Integrating pattern matching within string scanning

John H. Goettsche
Dept. of Computer Science
University of Idaho

January 21, 2015

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 at that time. The goal of this project is to make the pattern data type accessible to the Unicon string scanning and vice versa. To accomplish this, a Unicon string scanning 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 lexeme was revised to be more consistent with the Unicon language.

Contents

1 Introduction

In order to enhance programmers' productivity in analyzing strings, many different string scanning and pattern matching systems have been developed. Modern highlevel 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.

A successor to SNOBOL4 was Icon, 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. Many of the Unicon 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].

When Unicon was developed, its core elements came directly from Icon, including its string scanning facilities. [5] 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.

This paper briefly explores the background of SNOBOL4 patterns and Unicon string scanning functions. It then presents a proposed design for integrating SNOBOL4 patterns within Unicon string scanning environments, and a description of how that implementation can be accomplished.

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 as 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 a cursor advances through that string. Unlike most other languages, Icon considers the string as a 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 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.

3 Design Considerations

To integrate pattern matching with Unicon string scanning required consideration of how to implement patterns in the string scanning environment and how to implement string scanning in the pattern matching operation. How they are going to be utilized and under what conditions for each implementation. 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 statement in SNOBOL4 and Gaikaiwari's Unicon implementation are in the following statements:

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. Gaikaiwari's is easier to read and is a better match to Unicon syntactically.

To integrate pattern matching into the Unicon language, it is necessary to consider which mode is appropriate considering the environment it is in. 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 would be appropriate for basic pattern matching operations. This allows the 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 it checks then the entire pattern will fail 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 the pattern can be defined starting with `Pos(1)` which will have the desired effect.

3.1.1 Pattern matching within String Scanning

When executing pattern matching from within the Unicon string scanning environment, the `&pos` is established and string scanning environment uses an inductive method of executing a series of expressions to manipulate and analyze a subject string. The cursor location or index is adjusted depending on the success or failure of each expression. In order to maintain this systematic process in the string scanning environment, it was decided to execute the pattern matching operation in the anchored mode.

The Unicon string scanning evironment is initialized in the following expression:

```
subject ? expr
```

The `?` operator sets the cursor location to the first position in the subject string and the function or the block of functions that are called in the expression

are 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

According Gaikaiwari's thesis, his pattern matching environment allows simple single depth function calls to be used as unevaluated variables and functions. It does not allow for function calls from within function calls like `tab(upto(cset))`. [2] I propose expanding the unevaluated variables and functions to handle more complex functions calls that 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. Gaikaiwari's operators for concatenation and alternation are lexically different from Unicon concatenation and alternation and the assignment operators are lexically inconsistent.

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 concatenation was modified to recognize whether the expression contains a pattern.

The assignment operators was changed lexically to use the > symbol to represent an assignment within patterns. The Immediate Assignment, Conditional Assignment and Cursor assignment are =>, − > and .> respectively.

3.3 SNOBOL4 and Unicon pattern functions

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 analyzed by injecting the functions into the subject string. While the patterns are pre-defined and are used deductively to search a subject string for the pattern. In the pattern matching operation many of the functions operate independently of the cursor location and later determine a new cursor location as a part of determining the results of the pattern match. This difference led the author to conclude that the SNOBOL4 pattern function names should be retained as much as possible while altering them to fit Unicon naming conventions.

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	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 Logically Redundant Function

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 `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

`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 the number of spaces subtracted from 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

***** UNDER REVIEW *****

Fail is a already taken as a keyword in Unicon. Attempts to use this name 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. 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()` would be a more appropriate term for the `ABORT` function.

****

4 Implementation

To implement these changes 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 function calls as a parameter in string scanning function calls.
- Add pattern scan to identify its contents and determine the pattern matching mode.
- 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 non-anchored pattern matching operation was implemented by Sudarshan Gaikaiwari in his 2005 Master's thesis at New Mexico State University. A non-anchored pattern matching expression consists of a subject followed by the comparison operator `??` followed by a pattern and appears as follows:

```
subject ?? pattern
```

The anchored pattern matching operation was defined in the pattern resources as a part of the internal match function, 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 were changed so the mode would be passed in along with the current index. This allows the internal match to be called and initiated in the proper location of the subject string.

4.1.1 Anchored mode from string scanning

It was decided that integrating the pattern matching system into the Unicon string scanning environment would require the pattern matching be performed in the anchored mode, since the string scanning functions are dependent upon 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 the 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 pattern data types. In the event that a pattern was its argument then it initiates a pattern match in the anchored mode, otherwise it functions normally. This section of code identifies whether the argument is a pattern or a string and assigns its return value.

```
operator{*} = tabmat(x)
   declare {
      int use_trap = 0;
   }
/*
 * x must be a pattern or convertible into a string.
 */
   if is:pattern(x) then {
      inline {
         use_trap = 1;
      }
      abstract {
         return string
      }
   } else if !cnv:string(x) then {
      runerr(103, x)
   } else
      abstract {
         return string
      }
```

For the pattern match, the body of code had to assign the values for each of the variables required in the pattern match. 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 update the index or cursor locations, if it failed then it would revert to the previous index. The following code was added to the body of the tabmat operator:

```
/*
 * set cursor position, and subject to match
 */
```

```
oldpos = curpos = k_pos;
pattern_subject = StrLoc(k_subject);
subject_len = StrLen(k_subject);
pattern = (struct b_pattern *)BlkLoc(x);
phead = 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, curpos - 1, 1)){
   /*
    * Set new &pos.
    */
   k_pos = stop + 1;
   EVVal(k_pos, E_Spos);
   oldpos = curpos;
   curpos = 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)) {
      curpos += 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);
}
```

4.1.2 String Scanning Functions as Unevaluated Expressions

In Sudarshan Gaikaiwari's implementation of SNOBOL4 patterns in Unicon, functions can be called in a pattern using the unevaluated expression notation by placing the function call in back quotes.

The first phase of the analysis an unevaluated expression occurs during compile time. In this process The `emit_code_for_uneval` procedure in the tree.icn file in the Unicon implementation is called and generates a list to be interpreted during runtime.

```
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 with this procedure is similar to LISP in that the first element is the function to be called and the remaining elements are the parameters. Unfortuneately, Gaikaiwari's implementation could not handle functions calls as parameters in another function call as is common with some string scanning functions. For example `tab(upto('e'))`. In this example the tab function requires an integer value which is acquired by the upto function which scans a string until it reaches an 'e' character. His 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. It was decided that a recursive procedure generating the list for each parameter which is a call to a function. It was implemented with the following two procedures:

```
procedure writes_code_for_uneval(L)
   writes(yyout, "[")
   every temp := !L\ (*L -1) do {
      if type(temp) == "list" then writes_code_for_uneval(temp)
  else writes(yyout, "\"", temp,"\",")
      }
   if type(L[*L]) == "list" then writes_code_for_uneval(L[*L])
   else writes(yyout, "\"", L[*L], "\"")
   writes(yyout, "]")
   return
end

procedure emit_code_for_uneval(funcname)
   L := []
   tab(many("‘"))
   temp := tab(upto("("))
   if \temp == "\\" then temp := "\\\\"
   put(L, \temp)
   List := L
   while tab(upto(&letters ++ &digits ++ '\'\"')) do {
      temp := tab(many(&letters ++ &digits ++ '\'\"'))
         if proc(temp) then {
            L1 := []
            put(L1, temp)
            put(List, L1)
            List := L1
            }
         else put(List, temp)
      }
   writes(yyout, funcname, "(")
   writes_code_for_uneval(L)
   writes(yyout, ")")
   return
end
```

It was also decided that variables, strings and csets be differentiated by including 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 function "tab" along with a parameter containing a list with the function "upto" with the parameter 'e'.

To evaluate this list during runtime the

```
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, nargs;
   tended struct b_list *lpsrc;
   tended struct b_list *lpdest;
   struct b_list *lptemp;
   tended char *temp;

   lpsrc = lp;
   lpdest = alclist(lpsrc->size,lpsrc->size);

   DEBUGF(20,(stdout, "Resolving function name and parameters"));
   nargs = lpsrc->size -1;
   elsrc = (struct b_lelem *)lpsrc->listhead;
   proc.dword = D_Proc;

   BlkLoc(proc)= (union block *)strprc(&elsrc->lslots[0],nargs);
   if (BlkLoc(proc) == NULL) {
      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;
         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) && (strcspn(varname, "\'") == 0)) {
               /* drop the single 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]);
               parm = elsrc->lslots[i];
               cnv:C_string(elsrc->lslots[i], temp);
               }
            else if ((StrLen(elsrc->lslots[i])>0) && (strcspn(varname, "\"") == 0)) {
               /* drop the double 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]);
               parm = elsrc->lslots[i];
               cnv:C_string(elsrc->lslots[i], temp);
               }
            else {
               /* look up variable */
               cnv:C_string(elsrc->lslots[i],varname);
               if ( getvar(varname,&parm)== Failed) {
                  VariableLookupFailed(varname);
                  }
               eldest->lslots[i] = parm;
               cnv:C_string(parm, temp);
               }
            }
         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;
         }
      }
#   printf("List:\n");
#   printList(lpdest, 1);
   return lpdest;
   }
```

4.1.3 String and Pattern Concatenation Operators

It was decided to have a consistent concatenation operator for string manipulation and pattern definitions. The existing string concatenation operator

accepted only strings and csets, while Pattern concatenation accepts strings, c-sets and patterns. Also, a patterns and strings are different data types and string concatenation and pattern concatenation are handled in separate operations for these data types. 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 that 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 Revised Pattern Functions

The pattern function names have been revised to be lexically more consistent with the Unicon language by revising the function definitions file, `fdef.h`, and the patterns source file `fxpattrn.ri` in Unicon source files. The table below condenses the Table 2 to show the SNOBOL4 pattern functions with their corresponding Unicon functions for this implementation:

Table 3: Pattern Functions

SNOBOL4	Unicon
LEN(n)	Len(n)
SPAN(c)	Span(c)
BREAK(c)	Break(c)
ANY(c) and NOTANY(c)	Any(c)
TAB(n) and RTAB(n)	Tab(n)
REM	Rem()
POS(n) and RPOS(n)	Pos(n)
FAIL	Back()
FENCE	Fence()
ABORT	Cancel()
ARB	Arb()
ARBNO(p)	Arbno(p)
BAL	Bal()

4.2.1 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_RPos);
      } else {
         ConvertPatternArgumentInt(position,bp,PC_Pos);
      }
      return pattern(bp);
      }
end
```

The position that is passed to the function is a `struct descrip` in the Unicon virtual machine that is defined to be an integer. The details of the `struct descrip` can be found in Implementation of Icon and Unicon. [6] The `if` statement in this function checks to see if the value of the descrip object is negative. If it is negative then its value is changed to its complement the 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 changes ware made to Tab function.

The Rpos(n) and Rtab 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 type patterns in Unicon, example code to execute a benchmark problem will be compared with SNOBOL4 and Gaikaiwari's pattern statements, this proposal's pattern statements from both the anchored string scanning environment and the non-anchored pattern matching environment, and equivalent Unicon string scanning functions. They will be checked 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 to what it is attempting to achieve. Simplicity will be measured by the number of lines 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 used in Gaikaiwari's thesis were:

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

5.1 Decomposing phone numbers

For the purpose of Decomposing phone numbers in North America, each piece of example code will have to be able to identify the area code, trunk and the

remainder 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` and the remainder of the number as being `1212`.

In the string scanning environment it can be achieved with the following 22 lines of code:

```
procedure main()
   line := "Uncle Sam: (800)555-1212 or uncle.sam@us.gov"
   sep := ""
   a := 0
   line ? {
      tab(upto(&digits))
      if line[&pos - 1] === "(" then {
         tab(&pos - 1)
         a := move(5)
         a ? area := move(1) || tab(many(&digits)) || tab(any(')'))
      } else {
         area := tab(many(&digits))
         sep := tab(any(' -.'))
      }
      prefix := tab(many(&digits))
```

```
      if sep === "" then sep := tab(any(' -.'))
      else tab(any(sep))
      number := tab(many(&digits))
   }
   if *a = 5 then write(area || prefix || sep || number)
   else write("(" || area || ")" || prefix || 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 it can be achieved with the following 14 lines of code:

```
procedure main()
   line := "Uncle Sam: (800)555-1212 or uncle.sam@us.gov"
   threedigit := &digits || &digits || &digits
   fourdigit := threedigit || &digits
   area := (Any("(") || threedigit || ")" )
   pattern := ((threedigit => areaCode) || (Any(' -.') => sep)
         || (threedigit => trunk) || Any(`sep`)
         || (fourdigit => remainder))
      .| ((area => areaCode) || (threedigit => trunk)
         || (Any(' -.') => sep) || (fourdigit => remainder))
   line ?? pattern
   if *areaCode == 5 then write(areaCode || trunk || sep || remainder)
   else write("(" || areaCode || ")" || trunk || sep || remainder)
end
```

In this example, the pattern is defined near the beginning of the code and allows the phone number to be found deductively. It is the shortest of the three examples.

When using patterns within the string scanning environment it can be achieved with the following 20 lines of code:

```
procedure main()
   out := &output
   line := "Uncle Sam: (800)555-1212 or uncle.sam@us.gov"
   threedigit := &digits || &digits || &digits
   fourdigit := threedigit || &digits
   area := (Any("(") || threedigit || ")" )
   pattern := ((threedigit => areaCode) || (Any(' -.') => sep)
         || (threedigit => trunk) || Any(`sep`)
         || (fourdigit => remainder))
      .| ((area => areaCode) || (threedigit => trunk)
         || (Any(' -.') => sep) || (fourdigit => remainder))
   result := ""
   line ? {
      tab(upto(&digits))
      if line[&pos - 1] === "(" then {
         tab(&pos - 1)
         result := =pattern
      } else result := =pattern
   }
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.

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

Sudarshan Gaikaiwari used the first two examples in [2]. The first example demonstrates how this problem can be resolved using the string scanning environment.

```
procedure main()
   in := open("testfile.txt", "r") | stop("open failed")
   out := open("output.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 second example Gaikaiwari provided was how the same problem can be solved by using the pattern matching system he implemented. The code below has been revised to match this integration.

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

Again the code is much shorter.

In this 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 that the one used in the previous example.

```
procedure main()
   in := open("testfile.txt", "r") | stop("open failed")
   out := open("output.txt", "w")
   double := (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
```

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

```
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

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

procedure main()
   p := Pos(1) || Span("a" || .> a || Span("b" || .> b || Span("c") || .> c || Pos(0) || `te
   while write(line := read()) do
      if(line ?? p) then write(" accepted")
      else write(" rejected")
end

procedure main()
   p := Pos(1) || Span("a" || .> a || Span("b" || .> b || Span("c") || .> c || Pos(0) || `te
end
```

5.4 Number of times a word is used

```
procedure main(args)
   text := "This is a test if the emergency broadcast system. It is
      only a test. Had this been a real emergency, there would be
      instructions on what to do in the event of an emergency.  This
      is only a test."
   word := "emergency"
   pattern := ~&letters || word || ~&letters
   count := 0
   every text ?? pattern do count +:= 1
   write(count)
end
```

6 Conclusions

The implementation of patterns within the string scanning environment was successful. A unicon programmer is now able to use pattern matching in the string scanning environment in the anchored mode.

7 Future Work

For the pattern data type to be fully integrated into the Unicon Language, the string scanning functions need to be integrated into the pattern data type and pattern matching operation.

A
Pattern Facilities Language Reference

A.1 Pattern Variables

variable

signifies a static variable

‘variable‘

signifies an unevaluated variable in a pattern

A.2 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 alteration 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 enviornment to see if there are is match of the pattern on the right in the anchored mode.

A.3 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.

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.

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.

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
```

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.

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.

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.

References

[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*.