Expression Evaluation in Icon*

Ralph E. Griswold

TR 80-21

August 1980

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

---

*This work was supported by the National Science Foundation under Grant MCS79-03890.

Expression Evaluation in Icon

1. Introduction

Icon [1,2] is a high-level, general-purpose programming language that emphasizes string and structure processing. Icon bears a heritage from SNOBOL4 [3] and SL5 [4] and is partly the result of attempts to improve on these languages and to correct some of their notable defects.

One of the most significant aspects of Icon is the goal-directed evaluation of expressions. This mode of evaluation is a general feature of Icon and is not limited to a specific part of the language, as it is to pattern matching in SNOBOL4 [5].

An important aspect in making goal-directed evaluation a useful tool in Icon is the concept of *generators*, expressions that are capable of generating a sequence of values if that is necessary to achieve successful evaluation of expressions in which they are contained. Unlike CLU [6] and Alphard [7], generators in Icon are a completely general feature of the language and are not limited to specific constructs or the processing of particular data structures. Unlike typical AI languages [8], goal-directed evaluation in Icon is used in general computation, not data base searching.

Although Icon emphasizes string and structure processing and these features provided the original motivation for generators and goal-directed evaluation, the consequences of the expression evaluation mechanism in Icon are more far reaching.

Early use of Icon emphasized string and structure processing, and it was some time before the potentials of its expression evaluation mechanism were appreciated, even by the designers and implementors of the language. If understanding has proved difficult, exploitation has been even more so.

With the expression evaluation mechanism of Icon came unexpected problems in description and understanding. It is tempting to attribute these problems to flaws or inconsistencies in Icon. In fact, it now appears the expression evaluation of Icon is simply more general than that in most programming languages and that this greater generality leads new Icon programmers to assume erroneously that constructions with familiar appearances are limited in their behavior to what they do in other programming languages. For example

```
if e1 then e2 else e3
```

is as traditional a control structure as one could hope to have. Nonetheless, it has possibilities in Icon that are very different from those in other programming languages. Similarly, those familiar with SNOBOL4 assume that Icon expressions such as

```
(e1 | e2) & e3
```

are patterns in disguise and fail to appreciate their potential in more conventional programming contexts.

In fact, newcomers to Icon commonly assume the conjunction operation, `e1 & e2`, has a special role in goal-directed evaluation and imbue it with an undeserved mystique (conjunction is, in fact, the simplest of all operations in Icon). Similarly, the handling of `e1` in

```
if e1 then e2 else e3
```

is viewed as having some special status in goal-directed evaluation (it does not).

One person new to Icon commented that learning most new programming languages is largely a matter of learning new syntax and discovering how to cast familiar operations in the new language; Icon, however, seems to involve more learning of new semantics [9].

This paper describes the evaluation of expressions in Icon and illustrates its generality. No attempt is made to describe other aspects of Icon, such as high-level string processing. The next section uses an informal approach, introducing terminology for new concepts, describing the steps by which expressions are evaluated, using a few illustrative examples. Section 3 uses a more formal approach in which expression evaluation is explicated by a program that provides interpretive semantics.

The remainder of this paper assumes that the reader is familiar with traditional high-level programming languages, such as Algol 60 and has a cursory knowledge of Icon. A familiarity with SNOBOL4 will help in placing some matters in perspective.

The semantics of expression evaluation described here correspond to Version 3 of Icon [2]. In many respects, these semantics are the same as those for Version 2 [10]; differences are noted at the places that they arise.

2. An Informal Description of Expression Evaluation

2.1 Basic Concepts and Terminology

The concepts and terminology common to most programming languages are assumed here. The terms *value* and *variable* are assumed to be primitive and well understood. Values may be represented in a program by *literals* or they may be computed as the results of evaluation of *expressions*. Variables may be represented in program text as *identifiers* or they may be computed as the results of expressions. To simplify description of the evaluation process, both literals and identifiers are considered to be expressions, although actual implementations of Icon may treat them specially.

In Icon, the term *result* is used to describe either a value or a variable. For example, the result of the addition of two numbers is a value, while the result of an assignment expression (in Icon) is a variable.

Icon is an expression-based language; there are no statements as such. Expressions in Icon are divided into two classes: *functions* and *control structures*. Functions may have zero or more *arguments*. As a point of terminology, *operator* is taken to be synonymous with *function*, and *operand* with *argument*, since the differences are syntactic, not semantic. Similarly, expressions in control structures are also referred to as arguments. Functions may be built-in as part of the Icon language, or supplied as procedures in the program. The difference is inessential for the purposes of discussion, and procedures are not described here.

The difference between functions and control structures is based on how their arguments are evaluated. For functions, arguments are always evaluated in the same way, in a strictly left-to-right order and prior to invocation of the function; the evaluation of arguments is independent of the computation performed by the function. For control structures, the evaluation of arguments depends on the specific control structure. For example, in

```
if e1 then e2 else e3
```

e1 is evaluated first and its outcome determines which *arm*, e2 or e3, is evaluated.

In general, expressions produce results. This is true of control structures, such as if-then-else, as well as of ordinary computational expressions, such as addition. In some circumstances, the evaluation of an expression may fail to produce a result. The term *outcome* is used to describe the consequences of evaluating an expression, whether it be a result or the lack of one.

In Icon, the failure of an expression to produce a result is significant and depends,.typically, on the computation that is performed and on the outcome of other expressions contained in it. The absence of a result is not a static property, as it is in some languages in which the result of a statement is meaningless.

Traditionally, the terms *success* and *failure*, called *signals*, have been used in Icon and its predecessors, SNOBOL4 and SL5, to indicate, respectively, an outcome that produces a result or fails to do so. In Icon (unlike SL5), the concept of signal is redundant, since success occurs if and only if a result is produced. The concept of signal is not used here, although the terms *succeeds* and *fails* are used as abbreviations for "produces a result" and "fails to produce a result", respectively. *Success* and *failure* and other grammatical variants are used similarly.

The term *conditional* is used to describe an expression that may fail to produce a result. In Icon, unlike Algol, for example, control structures are "driven" by the success or failure of expressions in their control clauses, rather than by the Boolean values *true* and *false*. In the control structure `if e1 then e2 else e3`

the success or failure of the control clause **e1** determines which of the arms is evaluated For example

`if x > y then z = x else z = y`

appears to be the same in Icon and Algol, but the mechanism by which one of the two arms of the control structure is selected differs in this subtle manner

If a conditional in Icon succeeds, it produces a result (by definition) For arithmetic comparison operators, the result is the value of the right operand This allows, for example, constructions such as

`if x > y > z then z = x else z = y`

It is important to note that the result produced by a conditional may be useful and that it is not restricted to a Boolean value Neither do conditionals produce more than one type (such as numeric and Boolean)

In functions, failure is an "inherited" property In simple cases, failure in the evaluation of an argument to a function causes the function expression itself to fail More precisely, unless all arguments of a function produce results, the function is not invoked and the expression fails For example, the expression

`x = (y > z)`

fails if the conditional fails While it is convenient to say "the assignment fails", the assignment operation is never invoked and (hence) no assignment is made, the value of **x** is unchanged The inheritance of failure in Icon allows a greater generality and richness of expression in control clauses than is available in languages in which only conditional expressions may appear in control clauses An example is

`if x = (y > z) then y = z else y = 0`

Inheritance of failure is a general property of functions, only control structures "intercept" failure Expressions that only contain functions are called *functional expressions*

2.2 Generators

Most functions are not conditional and always produce a single result Examples are arithmetic computations and assignment

The evaluation of expressions in Icon is enriched (and complicated) by *generators*, expressions that may produce more than one value

The logical possibility of producing more than one value is afforded by operations such as the following Consider two strings **s1** and **s2** (e g , "sum" and "(sum*delta-sum)") In general **s1** may occur as a substring of **s2** (or it may not) Furthermore, as in the example above, **s1** may occur more than once as a substring of **s2** (twice in the example above) In Icon, the function **find(s1,s2)** returns the smallest (integer) position at which **s2** occurs as a substring in **s2**, failing if **s1** does not occur as a substring of **s2** For the example above, the value of **find(s1,s2)** is 2, i e at the second character of **s2** Note that, in general, **find** is a conditional as defined earlier The function **find** is also a generator with the capability of producing more than one result (2 and 12 in the example above) The potential results of a generator constitute a sequence The order of this sequence depends on the particular generator and its arguments In the case of **find**, the sequence is in increasing numerical order The sequence produced in the example above is 2, 12

In ordinary computation, **find** produces the smallest result, so that, for the example above

`x = find(s1,s2)`

assigns the value 2 to **x**

If a generator is an argument of a function and the function fails, the generator is "reactivated" to produce another result If there is one, the function is called again Consider

`x = (y < find(s1,s2))`

Suppose **s1** and **s2** are as given earlier and the value of **y** is 5 The first result produced by **find** is 2 and the comparison fails The function **find** generates its second value, 12, the comparison succeeds and the value 12 is assigned to **x**

If, however, the value of y is 20, the comparison fails for the values 2 and 12 At this point find fails to produce a result when reactivated the second time, this failure is inherited by the comparison and assignment operators, and the entire expression fails (The value of x is unchanged, since the assignment operation is never invoked)

As indicated, functions may be activated more than once The term *invoked* is used to describe the initial activation of a function The term *reactivated* is used to describe subsequent activations of a function, which may produce subsequent results

In more complex expressions, in which there may be many generators, the order in which generators produce results becomes an important issue

2.3 The Evaluation of Arguments of Functions

While arguments of functions are evaluated in strictly left-to-right order, the reactivation of generators depends on outcome of intermediate evaluations In fact, any failure that occurs in the evaluation of a functional expression causes *control backtracking*[1] to the most recently evaluated generator in "search of" another result This mode of evaluation is called *goal-directed* in the sense that success of the evaluation of functional expressions is a goal and that the strategy used ensures that all results of all generators are produced, if that is necessary to achieve this goal

Consider again the previous example, assuming y has the value 5

```
x = (y < find(s1,s2))
```

Technically, the operators are syntactic representations of functions and this expression can be cast in function prefix form as

```
=(x <(y find(s1,s2)))
```

The left-to-right order of evaluation for the arguments of = is x and <(y,find(s1,s2)) This leads to the evaluation of the arguments of <, y and find(s1,s2) This in turn leads to the evaluation of the arguments of find, s1 and s2 The function find is then called with the values of s1 and s2 and the value 2 is returned The function < is then invoked with y and 2 Since the value of y is 5, the comparison fails, < fails to produce a result What happens next is crucial to goal-directed evaluation find is reactivated (it is not invoked again with the same arguments, rather its former invocation is reactived at the point it produced its previous result, 2) The function find now returns the value 12, and the function < is invoked (afresh) with the arguments y and 12 This comparison succeeds and produces the result 12, the function = is invoked with the arguments x and 12, and the assignment is made

If, on the other hand, the value of y is 20, the second invocation of < fails, and find is reactivated Since s1 does not occur as a substring of s2 at any other position, find fails Technically, the identifier s2 is reactivated at this point (Although it is obvious that identifiers are not generators, it makes the description of goal-directed evaluation more uniform to assume that the "property of generation" is not known to the evaluation mechanism) The identifiers s2 and s1 both fail to produce another value, control backtracking to the previous identifiers y and x occurs with the same outcome There is no further point to which to backtrack (assuming that this example occurs in isolation), the function = is not invoked, and the entire function expressional fails to produce a result

This informal description of goal-directed evaluation is hardly precise or rigorous Furthermore, it does not illuminate the potential for the use of generators to provide concise representations for complex combinatorial computations A more precise description is undertaken in Section 3 There are some points about functional expressions worth summarizing here, however

(1) Evaluation of arguments is strictly left-to-right

(2) In the absence of failure, there is no control backtracking, no expressions are reactivated, and there are no manifestations of control backtracking

---

[1] Data backtracking the restoration of data to prior states such as "undoing" assignments is not a general feature of Icon and applies only to some specific operations such as reversible assignment and certain string scanning functions

(3) Control backtracking on failure always reactivates the most recently invoked function

2.4 Control Structures

The evaluation of control structures, unlike functional expressions, is idiosyncratic, each control structure has its own mode of evaluation The if-then-else control structure described earlier is familiar and requires no special interpretation for Icon, although there are issues about its outcome that are discussed later Similarly,

```
while e1 do e2
```

evaluates e1 repeatedly as long as e1 produces a result, and it evaluates e2 each time e1 produces a result (The outcome of evaluating e2 does not affect the evaluation of while-do) The outcome of while-do, when e1 fails to produce a result, is the null value

Other control structures are more interesting Alternation, perhaps the most basic generator, has the form

```
e1 | e2
```

Alternation first generates the sequence of results produced by e1 and then the sequence of results produced by e2

Consider first the case in which e1 and e2 are simple functional expressions, such as in

```
(x | y) > 3
```

Conceptually, this expression succeeds (and produces the value 3), if the value of either x or y is greater than 3

Suppose the value of x is 5 and the value of y is 4 Recasting the syntax as was done in the preceding section, this expression becomes

```
>(x | y,3)
```

Note that alternation is not recast in prefix form, since it is a control structure Although alternation looks like an operator, it is not, since it does not obey the evaluation rules for functions

The first argument of >, x | y is evaluated first The result is the variable x, which has the value 5 The second argument is 3, > is invoked with the values 5 and 3, and it returns the value 3

If, however, the value of x is 1, and the value of y is 4, > is invoked with x and 3 and it fails to produce a result At this point the literal 3 (which, like an identifier, can be considered to be a simple expression) is reactivated It has no other value and fails to produce a result Backtracking then reactivates alternation, which produces y The invocation of > with y and 3 produces the result 3 Note that > is invoked twice in this case

Finally, if the value of x is 2 and the value of y is 2, the second invocation of > fails the reactivation of 3 produces no result, the subsequent reactivation of the alternation produces no result, and the entire expression fails Note that > is also invoked twice in this case

One very useful control structure in Icon simply reactivates generators repeatedly to produce all results

```
every e1 do e2
```

every-do effectively "searches" e1 for all results and evaluates e2 for each one As with while-do, the outcome of evaluating e2 does not affect the evaluation of every-do The outcome of every-do itself if the null value

2.5 Expression Sequencing and Barriers to Backtracking

A sequence of expressions is separated by semicolons (either explicitly or implicitly — the Icon translator supplies semicolons where they are appropriate at the ends of program lines) In an expression sequence, expressions are evaluated from left to right For example, in

```
e1, e2, e3
```

The order of evaluation is e1, e2, and e3

Each expression in an expression sequence is isolated with respect to goal-directed evaluation; the semicolons can be considered as barriers to backtracking. Thus regardless of the outcome of evaluating e1, e2 is evaluated next. Furthermore, if evaluation of e2 fails, e1 is not reactivated.

Braces are used to enclose expression sequences and to allow sequences to be used where a single expression is expected. Braces also serve as barriers to goal-directed evaluation; an expression surrounded by braces is not reactivated even if it occurs in a context where another result is needed for a larger, enclosing expression to succeed. For example

```
{x | y} > 3
```

succeeds only if x is greater than 3. The braces serve as a barrier to reactivation of alternation.

While the example above is contrived to illustrate the effect of braces, there are circumstances where such an explicit barrier is useful to prevent undesired control backtracking.

3. Interpretive Semantics of Expression Evaluation

The preceding sections describe expression evaluation in Icon in terms of familiar concepts in other programming languages and rely on examples to provide an intuitive basis for understanding the mechanism. In this section, a more precise description of expression evaluation is given in the form of a program that constitutes "interpretive semantics" for this aspect of Icon.

This program, called expevl, is written in Icon itself. It may appear that an Icon program to explicate expression evaluation in Icon involves a hopeless circularity. Circularity is avoided in two ways — (1) expevl only attempts to describe a limited portion of the semantics of Icon, and (2) expevl itself uses none of the features it undertakes to describe.

The use of Icon to describe Icon constitutes a kind of semantic bootstrap. It builds on an understanding of the conventional features of Icon to provide an understanding of some of its more unconventional features. There are several advantages of using a high-level language in general and Icon in particular for describing semantics. The high-level programming language, while not providing the rigor of formal semantics systems, is nonetheless easier to understand, which is the aim here. The use of Icon, rather than another high-level language, has the advantage of establishing a single vehicle for discourse, avoiding the confusion of switching back and forth between two languages.

3.1 The Scope of expevl

expevl is a program that interprets Icon expressions and is primarily concerned with the order of evaluation of expressions. Only a few, representative kinds of expressions are included; expevl makes no attempt at completeness. Since expression evaluation is the issue, complicating aspects of Icon are omitted; in fact the only type of Icon data handled by expevl is the integer. This avoids issues of automatic type checking and coercion, which have nothing to do with expression evaluation, *per se*.

The following functional expressions are supported by expevl:

```
-e1
e1 + e2
e1 > e2
e1 := e2
e1 <- e2
e1 & e2
e1 to e2
```

Note that e1 to e2 is a functional expression, not a control expression; this matter is discussed later. The by clause is omitted for simplicity only.

expevl supports the following control expressions:

```
e1 | e2
if e1 then e2 else e3
while e1 do e2
e1 fails
e1, e2
{e1}
```

Semicolons and braces are treated separately to illustrate that both serve as barriers to goal-directed evaluation

3.2 The Representation of Icon Expressions in expevl

Icon expressions are represented by records in expevl Functional expressions are divided into unary and binary categories for technical reasons that are described later

```
record unary(func,e1)
record binary(func,e1,e2)
```

Here func is the name of the function (represented by an Icon string, such as "-") and e1 and e2 are records representing the argument expressions

There is a record type for each kind of control expression

```
record alter(e1,e2)
record if_then(e1,e2,e3)
record while_do(e1,e2)
record fails_(e1)
record compound(e1,e2)
record limit(e1)
```

Icon values and variables are also represented by records in expevl

```
record value(constant)
record variable(name)
```

Thus Icon values and variables are encapsulated as data objects in expevl and isolated from other data types used by expevl For example, the Icon expression

```
x := x + 1
```

is represented in expevl by

```
binary(":=",variable("x"),binary("+",variable("x"),value(1)))
```

(expevl contains a "compiler" that translates a more natural form of input into records such as these)

expevl also treats the lack of a result ("failure") in Icon as a separate record type

```
record noresult()
```

and the unique value

```
phi := noresult()
```

Treating the lack of a result in Icon by a specific data object phi in expevl permits expevl to be written without itself using failure of expression evaluation (hence avoiding a potential circularity as discussed above)

One further complication arises because while-do returns the null value, which is not an integer Since type checking and coercion, even between the null value and integers, is a complicated process that would obscure the essential aspects of expression evaluation, a record of type value with the integer value 0 is used to represent the null value in expevl

```
nullvalue := value(0)
```

Since the integer equivalent of the null value is 0, this device assures that expevl produces computationally correct results in the event that an uninitialized identifier or, less likely, a control structure such as while-do is used an arithmetic operation.

3.3 Coding Conventions in expevl

In order for expevl to serve as a valid semantic bootstrap, it must avoid circularity, as mentioned above. That is, it must not use any of the features of Icon that it is designed to explicate. There are two ways that this is accomplished.

(1) By using features of Icon that have direct counterparts in other, traditional programming languages. In this sense, expevl could be written in a variety of other programming languages and hence avoid problems of circularity. An example is given by

```
if e1 then e2 else e3
```

While in Icon this control expression is "driven" by the success or failure of e1, it is used in expevl only in ways that are consistent with usages in languages such as Algol 60, where e1 would be an expression that produces Boolean values *true* and *false*. This constraint amounts to restricting e1 to being a conditional expression (as opposed, say, to an assignment expression that "inherits" failure from one of its operand expressions).

(2) By excluding the use of features essential to expression evaluation. For example, expevl does not use built-in generators, since it is designed to explicate the effect of generators on expression evaluation.

These two kinds of constraints on expevl can be summarized in coding protocols:

(1) Arguments of procedures are variables; there are no expressions in argument lists. Hence no side effects are possible and the order of evaluation of arguments to procedures does not affect the behavior of expevl. More importantly, arguments can neither fail nor generate more than one value.

(2) All expressions in the control clauses of control expressions are simple conditionals. Furthermore, conditionals are only used in the control clauses of control expressions.

(3) No built-in generators are used anywhere in expevl.

(4) All arguments in functional expressions and control expressions (except control clauses) are simple — usually calls of procedures or built-in functions or operators. This protocol is not essential to the "correctness" of expevl, but is intended to make it easy to see that there are no "hidden tricks". In a few cases, expressions are nested one level deep for readability; they can be unnested easily.

(5) Control expressions are used only in contexts where they could be "statements"; that is, no use is made of the results produced by Icon control expressions.

(6) No computation is performed using phi; only its identity is tested.

There are several features of high-level languages that *are* used in expevl; all of these features can be found in other well-known high-level programming languages:

(1) records (as mentioned above), with reference to the values of their fields (but not assignment to fields).

(2) programmer-defined procedures with arguments passed by value.

(3) ordinary expression sequencing using traditional control structures.

(4) case expressions with literal selectors (these could be replaced by if-then-else expressions).

(5) Typical built-in operations and functions, such as arithmetic, assignment, conditionals, and type(x).

(6) repeat loops exited only by means of break (never as a result of expression failure).

(7) The string and object comparison conditionals such as x == y and x === y (the latter can be replaced by the former and the use of type(x)).

There are two "non-standard" constructs used in expevl, which are its "weak links" and deserve more discussion:

(1) Suspension of procedure invocation with the return of a result and the possibility of subsequent reactivation.

(2) Use of every-do to repeatedly activate procedures.

Procedures are used in expevl to model expressions in Icon. In general, arguments of such procedures are records, corresponding to the arguments of the equivalent Icon expressions. Such procedures always return results (a expevl value or variable) using suspend. Such procedures are the only generators in expevl. This is essentially the expevl model for built-in expressions in Icon and constitutes, as well, a model for the way expressions actually might be implemented in Icon. Procedure suspension cannot be claimed as a feature of other well-known, high-level programming languages. However, it is borderline in this respect, since the coroutine is not a *rara avis*, although coroutine mechanisms vary widely. To accept suspend, all that is needed is the acceptance by the reader that procedures may be implemented in a fashion that allows their activity to be suspended (and a result returned) without destruction of the procedure environment and with subsequent reactivation of the procedure at the point following the suspension. In any event expevl does not attempt to explicate either Icon procedures or suspend, so in this sense their use involves no circularity *per se* (although it does introduce a component that cannot be passed off to lower-level languages).

The use of every-do is at once more serious and less serious than the use of suspend. While appeal can be made to the concept of coroutines in the case of suspend, it is harder to find a familiar concept similar to every-do. The problem is made simpler in expevl by the fact that every-do is used in only one paradigm:

```
every x := f(y) do e2
```

where f(y) is the call of a procedure. That is, every-do is only used to repeatedly activate a procedure to obtain its successive values (recall that no built-in generators are used in expevl). Furthermore, this is the only context in which a procedure can be activated more than once in expevl.

The every-do construct of Icon as used in expevl can be modeled by coroutines and loops in other languages. For example, the Icon expression above has the following equivalent in SL5 [11]:

```
z := create f with y
while x := resume z do e2
```

One additional aspect of the use of every-do in conjunction with procedures that suspend is that such procedures never fail (or, for that matter, terminate with return). All every-do loops are terminated by breaks, never as a result of a procedure failing to return a result. Furthermore, the argument of suspend is always a variable (never an expression that might generate a value). (expevl could be made more compact by using alternatives in suspends, but this would violate the coding protocol.)

In summary, there are two ways in which suspend and every-do might cast doubts on expevl: (1) they fail outside of the "bootstrap" concept, since they cannot be taken from well-known features of lower-level languages; and (2) their semantics, possibly being in question, may cast doubts on the correctness of expevl. On the other hand, suspend does not introduce circularity, since expevl does not attempt to explicate it or programmer-defined generators.

4. The expevl Program

expevl consists of two major components, an "interpreter", and a collection of procedures that correspond to functions and control expressions in Icon. There are also a number of support routines. The following sections describe the expevl program. A listing of expevl is given in Appendix A.

4.1 Typical Functions

The general format of procedures that correspond to functions is given by minus, which performs the function -e1:

```
procedure minus(x1)
    local r
    x1 := deref(x1)
    r := value(-x1.constant)
    suspend r
    suspend phi
    error(11)
end
```

x1 is the result of evaluating the expression that is the argument of e1 (which may be complex, as in -(y+2). In general, the value of x1 is either a value or a variable. Dereferencing, done by deref, is done in Version 3 after functions are invoked, as shown here. In Version 2, variables are dereferenced before functions are called. The details of dereferencing are discussed below.

Once the argument is dereferenced, the negative is formed. Note that the argument passed to minus is not changed by dereferencing, since arguments are passed by value in Icon. A new value is produced, so that the Icon value being modeled is not changed.

Note that the use of the Icon minus operator to compute the negative involves no circularity, since expevl is not concerned with the semantics of computation.

The important part of this procedure resides in the last three lines. The suspend r corresponds to returning the value of -e1. Should -e1 be reactivated for another result, control is returned to minus at the next line. suspend phi corresponds to returning no result — "failure". That is, -e1 has only a single result, the negative of e1; it is not a generator.

If expevl is coded correctly, minus (or any other function), should never be reactivated after suspending with phi. For internal error checking purposes, a call to the procedure error is inserted at all places that should never be reached in expevl. Should it be called, it prints a diagnostic message and terminates execution:

```
procedure error(n)
    stop("internal inconsistency at site ",n)
end
```

A typical binary function is illustrated by sum, which performs the operation e1 + e2:

```
procedure sum(x1,x2)
    local r
    x1 := deref(x1)
    x2 := deref(x2)
    r := value(x1.constant + x2.constant)
    suspend r
    suspend phi
    error(14)
end
```

4.2 Variables and Assignment

The values associated with variables in expevl are maintained using a table:

```
sym := table() ::= nullvalue
```

The initial value nullvalue corresponds to the initial null value of variables in Icon.

The assignment operation, e1 := e2, implemented by assign, illustrates how values are inserted in this table:

```
procedure assign(x1,x2)
    if type(x1) ~== "variable" then runtime(121)
    sym[x1 name] = deref(x2)
    suspend x1
    suspend phi
    error(15)
end
```

Again, e1 and e2, which may be expressions, are reduced to variables or values by the argument evaluation mechanism before assign is called The first line of assign is a check to assure the first argument is a variable The procedure runtime handles error termination in Icon

```
procedure runtime(n)
    stop("runtime error ",n)
end
```

assign first inserts the value of the second argument, obtained by dereferencing x2 into sym according to the name of the variable It then returns its first argument (as a variable) If reactivated, it returns no result, using the same model as minus

Reversible assignment illustrates how data backtracking is done

```
procedure revasg(x1,x2)
    local temp
    if type(x1) ~== "variable" then runtime(121)
    temp - sym[x1 name]
    sym[x1 name] - deref(x2)
    suspend x1
    sym[x1 name] = temp
    suspend phi
    error(16)
end
```

Dereferencing involves obtaining the value of a variable from sym If the argument of deref is a variable, its value is looked up in sym If the argument of deref is a value, it is returned unchanged

```
procedure deref(x)
    case type(x) of {
        "variable"   return sym[x name]
        "value"  return x
        default  error(18)
        }
end
```

4.3 A Typical Conditional

Conditionals follow the same model as ordinary computational functions, the only difference being that they may fail to produce a result An example is greater, which corresponds to e1 > e2

```
procedure greater(x1,x2)
    x1 = deref(x1)
    x2 = deref(x2)
    if x1 constant > x2 constant then suspend x2
    suspend phi
    error(13)
end
```

Note that greater returns the value of its second argument if the comparison succeeds

4.4 A Typical Generator

Generators also follow the models given earlier, except more than one result may be produced. An example is given by to_, which corresponds to e1 to e2. The by clause is omitted here to avoid coding details concerning negative indexing and so forth that have nothing to do with the evaluation of expressions.

```
procedure to_(x1,x2)
    local r
    x1 := deref(x1).constant
    x2 := deref(x2).constant
    while x1 <= x2 do {
        r := value(x1)
        suspend r
        x1 := x1 + 1
        }
    suspend phi
    error(17)
end
```

Note that e1 to e2 is a function, not a control structure. Its arguments are evaluated before it is called, just like any other function. This is not true of control structures.

4.5 The Interpreter

The term *interpreter* is used here for the portion of expevl that evaluates arguments, implements control structures, and calls procedures that correspond to functions (such as minus). The procedure that implements these operations is lengthy and begins as follows:

```
procedure interp(node)
    local x1, x2, r
    case type(node) of {
        "value": {
            suspend node
            suspend phi
            error(1)
            }
        "variable": {
            suspend node
            suspend phi
            error(2)
            }
            .
            .
            .
```

The argument, node, may be any of the record types that correspond to expressions in Icon. The case expression selects processing according to type. The two simplest cases are value and variable, which simply return node and then indicate no result if reactivated. Although their code sections are the same, they are not combined to avoid the introduction of alternation in a case selector, which would be a violation of the coding protocol for expevl. In this and many other respects, expevl can be made more compact by allowing use of more features of Icon, once the semantic bootstrap has been effected.

4.6 A Traditional Control Structure

if-then-else is as familiar a control structure as there is and, in one form or another, populates hundreds of programming languages. In Icon, however, even this simple control structure raises a number of issues. The code for if-then-else is one portion of the case clause of interp:

```
"if_then" {
    r = interp(node e1)
    if r === phi
        then {
            r = interp(node e3)
            if r ~=== phi then suspend r
            }
        else {
            r = interp(node e2)
            if r ~=== phi then suspend r
            }
    suspend phi
    error(6)
    }
```

Note that the first argument is invoked by a simple call Thus, even if the first argument is a generator, only its first result (if there is one) is used There is no way that the first argument can be "backed into" (in case, for example, the selected arm fails to produce a result)

The selection of the arm to be evaluated depends on whether or not the first argument produces a result or not The treatment of the two arms is identical Again, the selected argument is invoked by a simple call, so that it may produce at most one result, which becomes the result of if-then (illustrating that it is, indeed an expression)

The fact that the selected arm of if-then-else can only return a single result, even if it is a generator, raises a number of interesting questions The semantics, as given, are the actual semantics of Versions 2 and 3 of Icon Since there is a (natural) tendency to use if-then-else as if it were a statement, the fact that its arms do not act as generators is not ordinarily noticed However, there is no essential reason why its arms could not act as generators Consider for example, an expression such as

```
every i = (if x > y then 1 to x else 1 to y) do f(i)
```

In Versions 2 and 3, only f(1) is called, regardless of whether or not x is greater than y, there are no subsequent calls, since the arms are not reactivated

This issue initially produced considerable controversy in the Icon design group The eventual result of discussion was to change the behavior of if-then-else (and other control structures) in future versions to allow their arguments to be generators, except in control clauses (such as the first argument of if-then-else) The arms of if-then-else can be allowed to be generators as follows

```
"if_then" {
    r = interp(node e1)
    if r === phi
        then {
            every r = interp(node e3) do
                if r === phi then break
                else suspend r
            }
        else {
            every r = interp(node e2) do
                if r === phi then break
                else suspend r
            }
    suspend phi
    return error(6)
    }
```

Determining why control clauses should not be allowed to act as generators provides a good test of the understanding of expression evaluation in Icon There are other situations in which expressions are not used as generators. One occurs in e1 fails:

```
"fails_": {
   r := interp(node.e1)
   if r === phi then suspend nullvalue
   suspend phi
   error(19)
   }
```

If interp(node.e1) were repeatedly activated, the result would eventually be phi and fails would always succeed.

4.7 Alternation

Alternation provides an interesting example of a control structure that is also a generator:

```
"alter": {
   every r := interp(node.e1) do
      if r === phi then break
      else suspend r
   every r := interp(node.e2) do
      if r === phi then break
      else suspend r
   suspend phi
   error(5)
   }
```

As indicated, alternation first calls interp recursively with its first argument (which represents an Icon expression) and returns each result that is returned, as produced by every. For example, if the first argument corresponds to e1 to e2, to_ is repeatedly activated and its results returned.

Activation of the first argument is terminated when phi is encountered, never by failure of interp to produce a result. expevl is written so that all procedures that correspond to Icon functions and control structures produce phi and then terminate program execution via error, should a coding mistake in expevl cause the phi to go undetected.

Once all the results from the first argument are produced, the second argument is treated in the same way. Note that alternation is not a function — its arguments are not evaluated prior to its invocation (and cannot be). Conversely, no function repeatedly activates its arguments; the arguments of a function are evaluated prior to the invocation of the function.

4.8 Repeated Activation of Generators

By way of interest, every-do can also be included in expevl. This is clearly a circularity, but it may add a little to the understanding of repeated evaluation of generators. In any event, it is a separable issue.

```
"every_do": {
   every r := interp(node.e1) do
      if r === phi then break
      else interp(node.e2)
   suspend nullvalue
   suspend phi
   error(8)
   }
```

Note that the outcome of interp(node.e2) is irrelevant to the operation of every-do.

4.9 Evaluation of the Arguments of Functions

One of the functions of interp is to evaluate the arguments of functions. Whereas this is a comparatively simple operation in most programming languages, it involves subtleties in Icon, as is illustrated by the section of interp that evaluates the arguments of unary functions.

```
"unary": {
    every x1 := interp(node.e1) do
        if x1 === phi then break
        else
            every r := dounary(node.func,x1) do
                if r === phi then break
                else suspend r
    suspend phi
    error(3)
    }
```

In the first place, an argument expression may be arbitrarily complex. Note that interp is called recursively and activated repeatedly to produce all possible results for each result of the argument evaluation, the specific unary function is called, using a common procedure, dounary. All unary functions are contained in dounary; the example minus above actually is not a separate procedure in expevl but is selected in a case clause from the value of node.func:

```
procedure dounary(func,x1)
    local r
    case func of {
        "-": {
            x1 := deref(x1)
            r := value(-x1.constant)
            suspend r
            suspend phi
            error(11)
            }
            .
            .
            .
```

Since the unary function may itself be a generator, dounary is also repeatedly activated until phi is produced. Each result produced (prior to phi) is returned by interp. Once dounary produces phi, the inner every loop is broken and the next result from the *argument* expression is produced, until it, too, produces phi.

This is the heart of goal-directed evaluation. It assures, for example, that

```
(x | y) > 1
```

compares y to 1 if x is not greater than 1. It also assures that in

```
f(x | y)
```

f(y) is called if f(x) fails. While these two cases are equivalent, except for syntax, the behavior of the latter expression is often overlooked, since it does not have the intuitive content that the former one does. It does illustrate one of the ways that generators can be used to provide concise representations of complex combinatorial computations.

The nested every loops in the evaluation of arguments of unary functions explicate the order in which expressions are evaluated, and are worth study. The extension the evaluation of arguments of binary functions is natural and straightforward:

```
"binary": {
   every x1 := interp(node.e1) do
      if x1 === phi then break
      else
         every x2 := interp(node.e2) do
            if x2 === phi then break
            else
               every r := dobinary(node.oper,x1,x2) do
                  if r === phi then break
                  else suspend r
   suspend phi
   error(4)
   }
```

Like the case for unary functions, all binary functions are encapsulated in the procedure dobinary. The binary functions given above are not actually coded as separate procedures, but rather within a case expression. See Appendix A for the actual form.

The code segments to evaluate arguments of unary and binary functions are instances of a more general model for evaluating an arbitrary number of arguments (note the "last-in-first-out" order embodied in the nesting of evaluation of the second argument of a binary function under the first). Unfortunately Icon lacks the facility for casting argument evaluation as a single code segment that is parameterized by the number of arguments to be evaluated. A language, such as SL5, that allows procedure environments to be treated as data objects and activated at will allows coding of such an argument-evaluation paradigm. See Appendix B for the actual SL5 code. Unfortunately, the coding in SL5 is somewhat contorted due to its lack of certain control structures. A thorough examination of equivalent Icon and SL5 code for argument evaluation is an illuminating exercise in the comparison of programming languages.

4.10 Conjunction

As mentioned in Section 1, conjunction is often viewed as mysterious or as having some special role in goal-directed evaluation. In fact, it is simply a binary function that returns its second argument:

```
procedure conj(x1,x2)
   suspend x2
   suspend phi
   error(13)
end
```

This procedure clearly illustrates that goal-directed evaluation is a property of the expression evaluation mechanism, not a property of functions themselves. Note that the result that is returned is not dereferenced.

4.11 Expression Sequencing and Barriers

As described in Section 2, the semicolons separating expressions serve as barriers to backtracking. This is illustrated by the handling of e1; e2:

```
"compound": {
   interp(node.e1)
   r := interp(node.e2)
   if r ~=== phi then suspend r
   suspend phi
   error(9)
   }
```

Note that interp is simply *called* to evaluate node.e1 and node.e2; neither can be reactivated for additional results. Thus if the evaluation of e2 fails, e1 is not reactivated; the semicolon serves as a barrier to goal-directed evaluation. Sequences consisting of more than two expressions can be composed by nesting. In Icon itself, of course, expression sequencing is handled in a more general manner.

The braces that enclose expression sequences also serve as barriers, as is illustrated by

```
"limit" {
   r = interp(node e1)
   if r ~=== phi then suspend r
   suspend phi
   error(10)
   }
```

The use of braces to enclose expression sequences and also to serve as barriers to goal-directed evaluation has also been the subject of some discussion within the Icon design group In future versions of Icon, braces will only serve to enclose expression sequences and there will be a separate control structure to limit goal-directed evaluation

5. Conclusions

There is a long-standing joke that the real formal semantics of a programming language consist of its implementation In that sense, the real "authority" on expression evaluation in Icon is the code that actually implements it However, such "formal semantics" inevitably contain a very large amount of detail and extraneous material that is irrelevant to any particular issue While the value of the study of an implementation to understanding a language should not be disparaged, it is nonetheless unsuitable for most purposes and most persons Hence **expevl** is, in some sense, a toy implementation of a deliberately limited portion of Icon

The implementation of interpretive semantics by the program **expevl** has proved to have a number of advantages beyond its use in explicating the expression evaluation mechanism of Icon **expevl** can actually be run This provides a tangibility and authenticity that can never be provided by verbal descriptions or formal semantic systems

One aspect of being able to run **expevl** is that it can be tested on a variety of expressions and the results compared with those of executing these expressions in Icon This aids the debugging of the semantics via debugging of **expevl** Of course, Icon itself may display bugs, although none were found during the development of **expevl** Some misunderstandings of expression evaluation on the author's part were discovered, however

Furthermore, **expevl** can be modified easily — certainly more easily than the actual implementation of Icon The ease of modification allows experiments that otherwise would be impractical In fact, **expevl** has been modified to determine the consequence of possible changes to Icon Examples are the removal of constraints on the arms of **if-then-else**, moving dereferencing out of functions and into the argument evaluation mechanism, and the possibility that the operation of dereferencing itself might fail in certain situations

It is also easy to implement and test different control structures For example, "normal" alternation in SUMMER [12] is simply

```
"or" {
   r = interp(node e1)
   if r === phi then r = interp(node e2)
   if r ~=== phi then suspend r
   suspend phi
   error(19)
   }
```

Thus, **expevl** can also serve as a design tool

Acknowledgement

I am indebted to Tim Budd, Cary Coutant, Dave Hanson, Tim Korb, and Steve Wampler for a number of illuminating discussions about the evaluation of expressions in Icon as well as for a number of specific suggestions related to the content of this paper.

References

1. Griswold, Ralph E., David R. Hanson, and John T. Korb. "The Icon Programming Language; An Overview". *SIGPLAN Notices*, Vol. 14, No. 4 (April 1979). pp. 18-31.

2. Coutant, Cary A, Ralph E. Griswold, and Stephen B. Wampler. *Reference Manual for the Icon Programming Language; Version 3.* Technical Report TR 80-2, Department of Computer Science, The University of Arizona. May 1980.

3. Griswold, Ralph E., James F. Poage, and Ivan P. Polonsky. *The SNOBOL4 Programming Language.* Second Edition. Prentice-Hall, Inc. Englewood Cliffs, New Jersey. 1971.

4. Griswold, Ralph E. and David R. Hanson. "An Overview of SL5", *SIGPLAN Notices*, Vol. 12, No. 4 (April 1977). pp. 40-50.

5. Griswold, Ralph E. and David R. Hanson. "An Alternative to the Use of Patterns in String Processing", *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 2 (April 1980), pp. 153-172.

6. Liskov, Barbara, Alan Snyder, Russell Atkinson, and Craig Schaffert. "Abstraction Mechanisms in CLU", *Communications of the ACM*, Vol. 20, No. 8 (August 1977). pp. 564-576.

7. Shaw, Mary, William A. Wulf, and Ralph L. London. "Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators", *Communications of the ACM*, Vol. 20, No. 8 (August 1977). pp. 553-564.

8. Bobrow, Daniel G. and Bertram Raphael. "New Programming Languages for Artificial Intelligence", *Computing Surveys*, Vol. 6, No. 3 (September 1974). pp. 153-174.

9. Moody, J. K. M. Private communication.

10. Griswold, Ralph E. and David R. Hanson. *Reference Manual for the Icon Programming Language; Version 2.* Technical Report TR 79-1a, Department of Computer Science, The University of Arizona. January 1980.

11. Hanson, David R. and Ralph E. Griswold. "The SL5 Procedure Mechanism", *Communications of the ACM*, Vol. 21, No. 5 (May 1978). pp. 392-400.

12. Klint, Paul. "An Overview of the SUMMER Programming Language", *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages.* January 1980. pp. 47-55.

Appendix A — The Program expevl

A listing of expevl follows. The only parts that have been omitted are the procedures compile, test, and the procedures they in turn use. compile accepts Icon expressions in a reasonably natural format and converts them into the required data structures. test is a simple procedure that calls interp and prints out the results. The complete program is available from the author on request.

```
global nullvalue, phi, sym

record noresult()
record value(constant)
record variable(name)
record unary(func,e1)
record binary(func,e1,e2)
record alter(e1,e2)
record if_then(e1,e2,e3)
record while_do(e1,e2)
record every_do(e1,e2)
record fails_(e1)
record compound(e1,e2)
record limit(e1)

#  The procedure interp(node) is the heart of expevl  It contains the code
#  for the control structures and for the evaluation of the arguments of
#  unary and binary functions

procedure interp(node)
   local x1, x2, r
   case type(node) of {
      "value"  {
         suspend node
         suspend phi
         error(1)
         }
      "variable"  {
         suspend node
         suspend phi
         error(2)
         }
      "unary"  {
         every x1  = interp(node e1) do
            if x1 === phi then break
            else
               every r  = dounary(node func,x1) do
                  if r === phi then break
                  else suspend r
         suspend phi
         error(3)
         }
      "binary"  {
         every x1  = interp(node e1) do
            if x1 === phi then break
            else
               every x2  = interp(node e2) do
                  if x2 === phi then break
                  else
                     every r  = dobinary(node func,x1,x2) do
```

```
                            if r === phi then break
                            else suspend r
        suspend phi
        error(4)
        }
    "alter" {
        every r = interp(node e1) do
            if r === phi then break
            else suspend r
        every r = interp(node e2) do
            if r === phi then break
            else suspend r
        suspend phi
        error(5)
        }
    "if_then" {
        r - interp(node e1)
        if r === phi
            then {
                r = interp(node e3)
                if r ~=== phi then suspend r
                }
            else {
                r = interp(node e2)
                if r ~=== phi then suspend r
                }
        suspend phi
        error(6)
        }
    "while_do" {
        repeat {
            r = interp(node e1)
            if r === phi then break
            interp(node e2)
            }
        suspend nullvalue
        suspend phi
        error(7)
        }
    "every_do" {
        every r = interp(node e1) do
            if r === phi then break
            else interp(node e2)
        suspend nullvalue
        suspend phi
        error(8)
        }
    "fails_" {
        r = interp(node e1)
        if r === phi then suspend nullvalue
        suspend phi
        error(19)
        }
    "compound" {
        interp(node e1)
        r - interp(node e2)
        if r ~=== phi then suspend r
        suspend phi
        error(9)
        }
```

```
        "limit" {
            r = interp(node e1)
            if r ~=== phi then suspend r
            suspend phi
            error(10)
            }
        default runtime(107)
        }
end

#  The procedure dounary(func,x1) evaluates the unary function func with the
#  argument x1  Although only one unary function is included here, the
#  procedure is organized so that others can be added

procedure dounary(func,x1)
    local r
    case func of {
        "-" {
            x1 = deref(x1)
            r = value(-x1 constant)
            suspend r
            suspend phi
            error(11)
            }
        default runtime(107)
        }
end

#  The procedure dobinary(func,x1,x2) evaluates the binary function func
#  with arguments x1 and x2

procedure dobinary(func,x1,x2)
    local temp, r
    case func of {
        "+" {
            x1 = deref(x1)
            x2 = deref(x2)
            r = value(x1 constant + x2 constant)
            suspend r
            suspend phi
            error(14)
            }
        ">" {
            x1 = deref(x1)
            x2 = deref(x2)
            if x1 constant > x2 constant then suspend x2
            suspend phi
            error(13)
            }
        " =" {
            if type(x1) ~== "variable" then runtime(121)
            sym[x1 name] = deref(x2)
            suspend x1
            suspend phi
            error(15)
            }
        "<-" {
            if type(x1) ~== "variable" then runtime(121)
            temp = sym[x1 name]
```

```
         sym[x1 name]  = deref(x2)
         suspend x1
         sym[x1 name]  = temp
         suspend phi
         error(16)
         }
      "&"  {
         suspend x2
         suspend phi
         error(12)
         }
      "to"  {
         x1  = deref(x1) constant
         x2  = deref(x2) constant
         while x1 <= x2 do {
            r  = value(x1)
            suspend r
            x1  = x1 + 1
            }
         suspend phi
         error(17)
         }
      default  runtime(107)
      }
end

#  The procedure deref(x) dereferences x  Note that x may be a value already,
#  in which case it is returned unmodified

procedure deref(x)
   case type(x) of {
      "variable"   return  sym[x name]
      "value"  return x
      default  error(18)
      }
end

#  The procedure error(n) terminates execution with an error message that
#  indicates the site in expevl that should be impossible to
#  reach (thus indicating an error in the coding of expevl)

procedure error(n)
   stop("internal inconsistency at site ",n)
end

#  The procedure runtime(n) terminates exection with an error message that
#  indicates a semantic error in the expression being evaluated  The error
#  number corresponds to the runtime error number in Icon itself

procedure runtime(n)
   stop("runtime error ",n)
end

#  The procedure main() initializes values and provides a loop that interprets
#  Icon expressions

procedure main()
   phi  = noresult()
   nullvalue  = value(0)
```

```
    sym := table() ::= nullvalue
    while line := read(&input) do
        if x := compile(line) then test(x)
        else write("erroneous input")
end
```

Appendix B — An SL5 Interpreter for Icon

The following section of code is a portion of an SL5 program that performs the same function as **expevl** It is included here primarily to illustrate a general paradigm for evaluation of an arbitrary number of arguments to a function

Since SL5 does not have a record facility, lists are used to represent Icon expressions The first element of the list is a string that identifies the type In the case of functions, the second element of the list is an SL5 procedure that interprets the corresponding Icon function For example, the expression x = 1 is represented by

```
["func",doeq,["variable","x"],["value",1]]
```

where the value of **doeq** is an SL5 procedure that implements the Icon conditional for numeric comparison **doeq** and **doconj** are included at the end of this program segment to illustrate how such procedures are coded

Because of the lack of some control structures in SL5, such as **break**, the coding is awkward in places Note that **break&0** generates a failure signal to break repeat loops (the identifier **break** was chosen to suggest the desired operation)

Note in particular the case selector for **func**, which contains the code that evaluates an arbitrary number of arguments Here a list corresponding to the number of arguments is formed and an environment for each of the argument expressions is created These environments are then resumed in left-to-right order to obtain the argument values If any argument fails to produce a value, the evaluation process reactivates the previous argument Similarly, if all arguments are evaluated successfully, but the function itself fails, the argument environments are reactived in reverse order

As in **expevl**, a compiler, which is not shown here, converts expressions to the required data structures

```
phi  =  ["noresult"],
nullvalue  =  ["value",0],

interp  =  procedure interp(node)
    private  n,  env,  args  i,  target,  r,  dir,
    case  node!1  of
        "value"  {
            return  node,
            return  phi,
            error(1),
            },
        "variable"  {
            return  node,
            return  phi,
            error(2),
            },
        "alter"  {
            env  =  create  interp  with  node!2,
            repeat  {
                r  =  resume  env,
                if  compare(r,phi)  then  break&0
                else  return  r,
                },
            env  =  create  interp  with  node!3,
            repeat  {
                r  =  resume  env,
                if  compare(r,phi)  then  break&0
                else  return  r,
                },
            return  phi,
            error(3),
            },
```

```
        "func" {
            n = length(node) - 2,
            env = list(n),
            args = list(n),
            dir = 1,
            i - 1,
            repeat {
                if i <= n then {                       # evaluate arguments
                    if dir = 1 then env!i = create interp with node!(i + 2),
                    args!i = resume env!i,
                    if compare(args!i,phi) then {
                        if i = 1 then {          # argument evaluation failure
                            return phi,
                            error(4),
                            }
                        else {
                            i = i - 1,           # backup
                            dir = 0,
                            },
                        }
                    else {
                        dir = 1,
                        i = i + 1,               # continue forward
                        }
                }
                else {                                 # invoke function
                    target = create node!2 with args,
                    repeat {
                        r = resume target,
                        if compare(r,phi) then break&0
                        else return r,
                        },
                    dir = 0,
                    i = n,
                    },
                },
          },
        default error(5),
        end,
end,

doeq = procedure doeq(args),
    args!1 = deref(args!1),
    args!2 = deref(args!2),
    if args!1!2 = args!2!2 then return args!2,
    return phi,
    error(7),
end,

doconj = procedure doconj(args),
    return args!2,
    return phi,
    error(6),
end,
```