An Expression Data Type for Icon*

Kenneth Walker

TR 86-20

September 9, 1986

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant DCR-8401831.

An Expression Data Type for Icon

1. Introduction

Icon [1] is one of a series of languages, starting with SNOBOL, for processing non-numeric (particularly string) data. SNOBOL4 [2] consists of two essentially separate languages: one for pattern matching and one for conventional computations [3]. The main goal of Icon's design was to integrate the goal-directed evaluation of SNOBOL4 pattern matching into a language containing conventional control structures.

Icon succeeded in meeting this goal, but something has been lost. In SNOBOL4 patterns are data objects. They can be constructed dynamically. Simple patterns can be created from strings and by using built-in functions. Complex patterns can be composed from simpler ones. The notation for manipulating patterns is concise and elegant. This notation can be compared to that of regular expressions or BNF grammars. Patterns capture the structure of a set of strings as opposed to the process of matching strings.

Icon replaces pattern matching with string scanning. The patterns themselves are replaced by matching expressions. These are expressions that obey a matching protocol [1]. Otherwise they are just ordinary Icon expressions and are purely syntactic constructions. Matching expressions describe the process of matching strings. Even though it is useful to view the strings matched as a set with certain characteristics, Icon matching expressions do nothing to encourage this abstraction.

This paper explores an approach to regaining what was lost in going from SNOBOL4 to Icon. The approach raises Icon expressions to the same level as SNOBOL4 patterns by creating an expression data type. Icon already has a co-expression data type, but it has the wrong semantics for this purpose. A co-expression activation produces at most one result even when the expression it was created from produces many. A co-expression maintains an evaluation environment separate from the one it is activated in. Co-expressions are used in situations where an expression generates a sequence of results and those results are needed in differently places in the program.

A feature for capturing and invoking expressions has been added to Icon. Except for variable scoping, the invocation of an expression behaves as if the expression had been copied to the invocation site.

2. Syntax and Semantics

The changes made to the language to support expressions are small. A control structure has been added to Icon for capturing expressions. It has the form

$ *expr*

$ has the same precedence as other unary operators. The element-generation operator ! has been extended to act as the invocation operator for expressions. For example, the code

```
out_of_range := $(sp < 0 | sp > stack_limit)
```

assigns the expression (sp < 0 | sp > stack_limit) to out_of_range. The expression can then be invoked later in the program. For example, as the condition of an if statement:

```
if !out_of_range then stop("error")
```

and in the control expression of a until statement:

```
until !out_of_range do pop_stk()
```

This example could have been coded using a procedure instead of using a captured expression:

```
procedure out_of_range()
   suspend (sp < 0 | sp > stack_limit)
end
```

and the invocation if the if statement would have been

```
if out_of_range() then stop("error")
```

However, there is a difference between procedures and expressions. A procedure has its own set of local variables. A new instance of these variables is created at each procedure invocation. An expression, on the other hand, shares local variables, including parameters, with the procedure invocation from which it is captured. This sharing of local variables is in contrast to what is done for co-expressions. The creation of a co-expression makes copies of the local variables. Consider the example:

```
procedure p()
   local e, i
   e := $(i := 2)
   i := 1
   !e
   write(i)
end
```

When the procedure p is invoked, new instances of the variables e and i are created. The expression (i := 2) is captured and assigned to e. The variable i in this expression is the same as the one in the procedure invocation. The invocation of e, !e, causes 2 to be assigned to this variable so the statement write(i) writes 2.

The same instance of these variables is used for each invocation of the expression, even when the expression outlives the procedure invocation from which it was captured. This means that the local variables can also outlive this invocation. Consider the following example.

```
global factor

procedure main()
   local e1, e2, e3
   e1 := p(1)
   e2 := p(2)
   e3 := p(3)
   factor := 3
   write(!e1, " ", !e2, " ", !e3)
end

procedure p(x)
   return $(x * factor)
end
```

Each expression e1, e2, and e3 is captured in a different invocation of the procedure p and therefore has a different variable x. On the other hand, the variable factor is global, and thus is shared by all three expressions. This program writes 3 6 9.

3. Simulating SNOBOL4 patterns

Icon and SNOBOL4 each have a rich repertoire of operations for doing string pattern matching. However, these operations are at different levels of abstraction. Icon matching expressions describe the process of matching a string, while the SNOBOL4 operations construct "higher level" patterns. The implementation of SNOBOL4 patterns uses matching functions and backtracking control structures, but these are hidden and not directly accessible to the programmer. Most of the patterns constructed by SNOBOL4's operations are easily simulated by Icon matching expressions.

The simplest SNOBOL4 pattern is a character string (which matches itself). The effect of this pattern can be accomplished with the Icon unary = operator. The SNOBOL4 statement

```
DEPT = "FINANCE"
```

can be simulated in Icon by (assuming DEPT is used as a pattern and not a string):

```
dept := $="FINANCE"
```

The SNOBOL4 pattern can be used in pattern matching to select input strings which contain the string FINANCE:

```
        DEPT = "FINANCE"
LOOP    S = INPUT                  :F(END)
        S DEPT                     :F(LOOP)
FOUND   OUTPUT = S                 :(LOOP)
END
```

An analogous Icon program is:

```
procedure main()
   local dept, s
   dept := $="FINANCE"
   while s := read() do
      if s ? (move(0 to *&subject) & !dept) then write(s)
end
```

In SNOBOL4, the pattern DEPT is automatically applied in the pattern matching context, however the Icon expression dept must be explicitly invoked during string scanning. The Icon expression move(0 to *&subject) is one way to simulate SNOBOL4's unanchored mode of pattern matching; without it !dept would only match a string at the beginning of s.

In SNOBOL4, the | operator represents pattern alternation. In Icon, | represents the alternation control structure. The SNOBOL4 statement

```
DEPT = "FINANCE" | "SHIPPING"
```

constructs a pattern which will match either the string FINANCE or the string SHIPPING and assigns the pattern to the variable DEPT. The Icon statement

```
dept := $(="FINANCE" | ="SHIPPING")
```

captures an expression which will match the same strings and assigns the expression to the variable dept.

In SNOBOL4, pattern concatenation is represented by a blank:

```
DEPT_ID = DEPT " DEPT."
```

In Icon, string concatenation is indicated with the || operator:

```
dept_id := $(!dept || =" DEPT.")
```

Note that in the SNOBOL4 version, the concatenation is done before the assignment. The variable DEPT is dereferenced at this time to obtain a pattern and this pattern is used to construct the larger pattern. On the other hand, in the Icon version the concatenation is really the concatenation of the *results* of two matching expressions. These matching expressions are not evaluated until the entire expression is evaluated during string scanning. As in the example above, the matching expression in dept must be explicitly invoked using !.

The built-in matching functions of Icon differ somewhat from those of SNOBOL4 but the capabilities are similar. For example, the SNOBOL4 statement

```
DIGITS = SPAN("0123456789")
```

can be simulated by the Icon statement

```
digits := $tab(many('0123456789'))
```

SNOBOL4 has four assignment operators, three of which are used in pattern matching. The operator $ is used for immediate assignment in patterns. The first operand is a pattern. The second operand is variable which is assigned the string matched by the pattern. The entire assignment expression is a pattern. For example, the

SNOBOL4 pattern

```
GETINT = " " (DIGITS $ I) " "
```

can be simulated by the Icon expression

```
getint := $(=" " || (i := !digits) || =" ")
```

The SNOBOL4 operator . is similar to $, but the assignment is only made after the entire pattern match is successful. It can often be simulated by the Icon operator <-. This operator does an immediate assignment which is undone if backtracking occurs. The effects of the assignments differs during pattern matching, but the ultimate effect (once the attempt at matching is complete) is the same.

SNOBOL4's unary operator @ assigns the current cursor position to its argument. It can be simulated by the Icon expression *var* := &pos.

SNOBOL4 uses dynamic scoping rules for variables, while Icon uses static (lexical) scoping. As long as the programmer is aware the this difference, it seldom, if ever, poses a problem for simulating SNOBOL4 pattern matching in Icon.

Captured expressions are very similar to SNOBOL4's unevaluated expressions. The $ control structure corresponds to SNOBOL4's unary * operator. The ! operator corresponds to SNOBOL4's EVAL function (although their corresponding effects on other data types are different). SNOBOL4's unevaluated expressions are automatically evaluated in a pattern matching context, while captured expressions must be explicitly invoked during string scanning as in other contexts. One important difference is that an Icon expression can produce a sequence of results, while a SNOBOL4 unevaluated expression produces at most one. However, a SNOBOL4 unevaluated expression may produce a pattern and that pattern may produces a sequence of results during pattern matching.

An important use of unevaluated expressions in SNOBOL4 is in the creation of recursive patterns. Because Icon expressions are by definition "unevaluated", recursive expressions can be written with no special mechanisms. Consider the following example (from [4]) to print each character from a string, one per line:

```
ROUT = LEN(1) $ OUTPUT *ROUT
```

Note that the SNOBOL4 version of the pattern expects the anchored mode of pattern matching. Icon string scanning is alway anchored, although a matching expression may simulate the unanchored mode as demonstrated in a previous example. The Icon version is:

```
rout := ${write(move(1)) || !rout}
```

The expression in this example is enclosed in braces to improve readability. Parentheses could just as well have been used to form the grouping.

One of the demonstrations of the expressive power of SNOBOL4 patterns is the ease with which a grammar can be converted into a set of patterns (possible recursive), which recognizes strings of the language described by that grammar. The process of converting a grammar into corresponding set of Icon expressions is also straightforward.

Consider a grammar for input to a simple spread-sheet program. The spread-sheet is a matrix of cells containing integer values. A cell is referenced by a pair of subscripts (integer constants) enclosed in square brackets, e.g. [3,2]. The input to the program is a formula that assigns an expression to a cell. An expression can be an if-then-else expression where the conditional test is "equal", "less than", or "less than or equal to". An expression also can be an arithmetic expression involving addition, subtraction, multiplication, division, and negation. The grammar is:

formula	::=	*subscrpt* := *expr*
expr	::=	if(*relation*)then(*expr*)else(*expr*) | *simexpr*
relation	::=	*expr* = *expr* | *expr* <= *expr* | *expr* < *expr*
simexpr	::=	*term* + *simexpr* | *term* - *simexpr* | *term*
term	::=	*factor* * *term* | *factor* / *term* | *factor*
factor	::=	- *basic* | *basic*
basic	::=	*digits* | *subscrpt* | (*expr*)
subscrpt	::=	[*digits* , *digits*]

The Icon program to recognize this grammar is

```
procedure main()
   local s
   local formula, expr, relation, simexpr, term, factor, basic, subscrpt, digits

   formula  := ${!subscrpt || =":=" || !expr}
   expr     := ${(="if(" || !relation || =")then(" || !expr || =")else(" ||
                 !expr || =")") | !simexpr}
   relation := ${(!expr || ="=" || !expr) | (!expr || ="<=" || !expr) |
                 (!expr || ="<" || !expr)}
   simexpr  := ${(!term || ="+" || !simexpr) | (!term || ="-" || !simexpr) |
                 !term}
   term     := ${(!factor || ="*" || !term) | (!factor || ="/" || !term) |
                 !factor}
   factor   := ${(="-" || !basic) | !basic}
   basic    := ${!digits | !subscrpt | (="(" || !expr || =")")}
   subscrpt := ${="[" || !digits || ="," || !digits || ="]"}
   digits   := $tab(many('0123456789'))

   while s := read() do
      if s ? !formula then
         write("valid formula")
      else
         write("error")
end
```

So far, examples of simulating SNOBOL4 patterns have used Icon expressions that defer all evaluation until string scanning is done. As noted above, this differs from SNOBOL4 patterns which are constructed before the actual pattern matching is started. It is possible to model the construction of SNOBOL4 patterns more closely.

It was demonstrated in Section 2 that an Icon procedure can return an expression whose characteristics depend on the arguments to that procedure. In fact, those arguments can themselves be expressions. This makes it possible to write Icon procedures that model SNOBOL4's concatenation and alternation operators along with a procedure to model SNOBOL4's ARBNO function. The first step is to write a procedure to coerce strings into "patterns".

```
procedure pattern(x)
   if type(x) == "expression" then return x
   if x := string(x) then return $=x
   stop("expression or string expected")
end
```

The procedure pattern is used by the procedures concat, alt, and arbno to ensure that their arguments are expressions. The procedure concat returns an expression that, when invoked, concatenates the results of invoking its two arguments. The procedure alt returns an expression that, when invoked and repeatedly resumed, produces the result sequence of its first argument followed by that of its second. The procedure arbno returns a recursive expression that, when invoked and repeatedly resumed, produces the concatenation of zero or more invocations of its argument:

```
procedure concat(e1, e2)
   e1 := pattern(e1)
   e2 := pattern(e2)
   return $(!e1 || !e2)
end
```

```
procedure alt(e1, e2)
   e1 := pattern(e1)
   e2 := pattern(e2)
   return $(!e1 | !e2)
end

procedure arbno(e)
   local rep_e
   e := pattern(e)
   return rep_e := $("" | (!e || !rep_e))
end
```

The following SNOBOL4 pattern matches a binary number enclosed in parenthesis.

```
X = "(" ARBNO("0" | "1") ")"
```

An analogous Icon expression constructed using the above procedures is:

```
x := concat("(", concat(arbno(alt("0", "1")), ")"))
```

SNOBOL4's syntax is simpler, but the pattern construction operations are a fixed primative part of the language. There is no way to simulate them with other features, nor is there any way to generize them. The ease with which they are simulated by Icon procedures demonstrates the expressive power of Icon. It is possible to write new pattern construction procedures in Icon beyond the operations of SNOBOL4. The generality of Icon expressions is explored farther in Section 4.

The last example could have been coded without using the procedures to construct the expression:

```
binary := $("" | (="0" | ="1") || !binary)
x := $(="(" || !binary || =")")
```

However, that is not true for all problems. Consider an Icon program that reads a file of keywords and constructs an expression that will match any of them.

```
procedure main()
   local file, keys, s

   file := open("keywords") | stop("cannot open keywords")
   keys := $&fail
   while keys := alt(keys, read(file))
   while s := read() do
      if s ? !keys then
         write("match")
      else
         write("no match")
end
```

&fail is a no-op in an alternation and provides a convenient starting point for building the expression. In the loop

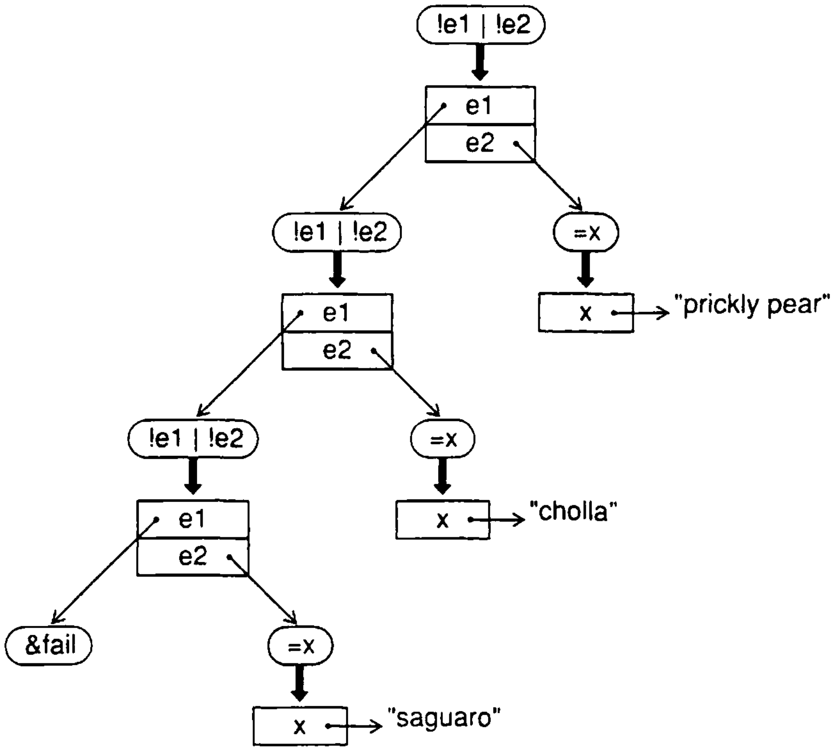
```
while keys := alt(keys, read(file))
```

the variable keys is used in the construction of its next value, but it is not a recursive definition. The variable must be dereferenced and its value used. This would not happen if keys were inside the expression being captured. Similarly, read must executed when the expression is constructed and not later when it is invoked. In addition, calling alt has the effect of allocating new variables (its parameters) to hold references to the two sub-expressions. Recall that these variables outlive the procedure invocation. This example dynamically creates a tree where the nodes are expressions and variables captured with each expressions reference the subtrees. Invocation of keys starts a tree walk which is driven forward by failure of the sub-expressions and resumption of the expression by the surrounding context (the nature of the tree walk is determined in part by the operator, |, used in the expressions).

Suppose the file keywords contains:

```
saguaro
cholla
prickly pear
```

When the file has been read, the variable keys will contain the expression represented by the tree:



Ovals contain the code for each expression and rectangles contain the variables for each expression. These are connected by bold arrows to form the complete expression. In general, variables can be shared among expressions, though sharing does not occur here. In the diagram, code is shown as if it were copied for each expression. In the implementation, it is really shared, but this does not affect the semantics. A thin arrows from each variable points to its value.

4. Beyond SNOBOL4 Patterns

Pattern matching in the style of SNOBOL4 is done using only concatenation, alternation, recursion, side effects (conditional and unconditional assignments) and a repertoire of built-in functions. There is no way for a SNOBOL4 programmer to write new matching functions. The Icon programmer is not limited to these operations. Any Icon operator, function, procedure, or control structure can be used during string scanning.

Some problems are awkward to solve using SNOBOL4 patterns. Consider Exercise 1.18 in [4]. The problem is to print all items in a list. The list is represented by a string with items separated by commas. There may or may not be a final comma. The solution given is:

```
NEXTI = BREAK(",") $ OUTPUT LEN(1) | (LEN(1) REM) $ OUTPUT ABORT | ABORT
GETI = NEXTI *GETI
PRINTI = POS(0) GETI
```

It is awkward to use ABORT to control backtracking. It takes some thought to see exactly what it is doing in the context of a given pattern. Contrast this with an analogous Icon solution:

```
nexti := ${(write(tab(upto(','))) & move(1)) | (not pos(0) & write(tab(0)))}
printi := $if !nexti then !printi
```

Once nexti has printed an item, it cannot be resumed because it is invoked within a bounded expression, the condition of the if expression. (Even better solutions can be obtained by using many(~',') rather than upto(',') and iteration rather than recursion.)

Another class of problems require putting pattern matching in a loop with conventional code; pattern matching extracts information from the string and the conventional code uses the information. In Icon, the loop and the conventional code can be incorporated with the matching. Consider the problem of reading entries into a table. In the input, the entries are of the form *key:value*. An input line may contain several entries separated by commas. A SNOBOL4 pattern to match an entry, extracting the key and value, is

```
GET_ENTRY = BREAK(":") $ KEY LEN(1) (BREAK(",") $ VAL LEN(1) | REM $ VAL)
```

The table can then be filled using a double loop. The outer loop reads input lines. The inner loop extracts information from each entry and removes the entry from the line:

```
          T = TABLE()
READLP    S = INPUT                        :F(EOF)
ITEMLP    S GET_ENTRY =                    :F(READLP)
          T<KEY> = VAL                     :(ITEMLP)
EOF
```

In Icon, entries can be added to the table during string scanning. Assuming the table is in the variable t, the following Icon expression will match an entry and put it in the table.

```
get_entry := ${t[tab(upto(':'))] := (move(1) & (tab(many(~',')) | ""))}
```

It is not necessary to remove the entry from the line. A loop can iterate over the entries without leaving string scanning. An expression to process an entire line is:

```
fill_tabl := $while !get_entry do move(1)
```

The table can now be read in with the code:

```
t := table()
while s := read() do
   s ? !fill_tabl
```

Using arbitrary Icon expressions in string scanning can simplify solutions to many problems, but in general these expressions will not obey the matching protocol. This protocol requires that matching expressions return the substring matched and preform data backtracking on &pos. In the last example, fill_tabl did not obey the matching protocol, but for that problem it did not matter.

For those problems where it does matter, the following procedure can be used. It wraps a matching protocol around an arbitrary expression.

```
procedure mtch_proto(expr)
    local init_pos
    return $(init_pos := (&pos <- &pos), !expr, &subject[init_pos : &pos])
end
```

The expression &pos <- &pos forces data backtracking on &pos.

Consider the problem of writing a matching expression to recognize strings in the classical context sensitive language $\{A^nB^nC^n \mid n \geq 0\}$. Given the auxiliary procedure

```
procedure span(c)
    suspend "" | tab(many(c))
end
```

the expression

```
        AnBnCn := ${(*span('A') = *span('B') = *span('C')) & pos(0)}
```

matchs the desired strings but does not produce the strings. mtch_proto can be used to solve the problem:

```
        procedure main()
            local AnBnCn, s

            AnBnCn := mtch_proto(${(*span('A') = *span('B') = *span('C')) & pos(0)})
            while s := read() do
               s ? write("\"", !AnBnCn, "\"")
        end
```

The usefulness of expressions is not limited to string scanning. There are many applications where the input includes some specifications that are used later in processing. The traditional approach to these applications is to store the specifications in a data structure and interpret them whenever they are needed. However, for some applications, it is simpler to translate the specifications into expressions and invoke the expressions whenever they are needed.

This is true of the spread-sheet program mentioned above. The following implementation is for a 3-by-3 spread-sheet. Three global lists associate information with each of the 9 cells. The value list holds the integer value of each cell. The dependent list associates a set of dependent formulas with each cell. For example, the formulas [1,1]:=3*[1,3] and [3,1]:=[1,3]-[2,3] are both dependent on cell [1,3]. These formulas are stored as expressions which implement them. Whenever the value of a cell changes, all the formulas in its dependent set are executed. In this way information is propagated throughout the spread-sheet, maintaining the relationships established by the formulas.

Whenever a formula for a cell is changed, the old formula must be cleared from any dependent sets it is in. The clear list contains an expression for each cell. This expression purges the current formula for that cell from all dependent sets it was put in.

Formulas are translated into expressions by augmenting the parser given in Section 3 with semantic functions. The global variables and the main procedure for the program are:

```
        global value, dependent, clear, ref_list

        procedure main()
           local s
           local formula, expr, relation, simexpr, term, factor, basic, subscrpt, digits

           formula  := ${ref_list := []; install(!subscrpt, =":=",  !expr)}
           expr     := ${ifexpr(="if(", !relation, =")then(", !expr, =")else(", !expr,
                           =")") | !simexpr}
           relation := ${eql(!expr, ="=", !expr) | leq(!expr, ="<=", !expr) |
                           lss(!expr, ="<", !expr)}
           simexpr  := ${plus(!term, ="+", !simexpr) | minus(!term, ="-", !simexpr) |
                           !term}
           term     := ${times(!factor, ="*", !term) | divide(!factor, ="/", !term) |
                           !factor}
           factor   := ${neg(="-", !basic) | !basic}
           basic    := ${num(!digits) | deref(!subscrpt) | 2(="(", !expr, =")")}
           subscrpt := $subconv(="[", !digits, =",", !digits, ="]")
           digits   := $integer(tab(many('0123456789')))
```

```
      value := list(9, 0)
      dependent := list(9)
      every dependent[1 to 9] := set([])
      clear := list(9, $&null)

      while s := read() do
         s ? ((!formula & print()) | write("invalid formula"))
   end
```

The semantic function used in digits is just the built-in function integer which translates the character string into the corresponding integer. The semantic function used in basic for a number is the procedure:

```
   procedure num(n)
      return $n
   end
```

It takes an integer (produced by digits) as an argument and simply returns an expression which evaluates to that integer.

The semantic function for addition is a procedure which takes two subexpressions and returns an expression which evaluates to the sum of the numbers obtained by evaluating the two subexpressions.

```
   procedure plus(e1, op, e2)
      return $(!e1 + !e2)
   end
```

The semantic function for the if expression returns a captured if expression with subexpressions invoked in the proper places.

```
   procedure ifexpr(s1, rel, s2, e1, s3, e2)
      return $if !rel then !e1 else !e2
   end
```

The semantic functions for subtraction, multiplication, division, negation, and the relations are similar, but they are not shown here. The print procedure for displaying the spread-sheet is also not shown.

The semantic function for subscripting converts the two subscripts into the list index for the cell. It also does a range check. Note that this semantic function returns an integer rather than an expression.

```
   procedure subconv(s1, i1, s2, i2, s3)
      if (1 <= i1 <= 3) & (1 <= i2 <= 3) then
         return (i1 - 1) * 3 + i2
      else {
         write("subscript out of range")
         fail
         }
   end
```

The semantic function for dereferencing a cell has the added task of creating a list of the cells that the current formula references; that is, the ones it depends on. Note that ref_list is initialized at the start of formula.

```
   procedure deref(cell)
      put(ref_list, cell)
      return $value[cell]
   end
```

The most involved semantic actions are preformed by the procedure install, which finishes building the formula and updates the global lists. The new formula, update, computes the value, updates the cell, and invokes any dependent formulas. It only does the update if the value of the cell changes. This allows circular dependencies among cells as long as computations converge.

Once the new formula has been created, any dependencies for the old formula are cleared and new dependencies are established along with a new clear expression. Finally the formula is executed.

```
procedure install(cell, s, e)
   local update, new_val

   if not pos(0) then fail
   update := ${
      new_val := !e
      if new_val ~= value[cell] then {
         value[cell] := new_val
         every !!dependent[cell]
         }
      }
   # establish new dependencies
   !clear[cell]
   clear[cell] := $&null
   every new_depend(update, cell, !ref_list)
   !update
   return
end

procedure new_depend(update, cell, ref_cell)
   local part_clear
   insert(dependent[ref_cell], update)
   part_clear := clear[cell]
   clear[cell] := $(!part_clear & delete(dependent[ref_cell], update))
   return
end
```

5. Implementation

Previously, the lifetime of local variables was the same a that of the procedure invocation they belonged to, so they could be allocated on the interpreter stack [5]. With the addition of expressions to Icon, local variables must exist as long as they can be referenced through an expression. This requires allocating local variables on the heap. An internal data type locals, similar to the record data type, has been added to Icon for this purpose. When a procedure is invoked, a locals block is allocated for it.

The expression data type is implemented as a block with two fields. One field points to the code for the expression and the other points to the locals block for the expression. When $ captures an expression, it allocates a new block and fills in the fields. The instruction pointer for the expression's code is fixed for each instance of $ and the locals pointer comes from the current interpreter state.

An expression is analogous to a procedure body. The code needed to implement it is similar to that for

```
suspend expr
```

Invocation is simpler for an expression than for a procedure because there are no arguments and the local variables have already been allocated. Invocation copies the fields from the expression block into the corresponding interpreter state variables, saving on the stack the information needed to return.

6. Conclusions

The expression data type is a valuable addition to Icon just as the pattern data type was a valuable feature of SNOBOL4. It allows pattern matching problems to be factored into manageable pieces and the corresponding matching expressions assigned to variables with suggestive names. This is analogous to breaking a complex numerical expression into subexpressions and computing each into a variable with a mnemonic name. In theory, procedures could be used to divide up a pattern matching problem, but in practice procedure syntax is too bulky for small expressions. Captured expressions on the other hand employ a very concise syntax.

Icon's captured expressions are similar to SNOBOL4's unevaluated expressions. For most problems a pattern may be an unevaluated expression, but for some problems parts of the pattern must be evaluated when the pattern is created. Icon programs can use procedures to construct expressions. The arguments to these procedures correspond to the evaluated parts of SNOBOL4 patterns.

Icon is an attempt to demonstrate that the separation of pattern matching and conventional computations found in SNOBOL4 is unnecessary. The expression data type fills in a major gap in that demonstration. In addition, expressions have proven useful for storing some kinds of data in executable form, eliminating the needed to interpret data structures when the information is needed.

Acknowledgements

The idea of capturing expressions in Icon originated from discussions with Dave Gudeman. Dave, Ralph Griswold, Janalee O'Bagy, and Kelvin Nilsen participated in discussions on the implementation. Ralph Griswold provided numerous helpful suggestions on the presentation of this report.

References

1. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
2. R. E. Griswold, J. F. Poage and I. P. Polonsky, *The SNOBOL4 Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, second edition, 1971.
3. R. E. Griswold and D. R. Hanson, "An Alternative to the Use of Patterns in String Processing", *ACM Trans. Prog. Lang. and Systems 2*, 2 (1980), 153-172.
4. R. E. Griswold, *String and List Processing in SNOBOL4; Techniques and Applications*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1975.
5. R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language*, Princeton University Press. In press.