Integrating Pattern Matching Within String Scanning

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

John H. Goettsche

Major Professor: Clinton Jeffery, Ph.D.

Committee Members: Robert Heckendorn, Ph.D.; Robert Rinker, Ph.D.

Department Administrator: Gregory Donohoe, Ph.D.

June, 2015

Authorization to Submit Thesis

This Thesis of John H. Goettsche, submitted forthe degree of Master of Science with a Major in Computer Science and titled "Integrating Pattern Matching Within String Scanning," has been reviewed in final form. Permission, as indicated by the signatures and dates below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor: _____ Date: _____
Clinton Jeffery, Ph.D.

Committee Members: _____ Date: _____
Robert Heckendorn, Ph.D.

_____ Date: _____
Robert Rinker, Ph.D.

Department Administrator: _____ Date: _____
Gregory Donohoe, Ph.D.

Abstract

A SNOBOL4 like pattern data type and pattern matching operation were introduced to the Unicon language in 2005, but patterns were not integrated with the Unicon string scanning control structure and hence, the SNOBOL style patterns were not adopted as part of the language at that time. The goal of this project is to make the pattern data type accessible to the Unicon string scanning control structure and vice versa; and also make the pattern operators and functions lexically consistent with Unicon. To accomplish these goals, a Unicon tabmat operator was changed to allow the execution of a pattern match in the anchored mode, pattern matching unevaluated expressions were revised to handle complex string scanning functions, and the pattern matching lexemes were revised to be more consistent with the Unicon language.

Acknowledgements

I would like to acknowledge my major professor Clinton Jeffery who guided my research on this project. I would also like to acknowledge the Special Information Services department of the National Library of Medicine for the financial support that made my research possible.

Dedication

To all my fellow Land Surveyors who have searched for or sought to develop software that is used the way surveyors actually work.

Table of Contents

List of Tables

List of Figures

1 Introduction

In order to enhance programmers' productivity in analyzing strings, many different string scanning and pattern matching systems have been developed. Modern high level languages tend to offer regular expressions and context free grammars for their string processing. Analysis of strings often requires functionality beyond what these classes of languages offer. SNOBOL4 was the most successful of the early string processing languages [1, 2]. A pattern data type was employed in SNOBOL4 along with pattern operators and functions to perform the analysis.

Icon is a successor to SNOBOL4, which expanded upon the goal directed evaluation with generators and backtracking that was implicit in SNOBOL4's pattern matching [2, 3]. It uses a string scanning control structure that passes through the subject string with a set of commands to manipulate and analyze its contents. When Unicon was developed, its core elements came directly from Icon, including its string scanning facilities [5]. Many of the Icon string scanning functions resemble SNOBOL4 patterns, but they are different in both how they are processed as well as their functionality. Many Icon programmers have expressed a desire for the functionality of SNOBOL4 patterns [4].

While SNOBOL patterns are instances of a structured data type that are used deductively to test whether the pattern exists within a subject string, the string scanning environment uses a set of functions to inductively analyze or extract data from the subject string. Patterns are pre-defined and applied later, while string scanning is performed as string scanning functions are executed. Patterns allow

composition and re-use in more flexible ways than is the case for code, which only has procedure and co-expression granularity. A co-expression is an independent, encapsulated thread-like context, where the results of an expression are retrieved one at a time [5].

This paper briefly explores the background of SNOBOL4 patterns and Unicon string scanning functions. It discusses design considerations for integrating SNOBOL4 patterns within Unicon string scanning environments, a description of how proposed changes were implemented, and the benchmarks used to evaluate the success of the integration.

2 Background

SNOBOL was developed by David Farber, Ralph Griswold and Ivan Polonsky at Bell Telephone Laboratories in 1962. SNOBOL4 was developed in 1967 and had many of the features that are included in popular dynamic programming languages including dynamic typing, eval and garbage collection. Its pattern data type was its most important contribution to string processing. Patterns could be as simple as a single character or a set of characters in a particular order, or they could be a complex arrangement with alternative character sets and pattern functions. The pattern data type enabled the user to define and store patterns in variables to be used later when they were desired [1].

One of the developers of SNOBOL4, Ralph Griswold, went to the University of Arizona and developed the Icon programming language which was more readable and simpler to use [5]. Griswold used his experience with SNOBOL4's generators and backtracking in its pattern matching to develop and implement goal directed evaluation [2]. SNOBOL4 patterns were not incorporated in Icon. Instead Griswold developed an extensive string scanning system where a variety of functions and operations are executed in order to analyze and manipulate a subject string as the cursor location advances through said string. Unlike most other languages, Icon considers the string as a primitive data type in its own right, rather than an array of characters [3].

Using the same Icon source code, Unicon was developed to include modern software features such as objects, networks and databases [5]. The string scanning

control structure of Icon is a part of the Unicon programming language, yet the desire of researchers and developers for SNOBOL4's pattern data structure and pattern matching environment has persisted. Sudarshan Gaikaiwari adopted SNOBOL4 patterns to the Unicon language for his master's thesis in 2005. In his thesis, he added the pattern data type, and provided pattern matching functions and operators to execute the pattern searches [2]. The pattern data type was kept separate from the string scanning environment, which may not execute pattern matching operations. This paper refines Gaikaiwari's work to be more naturally incorporated in the Unicon language, allowing string scanning environments to utilize the pattern matching functionality and vice-versa.

2.1 Pattern Matching

The purpose of a pattern matching operation is to determine if the presence of a predefined pattern exists within a subject string. The pattern matching operation is initialized in Unicon by using the `??` operator [2]. The following expression results in the `subject` having a pattern match performed with `pattern`:

```
subject ?? pattern
```

If the pattern is found in `subject` then the expression succeeds and the first substring matching the pattern is returned; otherwise it fails.

A pattern is constructed by assigning it to a series of simple pattern components and pattern operators, alternation `.|` and concatenation `||`. An alteration operation succeeds if either of the patterns on either side of the operator is found. A concatenation operation succeeds if the pattern on the left is immediately followed

by the pattern on the right.

For example, if the user wanted to define a pattern called "pat" that identifies whether "bean" or "been" as in a subject string. The follow code would produce that result:

```
pat := "b" || ( "ea" .| "ee" ) || "n"
```

The figure below illustrates the pattern definition. when the pattern match is per-

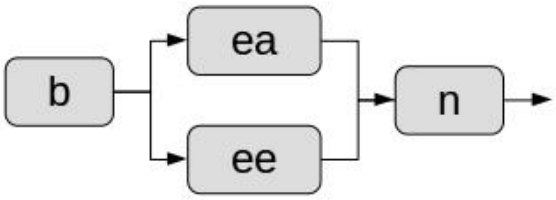

Figure 1: Pattern Example

formed on a subject string, it will first search for a "b". When it finds one, it will then check to see if it is followed by "ea". If it fails it will backtrack to the successful "b" and check to see if it is followed by "ee". If it fails again, it will backtrack again. There not being another option, it will backtrack again past the "b" and will search for another "b" and do the process again until it succeeds at each of the three parts of the pattern.

Lets consider the string "I've been there!". It will iterate through each of the letters until it reaches the "b". Then it will not match "ea" and then try "ee" where it succeeds. Then it will check for an "n" and succeeds at the end of the pattern and will suspend "been".

3 Design Considerations

To integrate pattern matching with Unicon string scanning required consideration of how a user would execute a pattern match in the string scanning environment and string scanning functions in the pattern matching operation. The following questions had to be addressed. What parameters are necessary for their execution? From what environment are they being called? How are they going to be utilized and under what conditions for each implementation? Also, the pattern functions and operators will have to be consistent lexically and functionally with the Unicon language.

3.1 Pattern matching statements

SNOBOL pattern matching can be executed in either anchored or non-anchored mode. The anchored mode requires the match to start on the first character of the subject string while in the non-anchored mode the match can start at any location in the subject string [1]. The pattern matching operation adapted by Gaikaiwari are generators and operate in the non-anchored mode [2]. They only produce one value at a time, but since they suspend instead of return that value, the cursor position is stored so the next pattern can be applied to the remainder of the string to produce the next value [5].

The pattern matching statements in SNOBOL4 and Gaikaiwari's Unicon implementation are in the following from:

SNOBOL4	Gaikaiwari's Unicon
`SUBJECT PATTERN`	`subject ?? pattern`

In both examples, the subject is scanned to see if it contains the pattern. If it succeeds, then a substring of the subject that fits the pattern is produced. Gaikaiwari's syntax starts the pattern matching operation with the use of the `??` operator while SNOBOL4 uses two spaces between the subject and pattern. Since the Unicon string scanning operator is `?` and used in the following syntax:

```
subject ? expr
```

Gaikaiwari's operator in the following syntax:

```
subject ?? pattern
```

is a better match to Unicon lexically and syntactically, and is easier to read than separating the subject and pattern with white space.

To integrate pattern matching into the Unicon language, it is necessary to consider which mode is appropriate considering the current environment. If it is being executed outside of a Unicon string scanning environment, the cursor position or index of the string has not been established. Therefore the non-anchored mode is appropriate for basic pattern matching operations. This allows a pattern to be matched anywhere within the subject string.

In the anchored mode the pattern match begins at the current cursor or index location. If the pattern fails at the first character then the entire pattern will fail to match and will not look for an alternate match later in the subject string.

Sometimes the user wishes to start the pattern match at the first location in the subject string, as though it was in the anchored mode. In these situations, defining the pattern with the first element as `Pos(1)`, has the desired effect.

3.1.1 Pattern matching within String Scanning

In the Unicon string scanning environment the functions operate in relation to the cursor position of the subject string. It is an inductive process where the data is analysed by executing a series of functions on the subject string. The cursor location or index is adjusted depending on the success or failure of each expression. When executing pattern matching from within a Unicon string scanning environment, the cursor or index location is established. Therefore, in order to maintain this indexing process in the string scanning environment, it was decided to execute the pattern matching operation in the anchored mode.

Unicon's `?` operator sets the cursor location to the first position in the subject string and then the scanning expression, usually a block expression containing a series of calls to scanning functions, is executed. Scanning functions normally move the cursor upon success. For this reason it was decided, in the event that a pattern is encountered with the tabmat `=` operator, the pattern match is performed in the anchored mode, with the cursor being advanced to the end of the matching pattern if there is success.

3.1.2 String Scanning within Patterns

Patterns are pre-defined and are used deductively to search a subject string for the pattern. A regular pattern match is performed in the un-anchored mode and

establishes the cursor locations for the beginning and end of a pattern when it is found. This is done by iterating through the subject string until a match is found. It is possible for a string scanning function to be performed as the pattern matching process iterates through the cursor or index locations.

Gaikaiwari's pattern matching facilities allow simple single depth function calls to be made with an unevaluated expression. His unevaluated expressions can handle procedure calls such as `‘foo(2 , "bar")‘` but can not handle function calls from within function calls like `tab(upto(somecset))` [2]. Therefore the unevaluated expressions implementation needed to be revised to handle more complex functions calls which are common when performing string scanning operations.

3.2 SNOBOL4 and Unicon pattern operators

The pattern operators for Unicon were defined by Gaikaiwari in his Master's Thesis. Although they are lexically different, they are functionally identical to SNOBOL4 pattern operators. The pattern concatenation operator is similar to the string concatenation operator in that it has the end result of the first pattern or string being followed by the second pattern or string. But, they differ in that string concatenation joins two strings, while pattern concatenation finds a pattern in the subject string that is followed by the second pattern.

Table 1: Pattern Operators

Operation	SNOBOL4	Gaikaiwari	New
Concatenation	<<implicit>>	&&	||
Alternation	|	.|	.|
Immediate Assignment	$	$$	=>
Conditional Assignment	.	− >	− >
Cursor Assignment	@	.$	.>
Unevaluated Expression	∗x	‘x‘	‘x‘

The Unicon operator for string concatenation was modified to recognize whether the expression contains a pattern. Since the pattern concatenation and Unicon's string concatenation operators do not have the same order of precedence, the assignment operators' order of precedence was adjusted to maintain their semantics. The cursor assignment has not changed in that it must be separated from the pattern functions and unevaluated expressions with a concatenation operator.

The assignment operators were changed lexically to use the `>` symbol in the right position of the lexeme to represent an assignment within patterns. This makes them more consistent lexically. The SNOBOL4 lexemes appear almost random in their selection and Gaikaiwari's also lacked consistency. The Immediate Assignment, Conditional Assignment and Cursor assignment are `=>`, `->` and `.>` respectively.

3.3 SNOBOL4 and Unicon pattern functions

Since the demand for SNOBOL4 patterns to be added to Unicon is coming from SNOBOL4 users, it is important that the Unicon pattern functions appear as similar to SNOBOL4 pattern functions as possible, while being lexically consistent with Unicon. Gaikaiwari's pattern functions are functionally the same as SNOBOL4 patterns, but appear somewhat disconnected from Unicon lexically. The way in which Unicon handles its csets and strings makes some functions redundant or inconsistent with Unicon, therefore some of the pattern functions will be removed.

The table below shows the SNOBOL4 primitive functions and the new Unicon pattern functions. In most cases, the function is lexically similar to SNOBOL4 with the first character being capitalized and the following letters in lower-case, with

exceptions for FAIL and ABORT. Other changes are described in the example uses of the functions below:

Table 2: Pattern Functions

SNOBOL4	Gaikaiwari	New
LEN(n)	PLen(n)	Len(n)
SPAN(c)	PSpan(c)	Span(c)
BREAK(c)	PBreak(c)	Break(c)
ANY(c)	PAny(c)	Any(c)*
NOTANY(c)	PNotAny(c)	
TAB(n)	PTab(n)	Tab(n)**
RTAB(n)	PRtab(n)	
REM	PRest()	Rem()
POS(n)	PPos(n)	Pos(n)**
RPOS(n)	PRpos(n)	
FAIL	PFail()	Back()***
FENCE	PFence()	Fence()
ABORT	PAbort()	Cancel()***
ARB	PArb()	Arb()
ARBNO(p)	PArbno(p)	Arbno(p)
BAL	PBal()	Bal()

* see logically redundant functions described in subsection 3.3.1 below

** see index related functions described in subsection 3.3.2 below

*** see backtracking and terminating functions in subsection 3.3.3 below

3.3.1 Complements with functions

The complement operator allows the user to get the complement of a given cset, or a cset containing all the characters not included in the given cset. Therefore the functionality of the `NotAny(c)` function can be achieved by using the complement operator with a cset in the `Any` function as `Any(~c)`. The run time for each method is the same on average.

3.3.2 Index Related Functions

The `POS(n)`, `RPOS(n)`, `TAB(n)` and `RTAB(n)` SNOBOL4 functions all work directly with the cursor location or index. In Unicon the index value is the number of spaces to the right from the left end of string with the first position being 1 or starting at the right end of the string starting with zero and subtracting the number of spaces to the right end of the string [5]. The illustration below demonstrates the Unicon cursor position values for the string "Unicon" with the vertical bars representing the index locations:

```
-6  -5  -4  -3  -2  -1  0
| U | n | i | c | o | n |
1   2   3   4   5   6   7
```

The SNOBOL4 cursor locations for the `RPOS(n)` and `RTAB(n)` functions the cursor locations are as follows:

```
RPOS(n) & RTAB(n)    6   5   4   3   2   1   0
                     | S | N | O | B | O | L |
POS(n) & TAB(n)      1   2   3   4   5   6   7
```

Integration of the SNOBOL4 `RPOS(n)` and `RTAB(n)` functions with the Unicon string indexes can be achieved with `Pos(-n)` and `Tab(-n)` functions, making `RPOS(n)` and `RTAB(n)` redundant.

3.3.3 Backtracking and Terminating Functions

`fail` is already taken as a reserved word in Unicon. In the internal C code of the implementation `Fail` is a macro. Attempts to use this name for a built in function

created a lot problems in compiling Unicon. Fail in Unicon means that there is not a successful result in the operation. While performing the pattern matching operation, `FAIL` is used to signify that there is not a successful result in the current pattern element and instructs the system to backtrack and to try another alternative [1]. Since the function is localized to a pattern match element and is not intended for a failure of the entire operation, it makes sense to use `Back()` for the SNOBOL4 `FAIL` function.

Likewise `ABORT` is a SNOBOL function that cancels the pattern matching operation, but does not halt the operation of the entire program. `Cancel()` is a more appropriate term for the `ABORT` function.

4 Implementation

The figure below shows the relationship of the Unicon tools for translation, compiling and execution of a Unicon program. The front-end compiler, called unicon, translates the unicon code into icon code [6]. This is where the pattern definitions and their component parts are identified and converted to their respective pattern function calls for the runtime system [6]. The function lists are generated for the unevaluated expressions. Icont compile Icon code down to virtual machine and C code [6]. Iconx is the Icon and Unicon virtual machine [6], where the `pattern_match()`, `ResolveReferences` and `ResolveList` functions are executed.

Figure 2: Unicon tools organization

To implement the integration of patterns and string scanning, the following changes to the Unicon language had to be made:

- Modify Unicon's runtime to execute pattern matching in the anchored mode when a pattern operand is used with the tabmat operator in a string scanning environment.
- Modify the unevaluated expression operand in the pattern matching operation to handle nested function calls within parameters; as used in string scanning function calls.

- Modify patterns to work off of the &subject and &pos keywords.
- Modify the pattern source files to address the functional changes for `Pos(n)` and `Tab(n)`.
- Modify function definitions for the Unicon build.
- Modify the concatenation operator to function with patterns.

4.1 Pattern matching statements

The pattern matching operation `&&` in Gaikaiwari's 2005 Master's thesis was performed in the non-anchored mode. The anchored pattern matching operation was defined in the pattern C library code as a part of the `internal_match` operation, but the `Anchored_Mode` identifier was set to false. The default location of the index was set to 1. The arguments for the `internal match` operation were changed to include both the mode and cursor location. This allows the `internal_match` to be called and initiated from any location of the subject string.

4.1.1 Anchored mode from string scanning

Integrating the pattern matching system into the Unicon string scanning environment requires that the pattern matching be performed in the anchored mode, since the string scanning functions are operate from the index or cursor position. For this implementation the Unicon tabmat operator `=` was determined to be an ideal choice for initiating a pattern match in a string scanning environment. The use of an equals `=` before a pattern variable triggers the anchored mode pattern matching operation.

```
   subjectString ?
      match := =pattern
```

The tabmat operator was modified to accept operands of type pattern. In the event that a pattern is its argument, tabmat initiates a pattern match in the anchored mode, otherwise it functions normally. This section of RTL (Real Time Language) code from omisc.r in Unicon's runtime system identifies whether the argument is a pattern or a string and assigns its return value, and performs the assignments required for the pattern match.

```
operator{*} = tabmat(x)
   if is:pattern(x) then {
      abstract {
         return string
         }
      body {
         int oldpos;
         int start;
         int stop;
         struct b_pattern *pattern = NULL;
         tended struct b_pelem *phead = NULL;
         char * pattern_subject;
         int subject_len;
         int new_len;
         CURTSTATE();

         /*
          * set cursor position, and subject to match
          */
         oldpos = k_pos;
         pattern_subject = StrLoc(k_subject);
         subject_len = StrLen(k_subject);
         pattern = (struct b_pattern *)BlkD(x, Pattern);
         phead = (struct b_pelem *)ResolvePattern(pattern);

         /*
          * runs a pattern match in the Anchored Mode and returns
          * a sub-string if it succeeds.
          */
```

```
      if (internal_match(pattern_subject, subject_len,
            pattern->stck_size, phead, &start, &stop,
            k_pos - 1, 1)){
         /*
          * Set new &pos.
          */
         k_pos = stop + 1;
         EVVal(k_pos, E_Spos);
         oldpos = k_pos;

         /*
          * Suspend sub-string that matches pattern.
          */
         suspend string(stop - start, StrLoc(k_subject)+ start);
         pattern_subject = StrLoc(k_subject);
         if (subject_len != StrLen(k_subject)) {
            k_pos += StrLen(k_subject) - subject_len;
            subject_len = StrLen(k_subject);
            }
         }

      /*
       * If tab is resumed, restore the old position and fail.
       */
      if (oldpos > StrLen(k_subject) + 1){
         runerr(205, kywd_pos);
         }
      else {
         k_pos = oldpos;
         EVVal(k_pos, E_Spos);
         }
      fail;
      }
   }
```

As shown above, the tabmat operator checks to see if the parameter is a pattern. If it is, the `k_pos` and `k_subject` which were assigned `&pos` and `&subject` respectively when `CURTSTATE()` was called. They are converted to the approriate type so they can be passed as parameters when the `internal_match()` function is called in the anchored mode. The index or cursor location had to be assigned so that the anchored pattern match would begin where the string scanning had left off.

Finally, if the pattern was successful, then it would have to suspend the matching pattern and update the index or cursor location. If it failed then it would revert to the previous index or cursor location.

4.1.2 String Scanning Functions as Unevaluated Expressions

In Gaikaiwari's implementation of patterns, functions can be called in a pattern using the unevaluated expression notation by placing the function call in back quotes, for example: `pattern := ‘tab(3)‘`. This allows the user to assign simple Unicon string scanning function calls and procedure calls in the pattern definition, but only if `tab(3)` impacts on subsequent pattern match operations.

The first phase of the analysis of an unevaluated expression occurs during compile time. In this process, the `process_uneval()` procedure calls the `emit_code_for_uneval()` procedure which generates a list to be interpreted during runtime. Both of these procedures are found in the tree.icn file of the Unicon implementation.

```
procedure emit_code_for_uneval(funcname)
   L := []
   tab(many("‘"))
   temp := tab(upto("("))
   if \temp == "\\" then temp := "\\\\"
   put(L,\temp)
   while tab(upto(&letters)) do {
      temp := tab(many(&letters))
      put(L,temp)
      }
   writes(yyout,funcname, "(")
   writes(yyout, "[")
   every temp := !L\ (*L -1) do {
      writes(yyout, "\"", temp,"\",")
      }
```

```
   writes(yyout, "\"", L[*L], "\"")
   writes(yyout, "])")
   return
end
```

The list generated in this process is similar to LISP in that the first element is the function to be called and the remaining elements are the parameters. Unfortunately, Gaikaiwari's implementation could not handle functions or procedure calls as parameters within another function or procedure call as is common with some string scanning functions. For example `tab(upto('e'))`, the tab function requires an integer value which is acquired by the `upto()` function that scans a string until it reaches an 'e' character. Gaikaiwari's procedure would generate a list `["tab", "upto", "e"]`, where `upto` would not be a function call, but instead treated as a string. This procedure had to be revised to handle function calls as parameters in a function call. To navigate nested function calls, a recursive procedure generating the list for each function that is used as a parameter in the parent function. It was implemented with the following procedures:

```
procedure writes_code_for_uneval(L)
local i
   writes(yyout, "[")
   writes("[")
   every temp := L[i := 1 to *L] do {
      if type(temp) == "list" then {
         writes_code_for_uneval(temp)
         }
      else {
         writes(yyout, image(temp))
         if i = 1 then {
            /list_of_invocables := []
            put(list_of_invocables, temp)
            }
         if i < *L then writes(yyout, ",")
         }
```

```
      }
   writes(yyout, "]")
   return
end

procedure make_list_for_uneval(L, word)
   if tab(upto(word)) then {
      temp := tab(many(word))
      if proc(temp) then {
         L1 := []
         put(L1, temp)
         L1 := make_list_for_uneval(L1, word)
         put(L, L1)
         }
      else {
         put(L, temp)
         }
      }
   return L
end

procedure emit_code_for_uneval(funcname)
   L := []
   tab(many('`'))
   temp := tab(upto('(.'))
   if \temp == "\\" then temp := "\\\\"
   put(L, \temp)
   word := &letters ++ &digits ++ '\'\"&~'
   L := make_list_for_uneval(L, word)
   writes(yyout, funcname, "(")
   writes_code_for_uneval(L)
   writes(yyout, ")")
   return
end
```

It is necessary that variables, strings and csets be differentiated. This was achieved by including within the generated string double quotes around strings and single quotes around csets, and no quotes for variables. In this case, when a the string scanning function `tab(upto('e'))` is called, it will generate the list: `["tab", ["upto", "'e'"]]`, which then can be interpreted with a list with the `tab()` function along with a parameter containing a list with the `upto()` function with the parameter

being the cset containing 'e'.

To evaluate this list of lists during runtime a ResolveList function was added to the fxpattrn.ri file in the Unicon runtime implementation.

```
struct b_list *ResolveList(struct b_list *lp)
{
   struct descrip proc;
   struct descrip var;
   tended struct b_lelem *elsrc;
   tended struct b_lelem *eldest;
   int i, b, nargs;
   tended struct b_list *lpsrc;
   tended struct b_list *lpdest;
   struct b_list *lptemp;
   tended char *temp;
   int complement;
   struct b_cset *cp, *cpx;

   lpsrc = lp;
   lpdest = alclist(lpsrc->size,lpsrc->size);
   nargs = lpsrc->size -1;
   elsrc = (struct b_lelem *)lpsrc->listhead;
   proc.dword = D_Proc;
   /* convert string in to process and store in proc*/
   BlkLoc(proc)= (union block *)strprc(&elsrc->lslots[0],nargs);
   if (BlkLoc(proc) == NULL) {
      fprintf(stderr, "Unable to find proc1\n");fflush(stdout);
      fatalerr(0, NULL);
      }
   eldest = (struct b_lelem *)lpdest->listhead;
   eldest->lslots[0] = proc;
   for (; BlkType(elsrc) == T_Lelem; elsrc =
         (struct b_lelem *)elsrc->listnext) {
      for (i = 1; i < elsrc->nused; i++) {
         tended char * varname;
         struct descrip parm;
         dptr pvar;
         if (is:string(elsrc->lslots[i])) {
            /* if a string constant, drop double quotes,
            else lookup using getvar() */
            cnv:C_string(elsrc->lslots[i], varname);
            if (StrLen(elsrc->lslots[i])>0) {
               complement = 0;
               if (strcspn(varname, "~") == 0) {
                  /* resolve complement */
```

```
   StrLoc(elsrc->lslots[i]) =
         StrLoc(elsrc->lslots[i]) + 1;
   StrLen(elsrc->lslots[i]) =
         StrLen(elsrc->lslots[i]) - 1;
   cnv:C_string(elsrc->lslots[i], varname);
   complement = 1;
   }
if (strcspn(varname, "\"") == 0) {
   /* drop the quotes, but pass string as a string */
   StrLoc(elsrc->lslots[i]) =
         StrLoc(elsrc->lslots[i]) + 1;
   StrLen(elsrc->lslots[i]) =
         StrLen(elsrc->lslots[i]) - 2;
   cnv:string(elsrc->lslots[i], elsrc->lslots[i]);
   }
else if (strcspn(varname, "\'") == 0) {
   /* drop the quotes, but pass string as a cset */
   StrLoc(elsrc->lslots[i]) =
         StrLoc(elsrc->lslots[i]) + 1;
   StrLen(elsrc->lslots[i]) =
         StrLen(elsrc->lslots[i]) - 2;
   cnv:cset(elsrc->lslots[i], elsrc->lslots[i]);
   }
else if (strcspn(varname, "&") == 0) {
   if ( getkeyword(varname, &parm) == Failed) {
      VariableLookupFailed(varname);
      }
   cnv:cset(parm, elsrc->lslots[i]);
   if(complement) {
      Protect(cp = alccset(), fatalerr(0,NULL));
      cpx = (struct b_cset *)BlkD(elsrc->lslots[i],
            Cset);
      for (b = 0; b < CsetSize; b++)
         cp->bits[b] = ~cpx->bits[b];
      elsrc->lslots[i].vword.bptr =
            (union block *)cp;
      }
   }
else if (strcspn(varname, "1234567890") == 0) {
   cnv:integer(elsrc->lslots[i], elsrc->lslots[i]);
   }
else {
   if ( getvar(varname, &parm) == Failed) {
      VariableLookupFailed(varname);
      }
   pvar = VarLoc(parm);
```

```
                elsrc->lslots[i] = parm;
                if(complement) {
                   Protect(cp = alccset(), fatalerr(0,NULL));
                   cpx = (struct b_cset *)StrLoc(elsrc->lslots[i]);
                   for (b = 0; b < CsetSize; b++)
                      cp->bits[b] = ~cpx->bits[b];
                   elsrc->lslots[i].vword.bptr = (union block *)cp;
                   }
                }
             parm = elsrc->lslots[i];
             }
          }
       else if (is:list(elsrc->lslots[i])) {
          /* recursively visit sublists, do same stuff */
          lptemp = (struct b_list *)BlkD(elsrc->lslots[i], List);
          lpdest = ResolveList(lptemp);
          }
       else {
          /* cset, integer constant, ... */
          parm = elsrc->lslots[i];
          }
       eldest->lslots[i] = parm;
       }
    }
 return lpdest;
 }
```

The above function identifies the procedure to be called, along with its parameters. In the event a parameter is a list then the function is recursively called. If it is a string then it first removes the outside quotation marks. Then it checks for a tilde ~ as a complement operator. If there is one, then a complement is tagged and the tilde is removed from the string. Then it identifies the parameter type as either a string, cset, reserved word or numeric value, by identifying the first character in the string. A " signifies a string, a ' for a cset, a & for a reserved word, a digit for a numeric value and all other values being checked for a user defined variable. In the event of a user defined variable or reserved word, `ResolveList()` replaces it with its current value.

To get the values of the reserved words, the following `getkeyword()` function was added to the rmisc.r file in the runtime environment of the Unicon implementation. It connects the keyword with its appropriate cset value.

```
int getkeyword(char *s, dptr vp)
{
   if (*s++ == '&') {
      switch(*s++) {
      case 'a':
         if (!strcmp(s, "scii")) {
            Kascii(vp);
            return Succeeded; }
         break;
      case 'c':
         if (!strcmp(s, "set")) {
            Kcset(vp);
            return Succeeded; }
         break;
      case 'd':
         if (!strcmp(s, "igits")) {
            Kdigits(vp);
            return Succeeded; }
         break;
      case 'l':
         if (!strcmp(s, "etters")) {
             Kletters(vp);
            return Succeeded; }
         else if (!strcmp(s, "case")) {
            Klcase(vp);
            return Succeeded; }
         break;
      case 'u':
         if (!strcmp(s, "case")) {
            Kucase(vp);
            return Succeeded; }
         break;
         }
      }
   return Failed;
}
```

4.1.3 Procedure and Method calls in Unevaluated Expressions

During runtime a pattern is implemented in two phases, the first being its construction where the arguments and variables are assigned, followed by pattern execution. The values for the arguments and variables are set during its construction. This static nature of pattern components is inefficient and a possible maintenance nightmare [2]. The unevaluated expression facility allows the use of the current values of variables and arguments during execution [2].

In the pattern `pat := Len(n)` the value of n is assigned during pattern construction. Every time the `pat` is used, `n` will have the value it was assigned when it was dereferenced and passed on to `Len()`. Had `n` not been assigned a value then it would be `&null` which would produce an error in the `Len()` pattern function. On the other hand, had the pattern been defined with `n` being an unevaluated expression, such as `pat := Len(‘n‘)`, then the current value of `n` would be used each time `pat` was used. If the value of `n` changed, then the next time `pat` is used, it would use the new value for `n`.

The same is true with making procedure and method calls within unevaluated expressions. To get the full advantage of making such calls, pattern functions were revised to support unevaluated procedure and function calls as arguments. Functions that require an integer argument call the `ConvertPatternArgumentInt` macro and those that require a cset argument call the `ConvertPatternArgumentCset` macro. These macros required the defined values `PC_Pred_Func` and `PC_Pred_MF` be added to the `switch(pe->pcode)` code block as follows:

```
#begdef ConvertPatternArgumentCset(arg, bp, ptype)
   type_case arg of {
      pattern: {
      struct b_pattern *pat = (struct b_pattern *)BlkLoc(arg);
      struct b_pelem *pe = (struct b_pelem *)pat->pe;
      switch(pe->pcode) {
         case PC_Rpat: {
            bp = pattern_make(0, &EOP,ptype ## _VP,1, pe->parameter);
            break;
            }
         case PC_Pred_Func:
         case PC_String_VF: {
            bp = pattern_make(0, &EOP,ptype ## _VF,1, pe->parameter);
            break;
            }
         case PC_Pred_MF:
         case PC_String_MF: {
            bp = pattern_make(0, &EOP,ptype ## _MF,1, pe->parameter);
            break;
            }
         default: {
            runerr(127);
            }
         }
      }
   default: {
      if (!cnv_cset(&arg, &arg))
         runerr(104, arg);
      bp = pattern_make(0, &EOP, ptype ## _CS,1, arg);
      }
   }
#enddef
```

By adding these two lines, the macro is able to recognize unevaluated procedure and method calls and use the appropriate arguments when calling the `pattern_make()` function.

During pattern evaluation, pattern functions that require an integer or cset argument, call either the `GetResultFromFuncCall()` function or the `GetResultFromMethodCall()` function. Again, Gaikaiwari's version of these functions did not support nested lists.

```
#begdef GetResultFromFuncCall()
   int nargs;
   struct descrip procargs[10];
   struct descrip proc;
   tended struct b_lelem *ep;
   int i;
   tended struct descrip cresult;
   dptr call_result;
   tended struct b_list *lp =
         (struct b_list *)BlkLoc(Node->parameter);
   nargs = lp->size -1;
   ep = (struct b_lelem *)lp->listhead;
   proc = ep->lslots[0];
   for (; BlkType(ep) == T_Lelem;
         ep = (struct b_lelem *)ep->listnext) {
      for (i = 1; i < ep->nused; i++) {
         procargs[i - 1] = ep->lslots[i];
         }
      }
   call_result = calliconproc(proc,procargs,nargs);
   if (call_result == 0  || call_result->vword.descptr == 0 ){
      goto Node_Fail;
      }
   cresult = *call_result;
#enddef
```

The above code can handle a function call with only integers, strings and csets as arguments. Nested function calls were not supported.

The `processFuncCallList()` function was revised to handle the nested lists generated by the new `emit_code_for_uneval()` procedure. `GetResultFromFuncCall()` was revised to get the resulting value of the recursive function `processFuncCallList()`.

```
#begdef GetResultFromFuncCall()
   tended struct descrip cresult;
   dptr call_result;
   tended struct b_list *lp = (struct b_list *)BlkLoc(Node->parameter);
   call_result = processFuncCallList(lp);
   if (call_result == 0  || call_result->vword.descptr == 0 ){
      goto Node_Fail;
```

```
      }
   cresult = *call_result;
   if(is:integer(cresult)) {
      internalIntStorage = cresult.vword.bptr;
      }
#enddef
```

By making the `processFuncCallList()` function recursive, method and procedure calls can be used as arguments in the unevaluated function call.

```
dptr processFuncCallList(struct b_list *lp){
   int nargs;
   tended struct b_lelem *ep;
   struct descrip proc;
   int i;
   dptr rv;
   struct descrip procargs[10];
   tended struct descrip cresult;
   dptr call_result;
   tended char *temp;
   nargs = lp->size -1;
   ep = (struct b_lelem *)lp->listhead;
   proc = ep->lslots[0];
      for (i = 1; i < ep->nused; i++) {
         if(is:list(ep->lslots[i])) {
            /*check for method call else function call*/
            if(isMethod(ep) >= 0)
               rv = processMethodCallList(BlkD(ep->lslots[i], List));
         else rv = processFuncCallList(BlkD(ep->lslots[i], List));
         rv = processFuncCallList(BlkD(ep->lslots[i],List));
         if (rv == 0) return 0;
         procargs[i - 1] = *rv;
         }
      else if (is:variable(ep->lslots[i])) {
         procargs[i - 1] = *(ep->lslots[i].vword.descptr);
         }
      else {
         procargs[i - 1] = ep->lslots[i];
         }
      }
   call_result = calliconproc(proc,procargs,nargs);
   return call_result;
   }
```

If a pattern is defined as `pat := Len(‘foo(bar(arg))‘)`, then a list for the unevaluated expression `foo(bar(arg))` would be `(["foo", ["bar", "arg"]])`. The value `"foo"` would be recognized as the name of the procedure being called. The first argument of `foo` would be the list `["bar", "arg"]`. After discovering that `"bar"` is also a procedure, the `processFuncCallList()` would be recursively called with `"arg"` as its argument.

The `GetResultFromMethodCall()` macro was handled in a similar manner. It calls the recursive function `processMethodCallList()` to handle the analysis of the list.

```
#begdef GetResultFromMethodCall()
   int nargs;
   tended struct descrip cresult;
   dptr call_result;
   tended struct b_list *lp =
         (struct b_list *)BlkLoc(Node->parameter);
   nargs = lp->size -1;
   if (is:list(Node->parameter)) {
      call_result = processMethodCallList(lp);
      if ( call_result == 0 ) goto Node_Fail;
      cresult = *call_result;
      if(is:integer(cresult)) {
         internalIntStorage = cresult.vword.bptr;
         }
      }
   else {
      ErrorDuringPatternMatch("error in method call parameters");
      }
#enddef
```

There are some differnces in how the data for method calls of objects are stored in the lists generated by `emit_code_for_uneval()`. In the procedure call list, the first cell is the name of the procedure. In method call lists, the first cell is the name of the object and the second cell is the method name. A pattern definded as `pat`

`:= Span(‘foo.bar(arg)‘)`, would result in the list `(["foo", "bar", "arg"]`. The `GetResultFromMethodCall()` macro calls the recursive function `processMethodCallList()`.

```
dptr processMethodCallList(struct b_list *lp){
   int nargs;
   tended struct b_lelem *ep;
   tended struct b_record *rp;
   union block *bptr;
   struct descrip methodptr;
   struct descrip proc;
   struct descrip self;
   struct descrip argmnt;
   int i;
   dptr rv;
   struct descrip procargs[10];
   dptr call_result;
   struct descrip var;
   tended char *varname;
   int i, nfields, fnum;
   int found__m = 0;
   nargs = lp->size - 1;
   ep = (struct b_lelem *)lp->listhead;
   cnv:C_string(ep->lslots[0], varname);
   if (getvar(varname, &var) == Failed) {
      VariableLookupFailed(varname);
      }
   procargs[0] = var;
   deref(&var,&var);
   if (!is:record(var))
      RunErr(107, &var);
   self = var;
   found__m = isMethod(ep);
   methodptr = ep->lslots[1];
   rp = (struct b_record *)BlkLoc(var);
   if (found__m == -1) {
      fprintf(stderr, "Trying to lookup method from a non object");
      fatalerr(0, NULL);
      }
   var = rp->fields[found__m];
   rp = (struct b_record *)BlkLoc(var);
   bptr = rp->recdesc;
   nfields = bptr->Proc.nfields;
```

```
for (i=0; i<nfields; i++) {
   if ((StrLen(methodptr) ==
         StrLen(bptr->Proc.lnames[i])) &&
         !strncmp(StrLoc(methodptr),
               StrLoc(bptr->Proc.lnames[i]),
               StrLen(methodptr)))
      break;
      }
   if (i<nfields)
      fnum = i;
   else {
      VariableLookupFailed(varname);
      }
   proc = rp->fields[fnum];
   for (i = 1; i < ep->nused; i++) {
      if(is:list(ep->lslots[i])) {
         /* check for method call else function call*/
         if(isMethod(ep) >= 0) {
            rv = processMethodCallList(BlkD(ep->lslots[i], List));
            }
         else {
            rv = processFuncCallList(BlkD(ep->lslots[i], List));
            }
         if (rv == 0) return 0;
         procargs[i - 1] = *rv;
         }
      else if (is:variable(ep->lslots[i])) {
         procargs[i - 1] = *(ep->lslots[i].vword.descptr);
         }
      else {
         if (i == 1) procargs[i - 1] = self;
         else if(strpbrk(ep->lslots[i].vword.sptr,
               "abcdefghijklmnopqrstuvwxyz
               ABCDEFGHIJKLMNOPQRSTUVWXYZ_") == NULL) {
            cnv:integer(ep->lslots[i], argmnt);
            procargs[i - 1] = argmnt;
            }
         else procargs[i - 1] = ep->lslots[i];
      }
   }
call_result = (dptr)calliconproc(proc,procargs,nargs);
return call_result;
}
```

It is possible to have nested procedures and methods in an unevaluated ex-

pression. There are differences in how to interpret the lists. With a procedure call the first cell is for the procedure name, while with the method call the first two cells are for the object name and the method name respectively. Take the list `["foo", "bar", "arg"]`, it could also be a procedure call with two arguments. The `isMethod()` function checks to see if there is a method matching the parameters given. In this case it checks to see if there is an object called `foo` and a method in that object called `bar`. If it succeeds then it will return its field number, otherwise it will return a -1. This helper function is used in `processFuncCallList()` and `processMethodCallList()` to determine if a list is for a method call.

```
int isMethod(struct b_lelem *ep){
   tended struct b_record *rp;
   union block *bptr;
   struct descrip var;
   tended char *varname;
   struct descrip procargs[10];
   struct descrip methodptr;
   int i, found__m = -1;
   cnv:C_string(ep->lslots[0],varname);
   if (getvar(varname,&var) == Failed) {
      VariableLookupFailed(varname);
      }
   procargs[0] = var;
   deref(&var,&var);
   if (!is:record(var))
      RunErr(107, &var);
   methodptr = ep->lslots[1];
   rp = (struct b_record *)BlkLoc(var);
   bptr = rp->recdesc;
   for( i = 0; i < bptr->Proc.nfields;i++) {
      if (!strcmp(StrLoc(bptr->Proc.lnames[i]), "__m")) {
         found__m = i;
         break;}
      }/* for ... nfields */
   return found__m;
   }
```

4.1.4 String and Pattern Concatenation Operators

Both string concatenation and pattern concatenation result in the joining of any two values that can be converted into strings, but in the case of pattern concatenation the values may not be defined until runtime. Pattern concatenation accepts values that can be converted into patterns, such as strings, csets and patterns. As patterns and strings are different data types, string concatenation and pattern concatenation are handled as separate operations. Therefore concatenation operator must identify whether a pattern is being used. Then it has to send the appropriate data to the appropriate concatenation operation.

The following code identifies whether a pattern is being used with the concatenation operator. If it is, then a pattern object will be returned; otherwise a string is returned.

```
declare {
   int use_trap = 0;
   }

if is:pattern(x) then {
   inline {
      use_trap = 1;
      }
   abstract {
      return pattern;
      }
   }
else if is:pattern(y) then {
   inline {
      use_trap = 1;
      }
   abstract {
      return pattern;
      }
   }
else {
```

```
      if !cnv:string(x) then runerr(103, x)
      if !cnv:string(y) then runerr(103, y)
      abstract {
         return string;
         }
      }
```

If a pattern is identified in the above code, then the body of the function will know to return a pattern, and execute the following code, where it will setup the data required for pattern concatenation and call the pattern concatenation operation.

```
body {
   if (use_trap == 1) {
      union block *bp;
      /* convert strings to pattern blocks */
      struct b_pattern *lp;
      struct b_pattern *rp;
      struct b_pelem *pe;
      type_case x of {
         string:
            cnv_str_pattern(&x,&x);
         cset:
            cnv_cset_pattern(&x,&x);
         pattern: {
            }
         default:{
            runerr(127);
            }
         }
      type_case y of {
         string:
            cnv_str_pattern(&y,&y);
         cset:
            cnv_cset_pattern(&y,&y);
         pattern: {
            }
         default:{
            runerr(127);
            }
         }
      lp = (struct b_pattern *)BlkLoc(x);
      rp = (struct b_pattern *)BlkLoc(y);
```

```
      /* perform concatenation in patterns */
      pe = Concat(Copy((struct b_pelem *)lp->pe),
            Copy((struct b_pelem *)rp->pe), rp->stck_size);
      bp = pattern_make_pelem(lp->stck_size + rp->stck_size,pe);
      return pattern(bp);
      }
      else {...
```

4.2 Index Related Functions

The pattern cursor location representation in the pattern functions have been revised to match the Unicon index location of strings as shown in section 3.3.2 of this paper. This was achieved by changing the Pos(n) function code to appear as follows:

```
function {1} Pos(position)
   abstract {
      return pattern;
   }
   body {
      union block *bp;
      /*
       * check if position is negative
       */
      if(position.vword.integr < 1) {
         /* change position to a positive value and use RPos */
         position.vword.integr = -position.vword.integr;
         ConvertPatternArgumentInt(position,bp,PC_Pos);
      } else {
         ConvertPatternArgumentInt(position,bp,PC_Pos);
      }
      return pattern(bp);
      }
end
```

The position that is passed to the function is a descriptor in the Unicon virtual machine that is defined to be an integer. The details of the descriptor can be found in Implementation of Icon and Unicon [6]. The `if` statement in this function checks

to see if the value in the descriptor is negative. If it is negative then its value is changed to its complement. The `ConvertPatternArgumentInt()` function is called while passing position, a block pattern and the `PC_RPos` call as arguments. If it is positive then position is not changed and the `PC_Pos` argument is used instead while calling the `ConvertPatternArgumentInt()` function. A nearly identical change was made to the Tab function.

The `Rpos(n)` and `Rtab(n)` functions are now redundant, but are still available for the user who may be more comfortable with SNOBOL. When using `Rpos(n)` and `Rtab(n)`, the user has to be aware that they are based on the SNOBOL4 pattern matching cursor location system.

5 Evaluation

To evaluate the how successful this implementation of SNOBOL4 style patterns with Unicon's string scanning environment, a set of benchmark problems is used. The benchmarks are used to compare the solutions using Unicon string scanning environment, Gaikaiwari's pattern statements, current pattern statements, an anchored pattern match from the string scanning environment, and pattern definitions with unevaluated expressions with string scanning functions. The solutions for each problem will be evaluated for clarity, simplicity and functionality. Clarity deals with the readability of the lines of code for each particular problem. Can the user or programmer read the code without any ambiguity as to what it is attempting to achieve? Simplicity will be measured by the number of lines, words and characters required to achieve each benchmark problem. Functionality of the example code will have to be consistent. Each example will have to achieve the same result for each benchmark problem. The benchmark problems are listed below:

- Decomposing phone numbers
- Detecting words with double letters
- Strings of the form $A^nB^nC^n$
- Number of times a word is used in a file

5.1 Decomposing phone numbers

For the purpose of decomposing phone numbers in North America, each piece

of example code must identify the following:

1. area code is three digits with or without parentheses.
2. optional separator
3. trunk is three digits
4. optional separator
5. remaining four digits of the number

The following example phone numbers should be recognizable:

1. 800-555-1212
2. 800 555 1212
3. 800.555.1212
4. (800) 555-1212
5. 1-800-555-1212
6. 1-(800) 555-1212

An input fragment of "`Home: (800) 555-1212`" should result in identifying the area code as `800`, the trunk as `555`, the remainder of the number as being `1212`.

In the string scanning environment it can be achieved with the following code:

```
procedure digits(N)
   if N = 0 then return ""
   else return tab(any(&digits)) || digits(N - 1)
end

procedure main()
   line := "Uncle Sam: (800)555-1212 or uncle.sam@us.gov"
   sep := ""
   a := 0
   line ? {
      tab(upto(&digits))
      areaCode := digits(3)
      sep := tab(any(' -.'))
```

```
      tab(upto(&digits))
      trunk := digits(3)
      if sep === "" then sep := tab(any(' -.'))
      else tab(any(sep))
      number := digits(4)
   }
   write("(" || areaCode || ")" || trunk || sep || number)
end
```

In this example, the phone number is found inductively. It starts by skipping all the text up to the point where the digits begin, then it starts extracting each section of digits. The first being for the area code; the second for the trunk; then identifying the character for the separation between the trunk and the remainder of the number; and finally the remainder of the number. This requires more coding than the other options described below, and will give erroneous answers if the length of the area code or trunk is not three characters in length. A couple more lines of code and the length of each part of the phone number can be defined.

In the pattern matching environment as implemented by Gaikaiwari, the problem can be resolved with the following code:

```
procedure main()
   line := "Uncle Sam: (800)555-1212 or uncle.sam@us.gov"
   threedigit := PAny(&digits) && PAny(&digits) && PAny(&digits)
   fourdigit := threedigit && PAny(&digits)
   area := ("(" && threedigit => areaCode && ")" )
   pattern := (threedigit => areaCode && PAny(' -.') => sep
         && threedigit => trunk && `sep`
         && fourdigit => number)
         .| (area && threedigit => trunk
         && PAny(' -.') => sep && fourdigit => number)
   line ?? pattern
   write("(" || areaCode || ")" || trunk || sep || number)
end
```

The current implementation of patterns the code is as follows:

```
procedure main()
```

```
   line := "Uncle Sam: (800)555-1212 or uncle.sam@us.gov"
   threedigit := Any(&digits) || Any(&digits) || Any(&digits)
   fourdigit := threedigit || Any(&digits)
   area :=  "(" || threedigit => areaCode || ")"
   pattern := (threedigit => areaCode || Any(' -.') => sep ||
         threedigit => trunk || `sep` ||
         fourdigit => number)
         .| (area || threedigit => trunk ||
         Any(' -.') => sep || fourdigit => number)
   line ?? pattern
   write("(" || areaCode || ")" || trunk || sep || number)
end
```

In both of the pattern matching examples, the pattern is defined near the beginning of the code and allows the phone number to be found deductively. Although there is greater use of the parentheses for the assignment notation, it is lexically more consistent with Unicon.

When using patterns within the string scanning environment it can be achieved with the following code:

```
procedure main()
   out := &output
   line := "Uncle Sam: (800)555-1212 or uncle.sam@us.gov"
   threedigit := Any(&digits) || Any(&digits) || Any(&digits)
   fourdigit := threedigit || Any(&digits)
   area := ("(" || threedigit => areaCode || ")" )
   pattern := (Arb() || threedigit => areaCode ||
         Any(' -.') => sep || threedigit => trunk ||
         Any(`sep`) || fourdigit => number)
         .| (Arb() || (area || threedigit => trunk ||
         Any(' -.') => sep || fourdigit => number)
   line ? =pattern
   write("(" || areaCode || ")" || trunk || sep || number)
end
```

In this example the user is able to separate the phone number pattern out of the string scanning environment by defining it as a pattern and then using the tabmat function to call an anchored pattern match when it is needed. In this option the

phone number is also found deductively, allowing the user more flexibility in using the pattern and improving the readability of the string scanning function.

The following solution uses an unevaluated expression of string scanning functions. It also illustrates the added efficiency of defining a pattern and using it repeatedly; as is shown with the `digit` pattern definition, when it is used three times in `threedigit` and once in `fourdigit`.

```
procedure main()
   line := "Uncle Sam: (800)555-1212 or uncle.sam@us.gov"
   digit := ‘tab(any(&digits))‘
   threedigit := digit || digit || digit
   fourdigit := threedigit || digit
   area := ( "(" || threedigit => areaCode || ")" )
   pattern := (threedigit => areaCode || Any(’ -.’) => sep ||
         threedigit => trunk || ‘sep‘ || fourdigit => number)
         .| (area || threedigit => trunk || Any(’ -.’) => sep ||
         fourdigit => number)
   line ?? pattern
   write("(" || areaCode || ")" || trunk || sep || number)
end
```

The following table shows the number of lines, words and characters that were used in these solutions.

Table 3: Decomposing phone numbers

	Lines	Words	Chars w/ ws	Chars wo/ ws
String scanning	21	74	510	364
Gaikaiwari’s Patterns	9	81	532	409
Patterns	9	80	504	401
Pattern in string scanning	10	91	548	432
Unevaluated expression	11	89	515	401

In this first benchmark the string scanning solution used more lines of code, but used fewer words and characters to accomplish the same result as the other solutions.

5.2 Detecting words with double letters

In order to identify each word from a source file that has double letters, the procedure must be able to identify words like

- tooth
- small
- tomorrow

The string scanning example was used in [2] and is shown below.

```
procedure main()
   in := open("mtent12.txt", "r") | stop("open failed")
   out := open("mtentpatternOut.txt", "w")
   while line := read(in) do {
      line ? {
         while(tab(upto(&letters))) do {
            word := tab(many(&letters))
            word ? {
               while c := move(1) do {
                  if move(1) == c then {
                     write(out, word)
                     break
                  }
               }
            }
         }
      }
   }
end
```

It uses a nested call to the string scanning environment, the first to identify a word and the second to test to see if the word contains a double letter. If it is successful then it outputs the word.

The next example provided in [2] shows how the problem can be solved using Gaikaiwari's pattern data type and matching system.

```
procedure main()
```

```
   in := open("mtent12.txt", "r") | stop("open failed")
   out := open("mtentpatternOut.txt, "w")
   double := PArbno(&letters) && PAny(&letters) $$ x && `x` &&
      (PSpan(&letters) .| "")
   every write(out, (line := !in) ?? double)
end
```

The sample code below is the same as Gaikaiwari's, but it has been revised for this integration of Unicon patterns.

```
procedure main()
   in := open("mtent12.txt", "r") | stop("open failed")
   out := open("mtentpatternOut.txt", "w")
   double := Arbno(&letters) || Any(&letters) => x || `x` ||
      (Span(&letters) .| "")
   every write(out, (line := !in) ?? double)
end
```

In both cases the code is much shorter than the string scanning example. This example the order of operations for the concatenation operator || require the assignments and alternations to be defined within parentheses.

In the following example the string scanning environment is initialized to identify each word, then each word is tested for any double letters, using a pattern sequence which is simpler than the combination of a while loop with a comparison as shown in the first example for this problem.

```
procedure main()
   in := open("mtent12.txt", "r") | stop("open failed")
   out := open("mtentpatternOut.txt", "w")
   double := Arb() || Any(&letters) => x || `x`
   while line := read(in) do {
      line ? {
         while(tab(upto(&letters))) do {
            word := tab(many(&letters))
            if word ? =double then write(out, word)
         }
      }
   }
end
```

The following is similar to the pattern example in that it replaces the `Any()` pattern function with the `tab(any())` string scanning functions as an unevaluated expression.

```
procedure main()
   in := open("mtent12.txt", "r") | stop("open failed")
   out := open("mtentpatternOut.txt", "w")
   double := Arbno(&letters) || ‘tab(any(&letters))‘ => x ||
         ‘x‘ || (Span(&letters) .| "")
   every write(out, (line := !in) ?? double)
end
```

Table 4: Detecting words with double letters

	Lines	Words	Chars w/ ws	Chars wo/ ws
String scanning	18	53	470	250
Gaikaiwari's Patterns	6	34	245	205
Unicon Patterns	6	34	255	203
Pattern and string scanning	13	48	371	254
Unevaluated expression	6	34	256	210

The pattern examples had the fewest number of lines, words and characters used. When a pattern match was performed using the tabmat operator in the string scanning environment, it resulted in fewer lines, words and characters as compared with using the string scanning environment alone. It also condensed the string analysis of each individual word to a single line testing for the presence of a double letter. For novice users of Unicon, this last example would be the easiest to understand.

5.3 Strings of the form $A^nB^nC^n$

The third benchmark problem is the common language that cannot be parsed by a CFG grammar as was shown in [2]. The programs write `accepted` when the string of characters containing "a"s, "b"s and "c"s is in the form of $a^nb^nc^n$ and write `rejected` when any other string is provided.

The string scanning example shown below is from [2] and [**?**]. It is able to resolve the problem with 12 lines of code.

```
procedure ABC(s)
   suspend =s | (="a" || ABC("b" || s) || ="c")
end

procedure main()
   while write(line := read()) do
      if line ? {
         ABC("") & pos(0)
      }
      then write("accepted")
      else write("rejected")
end
```

The program defines a procedure ABC which is called in a string scanning environment to see if it contains the form $a^n b^n c^n$ and no other characters following. Defining a procedure to test for a pattern is a common way to handle patterns when the pattern data type is not available.

Below is the example code Gaikaiwari provided to solve this problem using his implementation of patterns.

```
procedure test(a, b, c)
   return ((a - 1) = (b - a)) & ((a - 1) = (c - b))
end

procedure main()
   pattern := PPos(1) && PSpan("a") && .$ a && PSpan("b") && .$ b &&
         PSpan("c") && .$ c && PRpos(0) && ‘test(a, b, c)‘
   while write(line := read()) do {
      if(line ?? pattern) then write("accepted")
      else write("rejected")
   }
end
```

A test procedure was used to determine if the number of A's, B's and C's are equal. This test was called at the end of the pattern definition causing the pattern succeed

when they were all equal and fail when they were not.

The following is another example how this problem can be solved using patterns.

```
procedure main()
   pattern := Pos(1) || Span("a") || .> a || Span("b") || .> b ||
         Span("c") || .> c
   while write(line := read()) do {
      if(line ?? pattern) then {
         if (a - 1 == b - a & a - 1 == c - b) then write("accepted")
         else write("rejected")
      }
   }
end
```

This example defines a pattern that defines the cursor location for the end of a, b and c respectively. It then tests those values to see if they are a mathematical match for the language. The grammar requires the index location assignments to be made as a concatenation to an element in the pattern. Again the `&&` operator for concatenation can be hard to distinguish while reading a pattern definition.

The following example uses the pattern matching environment within the string scanning environment and is able to resolve the problem with 9 lines of code.

```
procedure main()
   pattern := Pos(1) || Span("a") || .> a || Span("b") || .> b ||
         Span("c") || .> c
   while write(line := read()) do
      line ? {
         =pattern
         if (a - 1 == b - a & a - 1 == c - b) then write(" accepted")
         else write(" rejected")
      }
end
```

This example also defines a pattern that defines the cursor location of for the end of a, b and c respectively. It then tests those values to see if they mathematically match for the language.

```
procedure main()
   pattern := Pos(1) || ‘tab(many(’a’))‘ || .> a || ‘tab(many(’b’))‘ || .> b ||
         ‘tab(many(’c’))‘ || .> c
   while write(line := read()) do {
      if(line ?? pattern) then {
         if (a - 1 == b - a & a - 1 == c - b) then write(" accepted")
         else write(" rejected")
      }
   }
end
```

Table 5: Strings of the form $A^nB^nC^n$

	Lines	Words	Chars w/ ws	Chars wo/ ws
String scanning	12	35	226	162
Gaikaiwari's Patterns	12	64	349	253
Unicon Patterns	9	56	290	195
Pattern and string scanning	9	53	275	185
Unevaluated expression	9	56	310	216

The string scanning example may have used more lines than the pattern examples, but required fewer words or characters than all the other solutions. Of the pattern examples, the `||` concatenation operator is more consistent lexically than the `&&` concatenation operator for the Unicon language.

5.4 Number of times a word is used

To count the number of times a word is used in a block of text you need the algorithm to count every time the word is used, but not when the word is a part of some other word. When counting the number of times the word `the` is used, counting `they, them, there` and so forth would not get the desired result. So the algorithm must be refined enough to recognize the difference.

The following sample code solves this problem using a string scanning environment by examining blocks of letters and testing to see if the block of text matches the testword.

```
procedure main(args)
   in := open("test.txt", "r") | stop("open failed")
   test := read()
   count := 0
   while line := read(in) do {
      line ? {
         while(tab(upto(&letters))) do {
            word := tab(many(&letters))
            if word === test then count +:= 1
         }
      }
   }
   write(count)
end
```

The following pattern matching examples show how they would appear in Gaikaiwari's and the current Unicon implementation respectively.

```
procedure main(args)
   in := open("test.txt", "r") | stop("open failed")
   word := read()
   pattern := PNotAny(&letters) && word && PNotAny(&letters)
   count := 0
   every w := !in ?? pattern do count +:= 1
   write(count)
end
```

```
procedure main(args)
   in := open("test.txt", "r") | stop("open failed")
   word := read()
   pattern := Any(~&letters) || word || Any(~&letters)
   count := 0
   every w := !in ?? pattern do count +:= 1
   write(count)
end
```

Both are the same solution to the problem. There are slight gains in typing in the second. This is a result of not having to use the `PNotAny()` function as the same result can be achieved with the `Any()` function while using the `~` before the cset.

Finally, the following code resolves the problem using pattern matching from the string scanning environment.

```
procedure main(args)
   in := open("test.txt", "r") | stop("open failed")
   word := read()
   pattern := Arb() || Any(~&letters) || word || Any(~&letters)
   count := 0
   every line := !in do {
      line ? {
         while =pattern do count +:= 1
      }
   }
   write(count)
end
```

In this example the pattern begins with an `Arb()` element. Since the pattern match is carried out in the anchored mode, matching some arbitrary junk before the word is matched is necessary otherwise the matching operation will not make the match unless the word starts at the current cursor location. This can cause the pattern match to get hung up in an endless loop.

In the following example, the `Any()` pattern functions were replaced with `tab(any())` string scanning functions. This requires more characters but essentially works the same as the pattern examples. It is shown as a demonstration of how the unevaluated expression can be used to make string scanning function calls.

```
procedure main(args)
   in := open("test.txt", "r") | stop("open failed")
   word := read()
   pattern := ‘tab(any(~&letters))‘ || "test" || ‘tab(any(~&letters))‘
   count := 0
   every w := !in ?? pattern do count +:= 1
   write(count)
end
```

Again the pattern examples require the fewest lines and characters, demonstrating that they are more efficient for the user to generate a solution. By allowing the use of `Any()` function in place of the `PNotAny()` function, reduces the number

Table 6: Number of times a word occurs in a file

	Lines	Words	Chars w/ ws	Chars wo/ ws
String scanning	14	43	309	208
Gaikaiwari's Patterns	8	34	223	179
Unicon Patterns	8	34	217	173
Pattern and string scanning	12	43	271	198
Unevaluated expression	8	34	240	189

of pattern functions required for the same functionality.

6 Conclusions

In the process of integrating pattern matching with string scanning, the pattern function set has been reduced, string scanning is able to perform pattern matching with the tabmat operator `=`, pattern matching can perform string scanning operations, and the unevaluated expressions can handle complex procedure and method calls.

6.1 Reduced Pattern Function Set

The integration of Gaikaiwari's pattern matching environment and the Unicon string scanning environment resulted in allowing users to utilize the pattern matching facilities from a string scanning environment. Pattern matching functions using the complement operator `~` made some SNOBOL4 pattern functions redundant. `NOTANY()`, `RTAB()` and `RPOS` are such examples and their functionality are available with `Any()`, `Tab()` and `Pos()` respectively.

In the first benchmark problem, there were several examples of how to extract a phone number from a body of text. A pattern function allowing the user to identify the number of times a pattern is replicated would have been very useful in defining the number of digits in each portion of the number. If a user was looking for a pattern that is repeated several times, the `Repl()` pattern function would make the code shorter and cleaner.

6.2 Pattern and String Scanning Integration

The tabmat operator `=` is used to make a pattern match within a string scanning operation. When combined with an evaluation, it allows the user to easily make comparisons to complex patterns. While performing a string scanning operation, the user wants to see if the next block of text matches a complex pattern that is defined the code would be `if =complexPatter then`. With such an expression the index location in the subject string would be advanced to the end of the pattern, to save the content of the pattern the code to use would be `if mat := =complexPattern then`.

The changes to unevaluated expressions allow the user to string scanning functions within a pattern.

string scanning with unevaluated expressions

6.3 Static Procedure, Method and Variable Calls

`pat := foo(arg)` `pat := foo(‘arg‘)`

6.4 Improvements to Unevaluated Exressions

`pat := Len(3) => arg || ‘foo(arg)‘` `pat := ‘foo(bar(arg))‘` `pat := ‘foo.bar(arg)‘`

not supported: `pat := ‘foo(arg[n:n])‘`

6.5 Benchmark Problems

In each of the benchmark problems it was shown that pattern matching in Unicon is a viable option in string analysis. Although, the modified pattern type developed for this thesis did not produce significantly improved code lengths as compared to

Gaikaiwari's patterns, it still produced a significant improvement over the string scanning environment of Unicon in two of the four benchmark problems.

Pattern matching does not always result in shorter and therefore simpler code, pattern matching is a useful improvement to the Unicon language. It allows those users that prefer the functionality of SNOBOL4 patterns to use a similar system of string analysis.

The lexical changes to Gaikaiwari's pattern matching are an improvement, as it is more consistent with the Unicon language.

7 Future Work

Although the pattern matching and string scanning have been integrated, there are areas that need improvement.

Patterns can handle procedure calls, but the values are evaluated at pattern construction time. Therefore it is necessary to use the unevaluated expression to evaluate them at pattern evaluation time if any of the parameters are assigned in the pattern itself. As it stands now, the unevaluated expressions can not handle procedure calls.

Additional primitive pattern functions would also reduce the complexity of many pattern definitions. For example: In the first benchmark problem, to identify three digits in a row required `Any(&digits)` to be repeated three times. A primitive `Dupl(p, n)` which allows the user to identify a pattern that has been duplicated for a fixed number of times would be useful. Had the user been repeating a pattern several dozen times, it would prove to be a great improvement in functionality in defining the pattern. There could be other such primitives that add to the flexibility and improve the functionality of defining patterns.

In the area of readability, more studies need to be done on the selection of lexemes for functions and operators in programming languages. The author searched for papers addressing readability in programming languages and found very few. What was discovered, is that there has been work done in Cartography on the use of symbols in thematic maps, and what fonts the most readable in a verity of languages. But, little on the choice of symbol sets to represent operations in computer languages.

Some questions that need to be studied include: How does the use of standard mathematical expressions help a user read and comprehend a computer language? How do lexically consistent symbol selection help users read and comprehend expressions in a computer language? What is the effect of having many lexemes? Are longer or shorter lexemes more effective at communicating the purpose of the functions and operators they represent?

Bibliography

[1] R. E. Griswold, I. P. Polonsky, and J. Poage, *The SNOBOL4 Programming Language*. Prentice-Hall, 1971.

[2] S. Gaikaiwari, "Adapting SNOBOL-style Patterns to the Unicon Language," Master's thesis, New Mexico State University, 2005.

[3] R. E. Griswold and M. T. Griswold, *The Icon Programming Language*. Prentice-Hall Englewood Cliffs, NJ, 3 ed., 1983.

[4] R. E. Griswold, *Pattern Matching in Icon*. University of Arizona, Department of Computer Science, 1980.

[5] C. Jeffery, S. Mohamed, R. Pereda, and R. Parlett, *Programming with Unicon*. 2004.

[6] C. Jeffery, *Implementation of Icon and Unicon*.

Appendix A: Pattern Facilities Language Reference

Pattern Variables

variable

a variable in a pattern definition that may not be changed during a pattern match operation.

‘variable‘

an unevaluated variable in a pattern definition that can be changed in a pattern match operation.

Pattern Operators

pattern1 || pattern2 **pattern concatenation**

pattern concatenation operator produces a new pattern containing the left operand followed the right operand.

---

pattern1 .| pattern2 **pattern alteration**

pattern alternation operator produces a pattern containing either the left operand or the right operand.

---

substring -> variable **conditional assignment**

assigns the substring on the left to the variable on the right if the pattern match is

successful.

---

result => variable **immediate assignment**

assigns the immediate result on the left to a variable on the right within a pattern.

.> variable **cursor position assignment**

assigns the cursor position of the string to a variable on the right within a pattern.

---

string ?? pattern **comparison operator**

compares the string on the left to see if there are any matches of the pattern on the right in the un-anchored mode.

=pattern **comparison operator**

compares the current string in the string scanning environment to see if there is a match of the pattern on the right in the anchored mode.

Pattern Built-In Functions

Any(s) **match any**

matches any single character contained in s appearing in the subject string.

Arb() **arbitrary pattern**

matches zero or more characters in the subject string.

Arbno(p) **repetitive arbitrary pattern**

matches repetitive sequences of p in the subject string.

Back() **pattern back**

signals a failure in the current portion of the pattern match and sends an instruction to go back and try a different alternative.

Bal() **balanced parentheses**

matches the shortest non-null string which parentheses are balanced in the subject string.

Break(s) **pattern break**

matches any characters in the subject string up to but not including any of the characters in s.

Breakx(s) **extended pattern break**

matches any characters up to any of the subject characters in s, and will search beyond the break position for a possible larger match.

Cancel() **pattern cancel**

causes an immediate failure of the entire pattern match.

Fence() **pattern fence**

signals a failure in the current portion of the pattern match if it is trying to backing up to try other alternatives.

Len(n) **match fixed-length string**

matches a string of a length of `n` characters in the subject string. It fails if n is greater than the number of characters remaining in the subject string.

Pos(n) **cursor position**

sets the cursor or index position of the subject string to the position `n` according the Unicon index system shown bellow:

```
-6  -5  -4  -3  -2  -1  0
| U | n | i | c | o | n |
1   2   3   4   5   6   7
```

Rem() **remainder pattern**

matches the remainder of the subject string.

Span(s) **pattern span**

matches one or more characters from the subject string that are contained in `s`. It must match at least one character.

Tab(n) **pattern tab**

matches any characters from the current cursor or index position up to the specified position of the subject string. `Tab()` uses the Unicon index system shown in `Pos()` and position `n` must be to the right of the current position.

Rpos(n) **reverse cursor position**

sets the cursor or index position of the subject string to the position `n` according the SNOBOL4 index system shown bellow:

```
6   5   4   3   2   1   0
| S | N | O | B | O | L |
1   2   3   4   5   6   7
```

Rtab(n) **pattern reverse tab**

matches any characters from the current cursor or index position up to the specified position of the subject string. `Rtab()` uses the SNOBOL4 index system shown in `Rpos()` and position `n` must be to the right of the current position.