fk r-sk r-fk))

; ':=' (variable assignment): evaluate exp and bind it to var-exp. The
; 'variable' is returned. This allows the outer environment to "get at"
; the name of the variable if it wants. Thus, to find the value of the
; variable, it must be dereferenced.
;
; Syntax: (:= var-exp exp)

(define (eval-assignment exp env sk fk r-sk r-fk)
  (eval (caddr exp) env
        (lambda (r-value)
          (eval (cadr exp) env (lambda (l-value)
                                 (if (starts-with? '&var (car l-value))
                                     (begin
                                       (bind-var (car l-value)
                                                 (deref (car r-value)))
                                      (sk (cons (car l-value) (cdr r-value))))
                                     (icon-error "Variable expected in assign-
ment"
                                                 (car l-value))))
                fk r-sk r-fk))
        fk r-sk r-fk))

; while: The expression list is executed repeatedly until cond fails.
;
; Syntax: (while cond exp)

(define (eval-while exp env sk fk r-sk r-fk)
  (eval (cadr exp) env
        (lambda (condition)
          (let ((eval-again (lambda ()
                              (eval-while exp env sk fk r-sk r-fk))))
            (if (null? (cddr exp)) (eval-again)
                (eval (caddr exp) env (lambda (v) (eval-again))
                      eval-again r-sk r-fk))))
        fk r-sk r-fk))

; every: The expression list is executed after every result generated by exp.
; exp is repeatedly resumed until it fails.
;
; Syntax: (every exp exp)

(define (eval-every exp env sk fk r-sk r-fk)
  (eval (cadr exp) env
        (lambda (e-value)
          (let ((resume (cdr e-value)))
            (if (null? (cddr exp)) (resume)
                (eval (caddr exp) env (lambda (v) (resume)) resume r-sk r-fk)
            )))
        fk r-sk r-fk))

; alternation: The first expression is evaluated. If the first ever fails, the
; second is then evaluated in its place.
;
; Syntax: (alt exp exp)

(define (eval-alt exp env sk fk r-sk r-fk)
  (eval (cadr exp) env sk
        (lambda () (eval (caddr exp) env sk fk r-sk r-fk)) r-sk r-fk))

; string scanning: defines first expression as &subject. Executes
; second expression in that environment.
;
; Syntax: (? exp exp)

(define (eval-string-scan exp env sk fk r-sk r-fk)
  (eval (cadr exp) env
        (lambda (subject-value)
          (bind-var (lookup '&subject global-env)
                    (type-convert (deref (car subject-value)) 'string
                                  (lambda () (icon-error
                                              "String expected for ?"
                                              (deref (car subject-value))))))
          (bind-var (lookup '&pos global-env) 1)
          (eval (caddr exp) env sk fk r-sk r-fk)) fk r-sk r-fk))

; if: cond is evaluated, if it generates a result, then-clause is evaluated,
; otherwise, else-clause is evaluated.
;
; Syntax: (if cond then-clause else-clause)

(define (eval-if exp env sk fk r-sk r-fk)
  (let ((condition (cadr exp))
        (then-exp (caddr exp))
        (else-exp (cadddr exp)))
    (eval condition env
          (lambda (v)
            (eval then-exp env sk fk r-sk r-fk))
          (lambda ()
            (eval else-exp env sk fk r-sk r-fk))
          r-sk r-fk)))

(define (eval-k k exp env sk fk r-sk r-fk)
  (if (null? (cddr k))
      (if (= (length (cdr exp)) 0)
          ((cadr k))
          (icon-error "Failure continuation requires zero parameters" exp))
      (if (= (length (cdr exp)) 1)
          (eval (cadr exp) env
                (lambda (arg)
                  ((cadr k) `(,(car arg) ,(caddr k))))
                fk r-sk r-fk)
          (icon-error "Success continuation requires one parameter" exp))))

; eval-app controls function application. The syntax for function application
; is as follows:
;
;       (func-exp actual1 ... actualn)
;
;
;

(define (eval-app exp env sk fk r-sk r-fk)
  (let ((&subject (lookup '&subject global-env))
        (&pos (lookup '&pos global-env))
        (saved-&subject &null)
        (saved-&pos 1))
    (eval-actuals exp env
                  (lambda (val-list)
                     (if (starts-with? '&k (deref (caar val-list))) (eval-k
(deref (caar val-list)) exp env sk fk r-sk r-fk)

                      (let ((function
                              (type-convert (deref (caar val-list)) 'procedure
                                            (lambda ()
```

```
                                        (icon-error
                                         "Procedure expected for invoca-
tion"
                                         (deref (caar val-list))))))
                          (actuals (cdar val-list))
                          (back-k (cdr val-list)))
                      (set! saved-&subject (deref &subject))
                      (set! saved-&pos (deref &pos))
                      (function actuals
                                (lambda (v)
                                  (bind-var &subject saved-&subject)
                                  (bind-var &pos saved-&pos)
                                  (sk v))
                                (lambda ()
                                  (bind-var &subject saved-&subject)
                                  (bind-var &pos saved-&pos)

                                  (back-k))))))
                  fk r-sk r-fk)))

; eval-begin simply evaluates each expression in a list. The success and
; failure continutation for any but the last expression are the same: continue
; evaluating the expressions. The success and failure continuations for the
; last expressions are determined by the caller.

; Syntax: (begin expr ... expr)

(define (eval-begin exp env sk fk r-sk r-fk)
  (if (null? (cdr exp)) (fk)
      (letrec ((eval-begin-helper
                (lambda (exp)
                  (if (null? (cdr exp)) (eval (car exp) env sk fk r-sk r-fk)
                      (let* ((fk (lambda () (eval-begin-helper (cdr exp)))))
                        (eval (car exp) env (lambda (v) (fk)) fk r-sk r-fk))))))
        (eval-begin-helper (cdr exp)))))

; eval-actuals evaluates each actual parameters, and gathers them in a list
;

(define (eval-actuals exp env sk fk r-sk r-fk)
  (eval (car exp) env
        (lambda (parm)
          (if (null? (cdr exp))
              (sk (cons (list (car parm)) (cdr parm)))
              (eval-actuals (cdr exp) env
                            (lambda (parm-list)
                              (sk (cons (cons (car parm) (car parm-list))
                                        (cdr parm-list))))
                            (cdr parm) r-sk r-fk)))
        fk r-sk r-fk))

; eval-list evaluates a sequence of expressions, passing the entire list
; of evaluated expressions to it's success continuation.

(define (eval-list exp env sk fk r-sk r-fk)
  (eval-actuals (cdr exp) env
                (lambda (list)
                  (sk (cons (cons 'list (deref-list (car list))) fk)))
                fk r-sk r-fk))

; mutual evaluation. Evaluates each, allowing backtracking into any of
; the previous expressions.

(define (eval-mutual exp env sk fk r-sk r-fk)
  (eval-actuals (cdr exp) env
                (lambda (list)
                  (sk (cons (list-tail (car list) (- (length (car list)) 1))
                            (cdr list))))
                fk r-sk r-fk))

; call with current continuation
;
; Syntax: (call/cc procedure)

(define (eval-call/cc exp env sk fk r-sk r-fk)
  (eval (cadr exp) env
        (lambda (arg)
          (let ((new-fk (lambda () (sk `(,&null ,fk))))
                (new-sk (lambda (v) (sk `(,v ,fk))))
                (proc
                 (type-convert (deref (car arg)) 'procedure
                               (lambda ()
                                 (icon-error "Procedure expected for call/cc"
                                             (cadr exp))))))
            (proc `((&k ,sk ,fk) (&k ,fk)) new-sk new-fk))) fk r-sk r-fk))

; ctrl-structs is used to find the name of a function when dispatching on
; a control structure. Also defines the number of expressions that is valid
; for a given control structure.
;

(define ctrl-structs
  `((while . (,eval-while (1 2)))
    (every . (,eval-every (1 2)))
    (:= . (,eval-assignment 2))
    (repalt . (,eval-repalt 2))
    (return . (,eval-return 1))
    (suspend . (,eval-suspend 1))
    (lambdafy . (,eval-procedure 2))
    (limit . (,eval-limit 2))
    (mut . (,eval-mutual any))
    (not . (,eval-not 2))
    (if . (,eval-if (2 3)))
    (alt . (,eval-alt 2))
    (call/cc . (,eval-call/cc 1))
    (? . (,eval-string-scan 2))
    (begin . (,eval-begin any))
    ))

; find-cs is the interface to ctrl-structs. ctrl-structs defines the
; function to evaluate a given control structure, and the number of expressions
; that the function expects.

(define (find-cs exp)
  (let* ((num-args (length (cdr exp)))
         (cs (assq (car exp) ctrl-structs)))
    (if cs (cond ((number? (caddr cs))
                  (if (= (caddr cs) num-args) (cadr cs)
                      (icon-error "illegal expression (wrong number of forms)"
                                  exp)))
                 ((pair? (caddr cs))
                  (if (reduce boolean/or ()
                              (map (lambda (value)
                                     (= num-args value)) (caddr cs)))
                      (cadr cs)
                      (icon-error "illegal expression (wrong number of forms)"
```

```
                              exp)))
                ((equal? (caddr cs) 'any) (cadr cs))
                )
        #f)))
```

File: types.scm

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  TYPES.SCM  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


; Functions for maintaining type harmony.

; converts a string to a cset

; checks that the type and number of arguments match.

(define (prepare-args args types)
  (if (= (length args) (length types))
     (map (lambda (arg new-type) (type-convert (deref arg) new-type)) args types)
      (icon-error "Wrong number of arguments" args)))

(define (string->cset set)
  (let ((len (string-length set)))
    (letrec ((cset-builder
              (lambda (index)
                (if (= index len) '()
                    (let ((cset (cset-builder (+ index 1)))
                          (char (string-ref set index)))
                      (if (memq char cset) cset (cons char cset)))))))
      (cons 'cset (sort (cset-builder 0) char<?)))))

; converts a cset to a string

(define (cset->string set)
  (list->string (cdr set)))

; type conversion routine. Eastablishes valid conversions from any type
; to any type. Far from complete.

(define (type-convert value type-desired . error-func)
  (let* ((id (lambda (v) v))
         (type-map (cond ((char? value) `((char . ,id)
                                          (string . ,(lambda (c) (list->string `(,c))))))
                         ((number? value) `((number . ,id)
                                            (string . ,number->string)))
                         ((string? value) `((string . ,id)
                                            (number . ,string->number)
                                            (cset . ,string->cset)))
                          ((procedure? value) `((procedure . ,id)))
                         ((starts-with? 'list value) `((list . ,id)))
                         ((starts-with? 'cset value) `((cset . ,id)
                                                       (string . ,cset->string)))
                         (else (icon-error "Not a valid type: " value))))
         (converter (assq type-desired (cons `(any . ,id) type-map))))
    (if converter ((cdr converter) value)
        (if error-func ((car error-func))
            (icon-error "Incompatable type conversion" value " to "
                        type-desired)
            ))))
```

File: util.scm

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  UTIL.SCM  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


; build-procedure receives the formal parameters and the body of the
; function. It returns a function that receives the actuals, dereferences
; them, binds their values to the formals, and evaluates body. Note that
; there is a cludge here: the addition of &local_top as variable in the
; environment. This is necessary becuase we must know that the pointer to
; the global environment points to a cell that is not the one pointed to
; by the local environment. Unfortunately, this should be changed. It really
; should be implemented such that if &local_top is ever referenced that this
; entry is skipped over, but that is for another time.
;
; Expects: (formals e1 ... en)

(define (build-procedure exp env)
  (let* ((formals (car exp))
         (body (cadr exp)))
    (lambda (actuals r-sk r-fk)
      (eval body (let ((args (deref-list actuals)))
                   (cons (list '&local_top "Icon &local_top")
                         (bind-parms formals args env)))
                (lambda (v) (r-fk)) r-fk r-sk r-fk))))

(define (load-icon exp env user-defined)
  (cond ((icon-procedure? exp) (load-procedure exp env user-defined))
        ((global? exp) (load-global exp env user-defined))
        (else (icon-error "invalid Icon definition" exp))))

; an icon function takes a list of actuals values, a return success
; continuation, and a failure continuation.

; Syntax: (procedure p-name (arg1...argn) body)

(define (load-procedure exp env ud)
  (if (not (= (length exp) 4)) (icon-error "illegal procedure definition (incor-
rect number of forms)"))
  (let* ((name (cadr exp))
         (eval-body (build-procedure (cddr exp) env)))
    (lookup-bind name eval-body env)
    (cond (ud (display "procedure ") (display name) (newline)))))

; Syntax: (global name value)

(define (load-global exp env ud)
  (if (not (= (length exp) 4)) (icon-error "illegal global definition (incorrect
number of forms)"))
  (eval (caddr exp) env
        (lambda (r-value)
```

```
         (eval (cadr exp) env (lambda (l-value)
                                (bind-var (car l-value) (deref (car r-value)))
                                (cond (ud
                                       (display "global ")
                                       (display (cadar l-value)) (newline))))
               fnull-k snull-k fnull-k))
        fnull-k snull-k fnull-k))

; exhausts the input by reading each object in and executing function on it.

(define (read-object port function)
  (let ((object (read port)))
    (cond ((eof-object? object) port)
          (else (function object) (read-object port function)))))

; opens an input port, exectutes function on each object, and closes the port

(define (each-object filename function)
  (close-input-port (read-object (open-input-file filename) function)))


; checks if a list's first element is exp

(define (starts-with? string exp)
  (if (not (pair? exp)) #f (eq? (car exp) string)))

; standard predidates

(define (fail? exp) (starts-with? 'fail exp))   ; fail keyword?
(define (cset? exp) (starts-with? 'cset exp))   ; cset type
(define (list? exp) (starts-with? 'list exp))   ; list type
(define (&fail? exp) (eq? exp '&fail))          ; is it &fail?
(define (icon-procedure? exp) (starts-with? 'procedure exp))
(define (global? exp) (starts-with? 'global exp))

; error displaying routine for Icon

(define (icon-error error-message . forms)
  (newline)
  (display "Icon error: ")
  (display error-message)
  (if (null? forms) (error "Icon stop"))
  (display " [Found: ")
  (map write (reverse forms)) (display "]") (newline)
  (error "Icon stop"))
```

File: vars.scm

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  VAR.SCM  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



; dereferences a variable. variables have the following form:
;
;       (&var variable-name variable-value)
;
; deref simply returns the value

(define (deref exp)
  (if (starts-with? '&var exp) (caddr exp) exp))

; dereferences a list of variables (formals)

(define (deref-list args)
  (map (lambda (value) (deref value)) args))

; bind-parms does just that. It takes a list of formals and actuals, binds
; their names and values, and prepends them to the environment. This new
; environment is returned as the value of the function.
;

(define (bind-parms formals actuals env)
  (cond ((null? formals) env)
        ((null? actuals) (cons (list (car formals) &null)
                               (bind-parms (cdr formals) '() env)))
        (else (cons (list (car formals) (car actuals))
                    (bind-parms (cdr formals) (cdr actuals) env)))))

; lookup looks up names in the environment. If an associated value is found,
; the non-dereferenced (icky I know) variable is returned, enabling the
; modification of its value at a later time. If a value is not found, the
; variable is bound to &null.
;

(define (lookup name env)
  (let ((binding (assoc name env)))
    (cons '&var
     (if binding binding
         (let ((new-binding (list name &null)))
           (set-cdr! env (cons new-binding (cdr env)))
           new-binding)))))

; bind-var binds a single variable to a value.
;

(define (bind-var var value)
  (set-cdr! (cdr var) (list value)))

; lookup-bind does a lookup of a variable, and then binds it to a new value.

(define (lookup-bind name value env)
  (bind-var (lookup name env) value))
```

File: built-ins.scm

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  BUILT-INS.SCM  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
; Place Icon functions (built-in) in the global environment. This is actually
; a function so that each invocation of the interpreter will have a brand
; spankin' new environment.

(define (make-global-env)
  `((write ,write-builtin)
    (/ ,/-builtin)
    (> ,>-builtin)
    (< ,<-builtin)
    (<= ,<=-builtin)
    (= ,=-builtin)
    (\ ,\-builtin)
    (+ ,+-builtin)
    (== ,==-builtin)
    (! ,!-builtin)
    (toby ,toby-builtin)
    (deref ,deref-builtin)
    (sizeof ,sizeof-builtin)
    (convert ,convert-builtin)
    (substring ,substring-builtin)
    (&version ,"Scheme Icon Version 1.0")
    (&subject ,&null)
    (&pos 1)
    ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;                              BUILT-IN FUNCTIONS
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


; Listed below are all of the Icon functions that are implemented in Scheme.
; This does not include functions that are implemented in Scheme Icon.
; Functions are only written in Scheme when it is not possible, due to the
; nature of Icon, to implement them in Scheme Icon. Write is a good example of
; this, it must be able to receive a variable number of arguments, a feature
; not available in Scheme Icon.

; It should be noted that these functions are a bit awkward. They act exactly
; like user-defined functions. Because of this, they use the same "calling
; convention." This requires that either the return-failure/success
; continuation be called. If it is a success continuation, the value is
; required to be a two element list, whose first value is the return value,
; and whose second value is the continuation to call to resume the function.
; For all of these functions the second value is always the return failure
; continuation that was passed to the function. This is because none of these
; functions are generators.


; allows definition of Icon operators in terms of scheme operators
; Example: + simply uses scheme's +.

(define (use-scheme-operator op . arg-types)
  (lambda (args r-sk r-fk)
    (let ((args (prepare-args args arg-types)))
      (r-sk (cons (apply op args) r-fk)))))

; allows definition of Icon predicates in terms of scheme predicates
; Example: < simply uses scheme's <.

(define (use-scheme-predicate pred . arg-types)
  (lambda (args r-sk r-fk)
    (let ((args (prepare-args args arg-types)))
      (if (apply pred args) (r-sk (cons (cadr args) r-fk)) (r-fk)))))

; some simple operators and predicates.

(define +-builtin (use-scheme-operator + 'number 'number))
(define --builtin (use-scheme-operator - 'number 'number))
(define <-builtin (use-scheme-predicate < 'number 'number))
(define <=-builtin (use-scheme-predicate <= 'number 'number))
(define >-builtin (use-scheme-predicate > 'number 'number))
(define =-builtin (use-scheme-predicate = 'number 'number))

; type conversion function.
;
; Syntax: (convert value "new-type")

(define convert-builtin
  (use-scheme-operator
   (lambda (value type)
     (type-convert value (string->symbol type)))
   'any 'string))

; substring operation
;
; Syntax: (substring lower upper)

(define substring-builtin
  (use-scheme-operator
   (lambda (string start finish)
     (substring string (- start 1) (- finish 1)))
   'string 'number 'number))

; variable dereference operation
;
; Syntax: (deref exp)

(define (deref-builtin args r-sk r-fk)
  (r-sk (cons (deref (car args)) r-fk)))

; write:  write the values of each parameter to the standard output.
; Returns a list containing the values.
;
; Syntax: (write exp1 .. expn)

(define (write-builtin args r-sk r-fk)
  (let ((args (deref-list args)))
    (map display (reverse args)) (newline)
    (r-sk (cons (cons 'list args) r-fk))))

; bang operation -- hardly complete
;
; Syntax: (! exp)

(define (!-builtin args r-sk r-fk)
  (let ((arg (car (prepare-args args '(any)))))
    (cond ((string? arg)
           (letrec ((gen-str
                     (lambda (index)
                       (if (= (string-length arg) index)
                           (r-fk)
                           (r-sk (cons (string-ref arg index)
                                       (lambda () (gen-str (+ index 1))))
```

```
                                )))))
            (gen-str 0)))
          ((or (cset? arg) (list? arg))
           (letrec ((gen-lst
                     (lambda (list)
                       (if (null? list)
                           (r-fk)
                           (r-sk (cons (car list)
                                       (lambda () (gen-lst (cdr list))))
                                 )))))
             (gen-lst (cdr arg))))
          )))

; size operation
;
; Syntax: (sizeof exp)

(define (sizeof-builtin args r-sk r-fk)
  (let ((arg (car (prepare-args args '(any)))))
    (r-sk (cons (cond ((string? arg) (string-length arg))
                      (else (icon-error "Invalid type for sizeof" arg)))
                r-fk))))

; string equivalence
;
; Syntax: (== exp exp)

(define (==-builtin args r-sk r-fk)
  (let ((args (prepare-args args '(string string))))
    (if (equal? (car args) (cadr args)) (r-sk (cons (car args) r-fk)) (r-fk))))

; '\':    if exp is NOT null, return exp.
;
; Syntax: (\ exp)

(define (\-builtin args r-sk r-fk)
  (let ((args (deref-list args))
        (argc (length args)))
    (if (> argc 1) (error "\: too many operands"))
    (if (eq? (car args) &null) (r-fk)
        (r-sk (cons (car args) r-fk)))))

; '/':    if exp is null, return null.
;
; Syntax: (/ exp)

(define (/-builtin args r-sk r-fk)
  (let ((args (deref-list args))
        (argc (length args)))
    (if (> argc 1) (error "/: too many operands"))
    (if (eq? (car args) &null) (r-sk (cons &null r-fk))
        (r-fk))))

; toby:
;
; Syntax: (toby lower upper [by])

(define (toby-builtin args r-sk r-fk)
  (letrec ((lower (type-convert (deref (car args)) 'number))
           (upper (type-convert (deref (cadr args)) 'number))
           (by (if (null? (cddr args)) 1
                   (type-convert (deref (caddr args)) 'number)))
           (resume (lambda ()
                     (let ((tmp-lower lower))
                       (set! lower (+ lower by))
                       (if (< upper tmp-lower) (r-fk)
                           (r-sk `(,tmp-lower . ,resume)))))))
    (resume)))
```


File: built-ins.sicn

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; BUILT-INS.SICN  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



; Standard Icon toby function.

(procedure to (lower upper by)
  (begin
    (:= counter lower)
    (while (<= counter upper)
      (begin
        (suspend counter)
        (:= counter (+ counter (if (\ by) by 1)))
      ))
  ))

; Standard Icon find function. Used for string scanning.

(procedure find (s1 s2 i j)
  (begin
    (:= s1 (convert s1 "string"))
    (if (\ s2) (begin (:= s2 (convert s2 "string")) (:= i (if (\ i) (convert i
"number") 1)))
            (begin (:= s2 &subject) (:= i (if (\ i) (convert i "number") &pos)))
        )
    (:= j (if (\ j) (convert j "number") (+ (sizeof s2) 1)))
    (while (<= (+ i (sizeof s1)) j)
       (if (== (substring s2 i (+ i (sizeof s1))) s1) (begin (suspend i) (:= i
(+ i 1))) (:= i (+ i 1)))
    )
  ))

; Standard Icon move function. Used for string scanning.

(procedure move(i)
  (begin
    (:= oldpos &pos)
    (if (< (sizeof &subject) (+ i &pos))
      (begin
        (:= &pos oldpos)
        (fail)
      )
        (return (substring &subject (deref &pos) (:= &pos (+ &pos i)))))
  ))


; Standard Icon tab function. Used for string scanning.
```

```
(procedure tab(i)
  (begin
    (:= oldpos &pos)
    (if (< (sizeof &subject) i)
      (begin
        (:= &pos oldpos)
        (fail)
      )
        (return (substring &subject (deref &pos) (:= &pos i))))
  ))

; Standard Icon match function. Used for string scanning.

(procedure match (s1 s2 i j)
  (begin
    (:= s1 (convert s1 "string"))
    (if (\ s2)
      (begin
        (:= s2 (convert s2 "string"))
        (:= i (if (\ i) (convert i "number") 1))
      )
      (begin
        (:= s2 &subject)
        (:= i (if (\ i) (convert i "number") &pos))
      ))
    (:= j (if (\ j) (convert j "number") (+ (sizeof s2) 1)))
    (if (== s1 (substring s2 i (+ i (sizeof s1))))
      (suspend (+ i (sizeof s1))) (fail))
  ))

; Standard Icon upto function. Used for string scanning.

(procedure upto (c s i j)
  (begin
    (:= c (convert c "cset"))
    (if (\ s)
      (begin
        (:= s (convert s "string"))
        (:= i (if (\ i) (convert i "number") 1))
      )
      (begin
        (:= s &subject)
        (:= i (if (\ i) (convert i "number") &pos))
      ))
    (:= j (if (\ j) (convert j "number") (+ (sizeof s2) 1)))
    (:= index 0)
    (every (:= chr (! (substring s i j)))
      (begin
        (:= index (+ index 1))
        (every (== chr (! c))
          (suspend index))
      ))
      ))
```