ADAPTING SNOBOL-STYLE PATTERNS TO THE UNICON LANGUAGE

BY

SUDARSHAN GAIKAIWARI

A thesis submitted to the Graduate School

in partial fulfillment of the requirements

for the degree

Master of Science

Subject: Computer Science

New Mexico State University

Las Cruces, New Mexico

July 2005

“Adapting Snobol-style Patterns to the Unicon Language,” a thesis prepared by Sudarshan Gaikaiwari in partial fulfillment of the requirements for the degree, Master of Science, has been approved and accepted by the following:

Linda Lacey
Dean of the Graduate School

Clinton L. Jeffery
Chair of the Examining Committee

Date

Committee in charge:

Dr. Clinton L. Jeffery, Chair

Dr. Joseph Pfeiffer, Jr

Dr. Steve Helmreich

DEDICATION

To

My parents

ACKNOWLEDGMENTS

I would like to thank my advisor, Clinton Jeffery, for his encouragement, interest, and patience.

ABSTRACT

ADAPTING SNOBOL-STYLE PATTERNS TO THE UNICON LANGUAGE

BY

SUDARSHAN GAIKAIWARI

Master of Science in Computer Science

New Mexico State University

Las Cruces, New Mexico, 2005

Dr. Clinton L. Jeffery, Chair

String scanning and pattern matching is used in diverse applications such as bioinformatics, natural language processing and web applications. However most mainstream languages do not provide facilities for string processing other than regular expressions. By directly supporting pattern matching, programming languages can provide programmers with greater power and productivity. One of the ultimate classic string processing languages is SNOBOL4, and its pattern data type is so much more powerful than the dominant paradigm of regular expressions that it is one of relatively few languages from the 1960's to survive to this day. Unfortunately, SNOBOL4 lacks essential concepts for large modern applications. Its successor Icon, intentionally lacks SNOBOL4's pattern data type. This thesis explores the reintroduction of SNOBOL-style patterns as a built-in type to a

modern object-oriented successor to Icon called Unicon.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Programming languages enhance productivity by providing facilities for common tasks. Searching for patterns in text and replacing them is one of the most common tasks in programming. This task is also interesting from the perspective of theoretical computer science. The present work explores enhancing Unicon with a more powerful pattern matching construct. The first chapter explores the context of this work by looking at applications of string processing. SNOBOL4 provided facilities for string processing in the form of its pattern matching framework, while Icon provides the same facilities in its string scanning framework. Both pattern matching and string scanning are also discussed in the first chapter. The second chapter presents the details of the syntax and usage of the pattern matching construct; it constitutes a user's guide to Unicon's new pattern matching construct. Chapter three deals with the internal implementation of the pattern data type and assists future maintainers of the code to understand and modify the pattern matching internals. Chapter four compares Unicon's pattern datatype with equivalent constructs in Perl, Icon and SNOBOL. Chapter five summarizes the results and discusses future work to enhance the pattern data type.

1.1 String processing and pattern matching

String pattern matching is a vital aspect of a wide range of fundamental application domains, and its role in analysis and synthesis of input/output guarantees its importance on a permanent basis. String pattern matching plays an important role in the fields of bioinformatics and natural language processing. Modern high-level languages that support string processing tend to cluster around the most comfortable theoretical ground, namely regular expressions and context free grammars, but the tasks performed during analysis and synthesis of strings often do not fit within these tidy classes of languages. Specialized libraries and tools such as yacc and lex have also been developed to perform string matching. In addition to the limits of their regular expression and context free notations, these tools are not integrated into the language. Programmers have to suffer the extra burden of having to learn a different syntax and notation, and they also face problems in debugging the code generated by these tools

One of the most successful early string processing languages is SNOBOL4 [7]. Its interpretive flexibility can be compared to Lisp, and its pattern data type is perhaps unsurpassed to this day. SNOBOL4's successor Icon offered numerous improvements but was less successful than SNOBOL4, possibly because it was a by-product of university research instead of an AT&T product, but possibly because in the process of successfully generalizing SNOBOL's pattern-matching primitives, Icon intentionally omitted SNOBOL's pattern data type in favor of a string scanning control structure. The pattern type turns out to be important for all the reasons that data is more convenient to manipulate than is code: easier composition and reuse, modification and dynamic altering of patterns on the fly, and so on. In addition, Icon users have been known to complain that their string scanning control structure is not as concise as SNOBOL patterns or regular expressions.

1.2 SNOBOL4

SNOBOL (StriNg Oriented symBOlic Language) is a computer programming language that was developed between 1962 and 1967 at AT&T Bell Laboratories by David J. Farber, Ralph E. Griswold and Ivan P. Polonsky. The first SNOBOL language was designed and implemented in 1962-63. The motivation for the development of SNOBOL was the need for a general purpose programming language for string processing. SNOBOL was superseded by SNOBOL3 in 1965 which in turn was superseded by SNOBOL4 in 1967. SNOBOL4 had many features that we see in popular dynamic programming languages such as dynamic typing, eval and garbage collection. From the string processing perspective its most important contribution was the pattern data type.

1.2.1 **Overview**

SNOBOL4 is really a language composed of two distinct sublanguages. One sublanguage is a conventional imperative language with a rich set of data types and a simple control structure. The other is the pattern language with its own substructure. SNOBOL programs consist of statements which are of the following form:

```
label subject = object goto
label subject pattern goto
label subject pattern = object goto
```

The first statement is an assignment statement. It assigns a value to the subject. The second statement is a pattern matching statement, which examines the value of the subject for a pattern. The third statement is a pattern replacement statement which modifies the part of the subject matched by the pattern. The optional label identifies the statement. The optional goto indicates the next statement to be evaluated.

SNOBOL4 supports integers, reals and strings as primitive data types. Depending on the operation, operands can be seamlessly converted from one type to another. SNOBOL4 also has support for heterogeneous arrays and tables. The following example SNOBOL program concatenates the first three elements of a table. In SNOBOL space seperated strings variables are concatenated. So `STRINGOUT ':' A<I>` concatenates the strings `STRIGNOUT, :, A<I>`

```
A = TABLE(6)
A<2> = 'two'
A<1> = 'one'
A<3> = 'three'
STRINGOUT I  =  1
STRINGOUT  =  A<1>:F(END)
STRINGOUT_1 I  =  I + 1
GT(I,3) :S(PRINT)
STRINGOUT  =  STRINGOUT  ':'  A<I>:S(STRINGOUT_1)
PRINT    OUTPUT = STRINGOUT
END
```

SNOBOL4 shows its age in its control structure, which is based on control signals that can be either SUCCESS or FAILURE. The optional goto field of a statement transfers control. This field takes one of the following forms

```
:(label)
:S(label)
:F(label)
:S(label1) F(label2)
```

White space (space or tab) is required before the colon. `label` is the identifier of the target statement, and must be enclosed in parentheses. If the first form is used, execution resumes at the referenced statement, unconditionally. In the second and third forms, transfer to the referenced statement occurs only if the statement has succeeded or failed, respectively. Otherwise, execution proceeds to the next statement in line. If the fourth form is used, transfer is made to label1 if the statement succeeded, or to label2 if it failed.

Since goto is the only form of control transfer in SNOBOL4, it is difficult to do structured programming in SNOBOL4. Hence SNOBOL4 is not used for large scale application development today.

1.2.2 Patterns

SNOBOL4 introduced patterns as a first class data object that can be created with functions and operators. This allowed construction of complicated patterns by composing them from simpler elements. The two basic pattern construction operators are concatenation and alternation. The concatenation of two patterns is a pattern that will match anything that its two components will match consecutively. The alternation of two patterns is a pattern that will match anything that either of its two components will match. Alternation is represented by a vertical bar and concatenation by a blank; e.g.

```
CHARACTER = 'COWARD' | 'CRAVEN'
ACTTYPE = CHARACTER  'LY'
```

The pattern CHARACTER matches either of the strings COWARD or CRAVEN, and ACTTYPE matches anything CHARACTER matches followed by the string 'LY' (i.e. 'COWARDLY' or 'CRAVENLY').

Pattern-valued functions generalize the concept of patterns and avoid special notations for each type. For example, the value returned by LEN(n) is a pattern that matches n characters, and the pattern returned by TAB(n) matches a substring through the nth character of the subject string. For example,

```
OPER = TAB(6) 'X'
```

creates a pattern that will match any string containing an X as its seventh character. Other pattern valued functions create patterns that match any one of a number of specific characters, search for specific characters, etc. Examples are SPAN('0123456789'), which matches a substring consisting only of digits, and BREAK(';,'), which matches the substring beginning at the current position up to the next comma or semicolon.

In SNOBOL4, pattern matching is left to right, and components must match consecutive substrings of the subject string. When a component fails to match, alternative matches are attempted. If no alternative is specified, the pattern-matching process backs up to earlier, successfully matched components, seeking other ways in which the entire pattern match can succeed. Conceptually, the pattern-matching process manipulates a cursor, which is an imaginary marker in the subject string indicating the current position of the match. Movement of the cursor is implicit, not under direct control of the programmer, although in some patterns there is a direct correlation. Thus, LEN(3) moves the cursor to the right three characters. The cursor cannot be moved to the left by a successful match.

SNOBOL4 also supports the assignment of the substring matched by a pattern component to a variable during pattern matching. The operator for assignment when the whole pattern succeeds is . The operator allows the cursor position to be assigned to a variable. The operator for immediate assignment is the binary $ operator, e.g.

```
HEAD = LEN(7) $ LABEL
```

constructs a pattern that matches seven characters. The seven characters, when matched, are assigned to LABEL, so LINE HEAD assigns the first seven characters of the value of LINE to LABEL. If the match fails (as it would because LINE is less than seven characters long), no assignment is made to LABEL. Another aspect of pattern matching is the ability to modify the pattern during matching depending on substrings matched by earlier components. Evaluation of an expression in a pattern may be deferred by prefacing the expression with *. The expression is then left unevaluated until it is encountered in pattern matching. An example of the power of this facility is given by

```
SPECPAT = LEN(1) $ D BREAK(*D) . SEARCH LEN(1)  BREAK(*D) . REPLACE
```

which recognizes strings of the form

```
/PROGRAMMING/CODING/
```

as well as

```
$PROGRAMMING$CODING$.
```

The separator is matched by LEN(1) and assigned to D. Since D is passed as unevaluated parameter to the BREAK pattern it is not evaluated during pattern matching unless it has been assigned by LEN. This allows the pattern to recognize various forms of the pattern regardless of what character is used as a separator.

SNOBOL allows users to create concise programs for grammars that are beyond the capablities of regular expressions and context free grammars. A SNOBOL pattern fragment (matching $a^n b^n c^n$) is

```
ANBNCN = POS(0) SPAN('A') @A SPAN('B') @B SPAN('C')
                @C RPOS(0) *((EQ(B - A,A) (EQ(C - B,A) )
```

1.3 Icon

Icon [8] is a successor to SNOBOL4 developed at University of Arizona from 1979-1983 and refined over the following decade. Icon's research contribution is to generalize and make explicit goal-directed evaluation, including generators and backtracking, that previously was implicit and built-in to SNOBOL pattern matching.

1.3.1 Overview

Icon is an imperative, procedural language with a syntax very similar to C or Pascal. In Icon variables are dynamically typed. Icon supports a large number of data types: co-expression, cset, file, integer, list, null, procedure, real, set, string, table and window. The aggregate types - sets, lists, tables, and records - can hold values of any type. Tables can be indexed by values of any type. Integers, reals, character sets, and strings are atomic values; operations on them produce new values. Aggregates use implicit pointer semantics; operations on them can change existing values as well as produce new ones. Strings and aggregates can be of arbitrary size, and their sizes can change during execution. Memory is managed using garbage collection.

Icon has an expression-oriented syntax; even control structures are expressions. Procedures consist of zero or more expressions separated by newlines or semicolons. Icon programs consist of one or more procedure definitions, and execution begins by calling the procedure named main.

Expressions can produce zero or more values. As in traditional languages, many Icon expressions produce a single value, for example, `a + 1` produces the sum of `a` and `1`. Other expressions can produce more than one value, for example, `a | 5` produces `a` then, if necessary, produces `5`. Such expressions are called *generators*; there are several built-in generators and procedures can be written to be generators. Other examples of built-in generators include `x to y`, which generates the integers from x to y, and `!x`, which generates all the characters from string x or the elements from aggregate x.

A *goal-directed* expression evaluation mechanism is Icon's most distinguishing characteristic. Expression evaluation seeks *success* - at least one value for an expression. An expression *fails* if it does not produce a value. The evaluation mechanism tries all combinations of values from generators in pursuit of a successful outcome. For example, `y < (x | 5)` first compares y to x. If y is less than x, evaluation succeeds and produces the value x. If y is not less than x, y is compared to 5, the next value generated by the subexpression `x | 5`. If y is less than 5, evaluation succeeds and produces 5. Otherwise, evaluation fails, and no value is produced. Comparison operators produce the value of their right operand if they succeed.

Failure drives control expressions and inhibits subsequent evaluation. For example, `max := max < x` sets max to x only when it is less than x. Likewise,

```
if y < (x | 5) then write("y=", y)
```

prints the value of y only if it is less than x or 5. The evaluation mechanism pervades Icon; for example, procedures are called only if all of their arguments succeed, so the last example could be written more succinctly as

```
write("y=", (x | 5) > y)
```

The backtracking implied by the goal-directed evaluation mechanism is limited to the expression in which it occurs. For example, in

```
max := max < x
write("y=", (x | 5) > y)
```

failure in the second expression does not affect the outcome of the first. The expression `every e1 do e2` ‘drives’ e1 - it evaluates e2 for every successful outcome of e1. So, if p is a list of 100 elements,

```
every i := (1 to 10) | (91 to 100) do write(p[i])
```

prints the first and last ten elements of p. The do clause is optional, and the evaluation mechanism often helps eliminate temporary variables like i, for example,

```
every write(p[(1 to 10) | (91 to 100)])
```

also prints the first and last ten elements of p, and

```
every !p := 0
```

sets each element of p to zero.

1.3.2 String Scanning

The pattern matching facilities in SNOBOL as well as other languages such as Perl focus on pattern specification. However the programmer has no control over how the pattern matching process works. String scanning in Icon is unique in that all of Icon's functionality is available during string scanning. String scanning in Icon is setup by the expression `s ? e`. This expression establishes `s` as the subject to which string processing operations in `e` apply. `e` contains a series of expressions which are typically string analysis operations,but may include any Icon operation. The expression `s ? e` creates a scanning environment which is characterized by a pair of implicit variables `&subject`,`&pos`. The subject is the string on which the scanning operations are applied. `&pos` is a location in the subject and changes as the subject is analyzed. It is initialized to 1. Once the scanning environment is setup `e` is evaluated. After evaluating `e`, the previous scanning environment, if any is restored.

The following functions are commonly used during string scanning in Icon

Function	Effect
move(n)	relative adjustment; string result
tab(n)	absolute adjustment; string result

Table 1.1: Functions that change `&pos`

Function	Effect
many(cs)	produces position after run of characters in cs
upto(cs)	generates positions of characters in cs
find(s)	generates positions of s
match(s)	produces position after s, if s is next
any(cs)	produces position after a character in cs

Table 1.2: Analytical functions that return a position in the subject

Function	Effect
pos(n)	tests if &pos is equal to n
bal(cs1,cs2,cs3)	locate balanced characters

Table 1.3: Other functions:

The following code prints the first seven characters of a line in Icon

```
line ? write(move(7))
```

To achieve the similar effect in SNOBOL we first have to construct the pattern, then apply it and finally print the output.

Icon's string scanning code can be more verbose than corresponding SNOBOL pattern matching. Here is an example Icon string scanning fragment (matching $a^n b^n c^n$)

```
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
```

1.4 Alternatives for improving upon string scanning

The author explored various approaches to improving string scanning in Unicon before implementing the approach presnted in this thesis. The first approach added operator overloading to Unicon and implemented a pattern matching class with overloaded operators. These operators had semantics similar to the regular expression operators. However this approach led to patterns that were too verbose and cumbersome. The other approach tried by the author was to embed a SNOBOL interpreter in Unicon. For this an existing public domain SNOBOL interpreter CSNOBOL4 [2] was used. Although this allowed programmers to embed SNOBOL patterns in Unicon code it was found that this approach had major performance issues.

CHAPTER 2

THE UNICON PATTERN DATA TYPE

2.1 Overview

Unicon patterns are closely related in their syntax and semantics to SPITBOL [3] patterns. It is trivial to translate SNOBOL4/SPITBOL pattern construction code to Unicon patterns. This chapter explains Unicon patterns in detail so that programmers who have never been exposed to SNOBOL can understand and use them. Where necessary differences between Unicon patterns and SNOBOL patterns are also pointed out. The simplest operands for pattern matching in Unicon are strings and character sets. These match themselves when used in a pattern matching expression. "??" is the pattern matching operator in Unicon, for example:

```
"Unicon" ?? "nic"
```

Like all Unicon expressions this expression can either succeed or fail. If the pattern is found in the subject then the expression succeeds and returns the first substring of the subject matched by the pattern. In the above expression a substring corresponding to the 2nd to 5th characters of the subject is returned. While strings and characters can be used in pattern matching expressions the pattern data type allows complicated patterns to be constructed and stored in variables. This pattern construction is done with the following operators and functions.

2.2 Operators

Patterns are mainly built up from simple components using alternation ".|" and concatenation "&&". These pattern operators were chosed to avoid clashes with Unicon's many other operators. A pattern formed by alternation of two elements succeeds if either of the elements matches. A pattern formed by concatenation of two elements matches if both those elements match consecutively. When two patterns are joined together by alternation they are called alternates. When two patterns P1 and P2 are combined using concatenation as P1 && P2 then P2 is the subsequent of P1. Pattern concatenation has higher priority than pattern alternation. So the pattern P1 .| P2 && P3 matches either the pattern P1 or the pattern formed by concatenation of P2 and P3. Parentheses can be used to group patterns differently. (P1 .| P2) && P3 matches P1 or P2 followed by P3.

Thus patterns can be composed from subpatterns. When a subpattern successfully matches a portion of the subject, the matching subject characters are bound to it. The next sub pattern in the pattern must match beginning with the very next subject character. If a subsequent fails to match, the pattern backtracks, unbinding patterns until another alternative can be tried. A pattern match fails when an alternative that matches cannot be found.

Suppose we wanted to construct a pattern that matched any of the following strings: 'COMPATIBLE', 'COMPREHENSIBLE' and 'COMPRESSIBLE'. The pattern can be constructed by

```
``COMP'' && (``AT'' .| "RE" && (``HEN'' .| ``S'') && "S") && ``IBLE''
```

One way to understand patterns is to construct bead diagrams for them. In a bead diagram, pattern matching is the process of attempting to pass a needle and thread through a collection of beads which model the individual pattern components. Pattern subsequents are drawn side-by-side, left-to-right. Pattern alternates are stacked vertically, in columns, with a horizontal line between each alternative. Here is a bead diagram for the above

Consider the following pattern and its application

```
Manufacturer := ``SONY'' .| ``DELL''
Type := ``Desktop'' .| ``Laptop''
Machine := Manufacturer && Type
``SONY Laptop'' ?? Machine
```

The pattern matching will succeed. However we would also like to determine which of the pattern alternatives actually matched. For this we use the conditional binary operator −>. The operator is called conditional, because assignment occurs only if the pattern match is successful. It assigns the matching

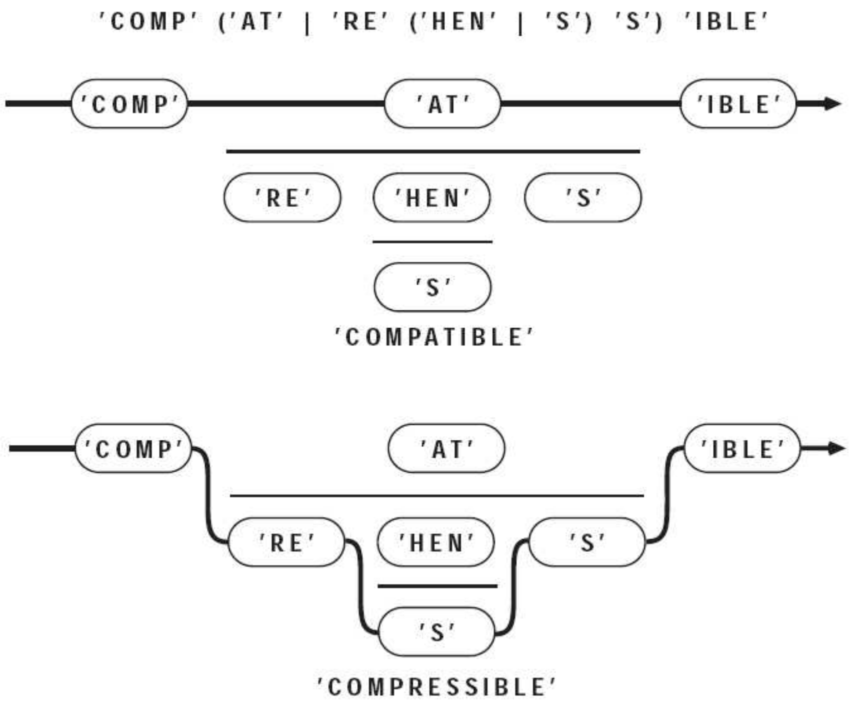


Figure 2.1: Bead Diagram

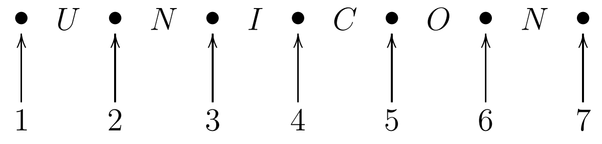
substring on its left to the variable on its right. Note that the direction of assignment is just the opposite of the assignment operator :=. Changing the above example to use conditional assignment.

```
        Manufacturer := (‘‘SONY’’ .| ‘‘DELL’’) -> Corporation
        Type := (‘‘Desktop’’ .| ‘‘Laptop’’) -> SysType
        Machine := Manufacturer && Type
        ‘‘SONY Laptop’’ ?? Machine
        write(Corporation)
        write(SysType)
OUTPUT
        SONY
        Laptop
```

The immediate assignment operator `$$` allows us to capture intermediate results during the pattern match. Immediate assignment occurs whenever a subpattern matches, even if the entire pattern match ultimately fails. Like conditional assignment, the matching substring on its left is assigned to the variable on its right. Immediate assignment is often used as a debugging tool to observe the pattern matching process. When used with unevaluated expressions immediate assignment allows creation of a powerful class of patterns as we will see later.

Both the assignment operators perform the write operation if the assignment variable happens to be a file handle.

During a pattern match, the cursor is Unicon's pointer into the subject string. It is integer valued, and points between two subject characters. It may also may be positioned before the first subject character, or after the final subject character. Its value may never exceed the size of the subject string by more than 1. Heres an example of the numbering for the substring string UNICON:



The cursor is set to 1 when a pattern match begins, corresponding to a position immediately to the left of the first subject character. As the pattern match proceeds, the cursor moves right and left across the subject to indicate where Unicon is attempting a match. The value of the cursor is assigned by the unary cursor position operator `.$` to a variable. It appears within a pattern, preceding the name of a variable. For example,

```
p := ( "b" .| "r") && ("e" .|"ea") && ("d" .| "ds")
pattern := .$x && p && .$y
write("the beads are red")
if "the beads are red" ?? pattern then
        write(repl(" ",x - 1) , repl("_", y - x))
```

The above code will underline the part of the substring that is matched by the pattern. The output is

```
the beads are red
    -----
```

Summary of Unicon pattern matching operators

Operator	Operation
&&	Pattern concatenate
.|	Pattern alternation
−>	Conditional assignment
$$	Immediate assignment
.$	Cursor position assignment

Table 2.1: Pattern Construction Operators

The previous examples used patterns created from literal strings. Instead of specific characters, qualities of the string to be matched can also be specified.The ability to specify these qualities makes patterns powerful at recognizing more abstract patterns. There are 3 different types of pattern construction functions that allow specification of pattern qualities:

- Integer pattern functions

- Character pattern functions

- Pattern primitives

2.3 Integer Pattern Functions

These pattern construction functions take an integer as a parameter and return a pattern as result. The integer pattern functions are:

2.3.1 PLen(I): Match fixed-length string

PLen(I) produces a pattern which matches a string exactly I characters long. I must be an integer greater than or equal to zero. Any characters may appear in the matched string. For example, PLen(5) matches any 5-character string, and PLen(0) matches the null string. PLen() maybe constrained to certain portions of the subject by other adjacent patterns:

```
        out := &output
        s := "abcda"
        s ?? PLen(3) -> out
        s ?? PLen(2) -> out && "a"
OUTPUT
abc
cd
```

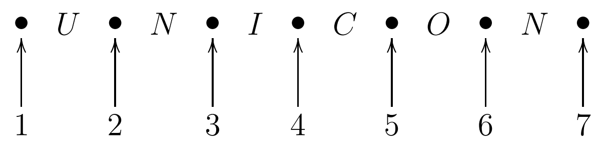
The first pattern match had only one constraint, that the subject had to be at least three characters long. Thus PLen(3) matched its first three characters. The second case imposes the additional restriction that PLen(2)'s match be followed immediately by the letter ”a”. This disqualifies the intermediate match attempts ”ab” and ”bc”.

Using PLen() with keyword &ascii as the subject provides a simple way to obtain a string of unprintable characters. For example, the ASCII control characters occupy positions 0 through 31 in the 256-character ASCII set. To obtain a 32-character string containing these control codes, use:

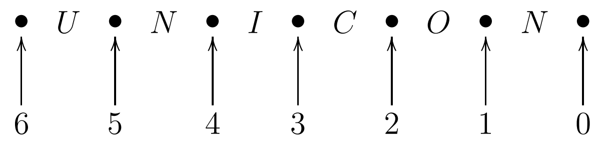
```
&ascii ?? PLen(32) -> controls
```

2.3.2 PPos(I), PRpos(I): Verify cursor position

The PPos(I) and PRpos(I) patterns do not match subject characters. Instead, they succeed only if the current cursor position is a specified value. They often are used to tie points of the pattern to specific character positions in the subject. The following shows the cursor positions as used by PPos():



The following are the cursor postions used by PRpos().



PPos(I) counts from the left end of the subject string, succeeding if the current cursor position is equal to I. PRpos(I) is similar, but counts from the right end of the subject. If the subject length is N characters, PRpos(I) requires the cursor be (N - I). If the cursor is not the correct value, these functions fail, and the pattern matcher tries other pattern alternatives.

```
        out := &output
        s := "abcda"
        if s ?? PPos(1) && "b" then
                write("Match succeeded")
        else
                write("Match failed")
        s ?? PLen(3) -> out && PRpos(0)
        s ?? PPos(4) && PLen(1) -> out
        if s ?? PPos(1) && "abcd" && PRpos(0) then
                write("Match succeeded")
        else
                write("Match failed")   out := &output
   s := "abcda"
   s ?? PLen(3) -> out
   s ?? PLen(2) -> out && "a"
OUTPUT
Match failed
cda
d
Match failed
```

The first example requires a ”b” at cursor position 1, and fails for this subject. PPos(1) anchors the match, forcing it to begin with the first subject character. Similarly, PRpos(0) anchors the end of the pattern to the tail of the subject. The next example matches at a specific mid-string character position, PPos(3). Finally, enclosing a pattern between PPos(1) and PRpos(0) forces the match to use the entire subject string. At first glance these functions appear to be setting the cursor to a specified value. Actually, they never alter the cursor, but instead wait for the cursor to come to them as various match alternatives are attempted.

2.3.3 PRtab(I), PTab(I): Match to fixed position

These patterns are hybrids of PArb(), PPos(), and PRpos(). They use specific cursor positions, like PPos() and PRpos, but match subject characters, like PArb(). PTab(I) matches any characters from the current cursor position up to the specified position I. PRtab(I) does the same, except, as in PRpos(), the target position is measured from the end of the subject. TAB and RTAB will match the null string, but will fail if the current cursor is to the right of the target. They also fail if the target position is past the end of the subject string. These patterns are useful when working with tabular data. For example, if a data file contains name, street address, city and state in columns 1, 30, 60, and 75, this pattern will break out those elements from a line:

```
P = PTab(30) -> NAME && PTab(60) -> STREET && PTab(75) -> CITY && PRest() -> ST
```

The pattern PRtab(0) is equivalent to primitive pattern PRest(). It counts from the right end of the subject, but matches to the left of its target cursor. Example:

```
        out := &output
        s := "abcde"
        s ?? PTab(3) -> out && PRtab(1) -> out
OUTPUT
ab
cd
Success
```

PTab(3) matches `"ab"`, leaving the cursor at 2, between `"b"` and `"c"`. The subject is 5 characters long, so PRtab(1) specifies a target cursor of 6 - 1, or 5, which is between the ”d” and ”e”. PRtab() matches everything from the current cursor, 3, to the target, 5.

2.4 Character Pattern Functions

These functions produce a pattern based on a character set argument. The argument passed to these functions is always converted to a character set

2.4.1 PAny(S), PNotAny(S): Match one character

PAny(S) matches the next subject character if it appears in the string S, and fails otherwise. PNotAny(S) matches a subject character only if it does not appear in S. Here are some sample uses of each:

```
        out := &output
        vowel := PAny("aeiou")
        dvowel := vowel && vowel
        notvowel := PNotAny("aeiou")
        "vacuum" ?? vowel -> out
        "vacuum" ?? dvowel -> out
        "vacuum" ?? (vowel && notvowel) -> out
OUTPUT
a
uu
ac
```

2.4.2 PBreak(S), PSpan(S): Match a run of characters

These are multi-character versions of PNotAny and PAny. Each requires a non-null string argument to specify a set of characters. PSpan(S) matches one or more subject characters from the set in S. PSpan() must match at least one subject character, and will match the longest subject string possible. PBreak(S) matches up to but not including any character in S. The string matched must always be followed in the subject by a character in S. Unlike PSpan() and PNotAny(), PBreak() will match the null string. These two functions are called stream functions because each streams by a series of subject characters. PSpan() is most useful for matching a group of characters with a common trait. For example, we can say an English word is composed of one or more alphabetic characters, apostrophes, and hyphens. A pattern for this is:

```
word := PSpan(&letter ++ "'-")
```

Some patterns that can be formed by PSpan() and PBreak().

Pattern	Function
a run of blanks	PSpan(" ")
a string of digits	PSpan(&digits)
a run of letters	PSpan(&lettters)
everything up to the next blank	PBreak(" ")
everything up to the next punctuation mark	PBreak(",.;:!?")

2.4.3 **PBreakx(): Extended PBreak() function**

Like SPITBOL, Unicon offers an extended version of PBreak() called PBreakx(). If necessary, PBreakx() will look past the place where it stopped to see if a longer match is possible. It will do this if some subsequent pattern element fails to match. The pattern matcher checks to see if extending PBreakx() might allow the subsequent pattern element match. If so, the operation succeeds. If not, other pattern alternatives (if any) prior to PBreakx() are attempted. Suppose the pattern needs to match everything before the first "e" in a subject string, as with:

```
        out := &output
        "integers" ?? PBreak("e") -> out
OUTPUT
int
```

PBreak works fine in this scenario, however if whatever comes before the first occurrence of a two-letter pattern ”er” is to be matched, then PBreakx() is more appropriate.

```
        out := &output
        "integers" ?? PBreakx("e") -> out && "er"
OUTPUT
integ
```

PBreakx() stopped at the first “e” in “integer”, and tried to match the next pattern element, the two letters “er”. But the next subject characters were “eg”, a mismatch, so PBreakx() was instructed to try again. PBreakx() extended itself to the next “e”, where “er” in the subject matches “er” in the pattern. The above example illustrates that PBreak(S) will never return a string containing any characters in S, while PBreakx(S) might, if a subsequent pattern requires it. PBreakx(S) provides a more selective, and more efficient version of the PArb() pattern. For example the following construction could have been used :

```
        out := &output
        "integers" ?? PArb() -> out && "er"
OUTPUT
integ
```

but PArb() pokes along one character at time, matching “i”, “in”, “int”, and “inte”, before finding the desired match, “integ”. In contrast, PBreakx() gets the right answer after only two attempts: “int” and “integ”. The increased efficiency is even more pronounced with a long subject.

2.5 Primitives

There are seven primitives built into the Unicon pattern matching system. They are:

2.5.1 PRest: Match remainder of the subject

PRest will match zero or more characters at the end of the subject string.

Example

```
        out := &output
       "NMSU Aggies" ?? "NMSU" && PRest() -> out
OUTPUT
 Aggies
```

The subpattern “NMSU” is matched at its first occurrence in the subject. PRest is matched from there to the end of the subject. If the example is changes to

```
        out := &output
        "NMSU Aggies" ?? "Aggies" && PRest() -> out
OUTPUT
```

the ”Aggies” matches at the end of the string leaving an empty remainder for PRest. PRest then matches the null string and the assignment to the out causes just a newline to be written to the standard output.

The pattern components to the left of PRest must successfully match some portion of the subject string. PRest begins after the last character matched by the earlier pattern component and matches all subject characters till the end of the string. There is no restriction on the particular characters matched.

2.5.2 **PArb(): Match arbitrary characters**

PArb matches an arbitrary number of characters from the subject string. It matches the shortest possible substring, including the null string. The pattern components on either side of PArb determine what is matched. Example:

```
        out := &output
        "Pragmatic Programmer" ?? "a" && PArb() -> out && "a"
OUTPUT
gm
        out := &output
        "Pragmatic Programmer" ?? "a" && PArb() -> out && "g"
OUTPUT
```

In the first statement, the PArb() pattern is constrained on either side by the known patterns “a” and “a”. PArb() expands to match the subject characters between, “gm”. Note the smaller substring between two occurences of “a” is matched. In the second statement, there is nothing between “a” and “g”, so PArb() matches the null string. PArb() behaves like a spring, expanding as needed to fill the gap defined by neighboring patterns.

2.5.3 PArbno(): Match zero or more consecutive occurences of pattern

This function produces a pattern which will match zero or more consecutive occurrences of the pattern specified by its argument. PArbno() is useful when an arbitrary number of instances of a pattern may occur. For example, PArbno(LEN(3)) matches strings of length 0, 3, 6, 9, ... There is no restriction on the complexity of the pattern argument. Like the PArb() pattern, PArbno() tries to match the shortest possible string. Initially, it simply matches the null string. If a subsequent pattern component fails to match, Unicon backs up, and asks PArbno() to try again. Each time PArbno() is retried, it supplies another instance of its argument pattern. In other words, PArbno(PAT) behaves like

```
( "" | PAT | PAT PAT | PAT PAT PAT | ... )
```

Also like PArb(), PArbno() is usually used with adjacent patterns to draw it out. Consider the following example which tests for a list of one or more numbers separated by commas and enclosed by parentheses.

```
        item := PSpan(&digits)
        list := PPos(1) && "(" && item && PArbno("," && item) && ")" &&
                PRpos(0)
        if "(12,345,6)" ?? list then
                write("Match succeeded")
        else
                write("Match failed")
        if "(12,,6)" ?? list then
                write("Match succeeded")
        else
                write("Match failed")
OUTPUT
```

```
Match succeeded
Match failed
```

PArbno() is retried and extended till the subsequent ")" matches. PPos(1) and PRpos(0) force the pattern to be applied to the entire subject string.

2.5.4 PAbort(): End pattern match

The PAbort() pattern causes immediate failure of the entire pattern match, without seeking other alternatives. Usually a match succeeds when we find a subject sequence which satisfies the pattern. The PAbort pattern does the opposite: if we find a certain pattern, we will abort the match and fail immediately. For example suppose we are looking for an `"a"` or `"b"`, but want to fail if `"1"` is encountered first:

```
        if "ab1" ?? PAny("ab") .| "1" && PAbort() then
                write("Match succeeded")
        else
                write("Match failed")
        if "1ab" ?? PAny("ab") .| "1" && PAbort() then
                write("Match succeeded")
        else
                write("Match failed")
OUTPUT
Match succeeded
Match failed
```

The second pattern matching expression deserves some elaboration as it shows how the pattern matching engine works. At each cursor position all pattern alternatives are tried. In the second pattern matching expression after the literal

“1” matches PAbort() matches causing the pattern to fail. The PAny() pattern thus does not get a chance to match the character ”a” at cursor position 2.

2.5.5 PBal(): Match balanced string

The PBal() pattern matches the shortest non-null string in which parentheses are balanced. (A string without parentheses is also considered to be balanced.) According to PBal() the following strings are balanced:

```
(X) Y (A!(C:D)) (AB)+(CD) 9395 (8+-9/2)
```

and these are not:

```
)A+B (A*(B+) (X))
```

PBal() is concerned only with left and right parentheses. The matching string does not have to be a well-formed expression in the algebraic sense. Like PArb(), PBal() is most useful when constrained by other pattern components. For example:

```
        out := &output
        "ab+(14-2)*c" ?? PAny("+-*/") && PBal() -> out &&
                        PAny("+-*/")
OUTPUT
(14-2)
```

An example of using PBal() primitive to manipulate algebraic expressions is provided in the next chapter.

2.5.6 PFail(): Seek other alternatives

The PFail pattern signals the failure of this portion of the pattern match, causing the pattern matcher to backtrack and seek other alternatives. PFail() will also suppress a successful match, which can be very useful when the match is being performed for its side effects, such as immediate assignment. For example this fragment will display the subject characters, one per line:

```
out := &output
subject ? PLen(1) $$ out $$ PFail()
```

PLen(1) matches the first subject character, and immediately assigns it to out. PFail() tells the pattern matcher to try again, and since there are no other alternatives, the entire match is retried at the next subject character. Forced failure and retries continue until the subject is exhausted. The difference between PAbort() and PFail() is that PAbort() stops all pattern matching, while PFail() tells the system to back up and try other alternatives or other subject starting positions.

2.5.7 PFence(): Prevent match retries

Pattern PFence() matches the null string and has no effect when the pattern matcher is moving left to right in a pattern. However, if the pattern matcher is backing up to try other alternatives, and encounters PFence(), the match fails. PFence() can be used to lock in an earlier success. Consider the following example:

The pattern succeeds if the first “a” or “b” in the subject is immediately followed by a plus sign.

```
"1ab+" ?? PAny("ab") && PFence() && "+"
```

In the example above, the pattern matcher matches the “a”s match and goes through the PFence(), only to find that “+” does not match the next subject character, “b”. The pattern matcher then tries to backtrack, but is stopped by the PFence() and fails. If PFence() were omitted, backtracking would match PAny to “b”, and then proceed forward again to match “+”. If PFence() appears as the first component of a pattern, the pattern matcher cannot back up through it to try another subject starting position. This allows Unicon to simulate SNOBOL’s anchored pattern matching mode without requiring a new &anchored keyword.

2.5.8 PSucceed(): Match Always

This pattern was added to Unicon to match SNOBOL4 pattern for pattern. The author does not know of any useful use of PSucceed(). The frivolous sawtooth example in the next chapter is the only place where the author has seen PSucceed() being used.

2.6 Unevaluated Expressions

Consider the following pattern construction example which captures the next N characters after a colon:

```
npat := ":" && PLen(N) -> item
```

This pattern is static in the sense that the value of N at time of pattern construction is captured by the pattern. Even if `N` subsequently changes the pattern uses the original value of `N`. One way to use the current value of `N` is to specify the pattern each time it is used.

```
SUBJECT ?? ":" && PLen(N) -> item
```

However this is not only inefficient, but also a possible maintainence nightmare. The "unevaluated expression" facility allows us to obtain the efficiency of static pattern construction yet use the current value of variables. An unevaluated expression is constructed by enclosing it in "‘‘". So we can construct the pattern `npat` as

```
npat := ":" && PLen(‘N‘) -> item
```

The pattern is only constructed once, and assigned to `npat`. `N`'s current value is ignored at this time. Later,when `npat` is used in a pattern match, the deferred evaluation operator fetches the then current value of `N`. Deferred evaluation may appear as the argument of the pattern functions PAny(), PBreak(), PBreakx(), PLen(), PNotAny(), PPos(), PRpos(), PRtab(), PSpan(), or PTab().

```
out := &output
pat := PTab(‘i‘) -> out && PSpan(‘s‘) -> out
sub := "123aabbcc"
i := 5
s := "ab"
sub ?? pat
i := 4
sub ?? pat
```

```
OUTPUT
123a
abb
123
aabb
```

Note that `i` and `s` were undefined when `pat` was first constructed.

2.6.1 Immediate Assignment

Immediate assignment can be used as a debugging aid to view the pattern matching process. However combined with unevaluated expressions immediate assignments give rise to a powerful class of patterns. In these patterns a variable that is assigned to during the pattern matching process is used later in the very same pattern match. Consider the following example where the first part of the subject contains the length of the field that is to be extracted. So the pattern first assigns the length to a variable `n` and uses that variable as a argument to PLen().

```
        fpat := PSpan(&digits) $$ n && PLen(‘n‘) -> field
        "12abcdefghijklmn" ?? fpat
        write(field)
OUTPUT
abcdefghijkl
```

2.6.2 Limitations due to lack of eval()

A major difference between the SNOBOL4 and Icon families is that Icon lacks a general eval facility that is available in SNOBOL. To overcome this limitation the pattern feature was augmented so that certain operations will be evaluated during the pattern matching phase instead of the pattern construction phase.

Unicon supports the following language constructs in unevaluated expressions.

- function call.
- method call.
- variable reference.
- field reference.

Note the function or method calls can only have identifiers or constants as parameters. Expressions as parameters are not allowed. Multiple field references are of the form `x.y.z` are not supported. An unevaluated function or method call is used in two ways. The pattern matcher performs the call and if it fails treats the failure as if a pattern sub component has failed and tries to backtrack in seach of other alternatives. However if the pattern call succeeds then the pattern matcher can either ignore the result of the function or use the result in the pattern matching process. To specify that the result should be used in pattern matching the unevaluated function call is enclosed in ‘ ‘. Unevaluated expressions can also include operators for example to check if a variable is non null

```
‘\(x)‘
```

binary operators can be used using the prefix notation. Only one operator or function call can be used in an unevaluated expression. Unicon programs that use unevaluated expressions must be compiled with the ”-f s” option which enables full string invocation.

2.7 Pattern expressions and Unicon control structure

One of the goals while introducing pattern types in Unicon was to ensure that it integrated well with Unicons syntax and features such as goal directed evaluation. Unicons features such as generators and limiting generators can be used along side the pattern matching features to write concise and readable code. The following code changes all lines of the format

year month day unchanged portion

in the input file to

month day year unchanged portion

in the output file.

```
procedure main()
        p := PFence() && PLen(4) -> yr && " " &&
                        PLen(4) -> mo && " " && PLen(2) -> day
        in := open("leninput", "r")
        out := open("lenoutput", "w")
        every write(out, (line := !in) ??
                             p := mo || " " || day || ", " || yr || " " )
end
```

The following code shows how the limiting generation control structure can be used to obtain the nth occurrence of a pattern. In this example the word `red` before the third occurrence of `fish` will be output

```
procedure main()
s := "One fish two fish red fish blue fish"
wrdpat := PBreak(&letters) && PSpan(&letters) -> word
p := wrdpat && PSpan(" ") && "fish"
every (s ?? p)\3
```

```
write(word)
end
```

CHAPTER 3

IMPLEMENTATION OF THE UNICON PATTERN DATA TYPE

Adding patterns as a first class data type to Unicon involved the following:

- Writing C data structures to represent patterns.
- Modify Unicon's runtime to recognize and handle these patterns
- Constructing different patterns that a user can specify
- The pattern matching engine.
- Modifications to Unicon's compiler to handle the new pattern operators

This chapter discusses all of the above. The implementation of patterns in Unicon closely follows the Ada module GNAT.Spitbol.Patterns. [4]

3.1 Internal representation of patterns

Both SNOBOL4 and SPITBOL represent patterns internally in form of a linked graph [5]. This representation is very natural for patterns as subsequents and alternants can be specified with links to different nodes. Each pattern discussed in chapter 2 is internally represented as a linked graph with one or more elements. To implement this list two different structures are used. The pattern header is represented by `struct b_pattern`

```
struct b_pattern {      /*Pattern header block */
    word title;         /*T_Pattern */
    word id;            /*Serial number of pattern */
    word stck_size;     /*max. size of stack required */
    union block * pe;   /*pointer to the first pattern element */
};
```

The individual elements of the pattern are represented by `struct b_pelem`

```
struct b_pelem {                   /* Pattern element block */
    word title;                    /* T_Pelem        */
    word pcode;                    /* Indicates Pattern type*/
    union block * pthen;           /* Pointer to succeding pointer element*/
    word index;                    /* position of pattern element in pointer
    struct descrip parameter;      /* parameter */
};
```

3.2 Adding pattern data type to Unicon

Due to Icon's optimizing compiler which performs type inferencing, the procedure for adding new types to Icon is a lengthy and complicated. This procedure is well documented in [10]. The two major types in Icon are value types and aggregate types. Aggregate types can contain other Icon types as their members. Since a pattern is composed from various sub patterns, it is considered an aggregate type by the iconx runtime. The details of adding new types to the runtime are covered in Appendix 1.

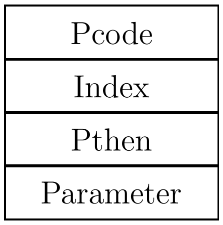


Figure 3.1: Default structure of pattern element

3.3 Structure of pattern types

Each pattern type such as PLen() consists of one or more individual pattern elements. The above figure contains the default structure for an individual pattern element. The Pcode is an enum value which identifies the type of a pattern node. It is also used by the case statement in the pattern matching engine to handle the node appropriately. Index is the serial number and its use is handled in a later section.

Pthen is a pointer to the successor node, the node to be matched if the current node successfully matches. If the current node is the last node then Pthen points to a dummy node whose Pcode PC_EOP signals pattern exit.

Consider the pattern element corresponding to the pattern PLen(). As seen in chapter 2, PLen() takes an integer argument and matches that many characters from the current cursor position. If the parameter to PLen() is an unevaluated expression then it must be evaluated and converted to an integer before use in pattern matching. It is possible at compile time to determine the type of the unevaluated expression. Since each type of unevaluated expression needs to be evaluated in a separate manner, corresponding to the type of the unevaluated expression different Pcodes are assigned to the Pcode field. This allows the pattern matching engine to perform the appropriate conversion without having to check the type of the expression every time during a pattern match. The different unevaluated expressions and the corresponding pcodes for PLen() are:

- function call: PC_Len_NF
- method call: PC_Len_NMF
- variable reference: PC_Len_NP
- field reference : PC_Len_NP

When the parameter to PLen() is not an unevaluated expression, then the Pcode assigned is PC_Len_Nat. The parameter passed is converted to an integer and stored in the parameter field of the node. Other patterns such as PAny(), PBreak(), PNotAny(), PPos(), PRpos(), PRtab(), PSpan() and PTab() are similarly constructed.

Other pattern types that correspond to alternation, PArbno or PBreakx are composed from more than one pattern element. The structure of these types is explained in the next section.

3.4 Pattern matching engine

To perform the actual match of a pattern against the subject, the run-time function `pattern_match` is called. This function is a generator function and suspends a trapped substring variable if the pattern successfully matches against the subject. The cursor position indicates the start of the portion of the subject which will be matched against the pattern. Initially this function attempts to match the pattern with the cursor position set to 0. If this function is resumed then it attempts to match the pattern with the portion of the string just after the part that was last matched. The `pattern_match` function is a wrapper over the function `internal_match` which performs the job of actual pattern matching.

The theoretical background for the pattern matching engine can be found in [1]. The pattern matching engine uses a stack to control backtracking when a match fails. The stack entries consist of a cursor value to be restored, and a node to be reestablished as the current node to attempt an appropriate rematch operation. The processing for a pattern element that has rematch alternatives pushes an appropriate entry on to the stack, and then proceeds. If a match fails at any point, the top element of the stack is popped off, resetting the cursor and the match continues by accessing the node stored with this entry. The stack entries are represented by the `stack_entry`

```
typedef struct stack_entry{
    int cursor;
    struct b_pelem *node;
}stack_entry;
```

The stack itself is a variable local to `internal_match`. This allows pattern matching to be invoked recursively. For example an unevaluated function call can contain a pattern matching expression. During pattern matching some pattern elements require a stack to be nested on the existing stack. This is implemented by the macros `Push_Region` and `Pop_Region`. `Push_Region` makes a new region on the history stack. The caller first establishes the special entry on the stack, but does not push the stack pointer. Then this call stacks a `PC_Remove_Region` node, on top of this entry, using the cursor field of the `PC_Remove_Region` entry to save the outer level stack base value, and resets the stack base to point to this `PC_Remove_Region` node. `Pop_Region` is used at the end of processing of an inner region. If the inner region left no stack entries, then all trace of it is removed. Otherwise a `PC_Restore_Region` entry is pushed to ensure proper handling of alternatives in the inner region.

The function `internal_match` consists of a giant switch statement. It switches to one of the cases based on the pcode field on the current node.

3.4.1 Implementation of alternation

The pattern `L .| R` constructs the following structure.

The A element in the figure has pcode field set to PC_Alt, the downward

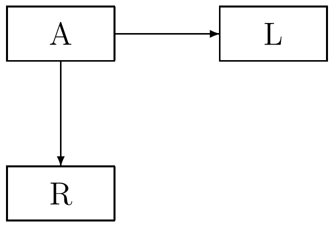


Figure 3.2: Structure of alternation element

arrow points to the contents of the parameter field. When the PC_Alt element is matched, it stacks a pointer to the leading element of R on the history stack so that on subsequent failure, a match of R is attempted. In this and the figures that follow a horizontal right pointing arrow is from the `pnext` field while a vertical downward arrow is from the `paramter` field.

3.4.2 Implementation of PArbno

There are 2 cases to the implementation of PArbno depending on the parameter `P` passed to PArbno. In the simple case the pattern matches at least one character if it succeeds and is known not to make any history stack entries. In this case PArbno is constructed with the following structure. The S (pcode

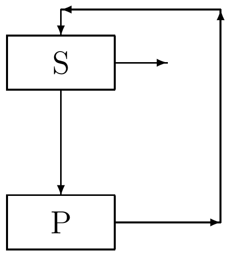


Figure 3.3: Structure of PArbno element simple case

PC_Arbno_S) node matches null stacking a pointer to the pattern P. If a subsequent failure causes P to be matched and this match succeeds, then node S gets restacked to try another instance if needed by a subsequent failure.

When the parameter `P` to PArbno can match null (or at least is not known to require a non-null string) and/or P requires pattern stack entries, the following structure is constructed for PArbno.

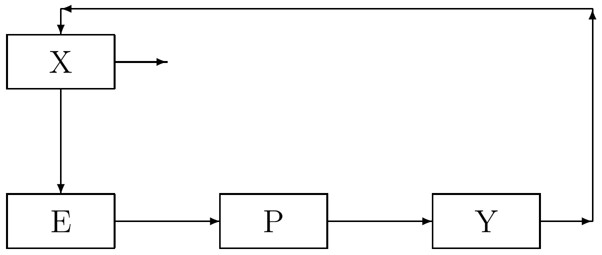


Figure 3.4: Structure of PArbno element complex case

The node X with pcode as `PC_Arbno_X` matches null, stacking a pointer to the E-P-Y structure used to match one PArbno instance. Here E is the `PC_R_Enter` node which matches null and creates two stack entries. The first is a special entry whose node field is not used at all, and whose cursor field has the initial cursor. The second entry corresponds to a standard new region action. A `PC_R_Remove` node is stacked, whose cursor field is used to store the outer stack base, and the stack base is reset to point to this `PC_R_Remove` node. Then the pattern P is matched, and it can make history stack entries in the normal manner, so now the stack looks like:

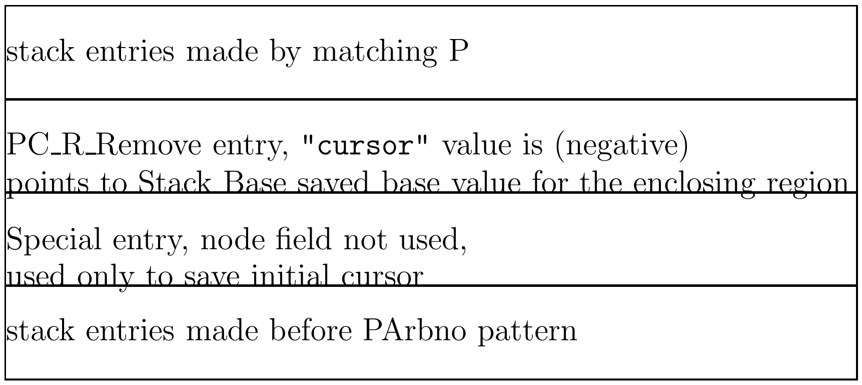

stack entries made by matching P
PC_R_Remove entry, "cursor" value is (negative)
points to Stack Base saved base value for the enclosing region
Special entry, node field not used,
used only to save initial cursor
stack entries made before PArbno pattern

Figure 3.5: Stack structure while matching P

If the match of P fails, then the `PC_R_Remove` entry is popped and it removes both itself and the special entry underneath it, restores the outer stack base, and signals failure. If the match of P succeeds, then node Y, the `PC_Arbno_Y` node, pops the inner region. There are two possibilities. If matching P left no stack entries, then all traces of the inner region can be removed. If there are stack entries, then we push an `PC_Region_Replace` stack entry whose "cursor" value is the inner stack base value, and then restore the outer stack base value, so the stack looks like:

After matching an instance of the PArbno pattern, its successor is considered. There are two cases. If the PArbno pattern matched null, then there is no point in seeking alternatives, as this would just match a whole bunch of nulls. In this the alternative node is passed over, and the pattern matching engine moves directly to its successor (i.e. the successor of the PArbno pattern). If on the other hand a non-null string was matched, we simply follow the successor to the alter-

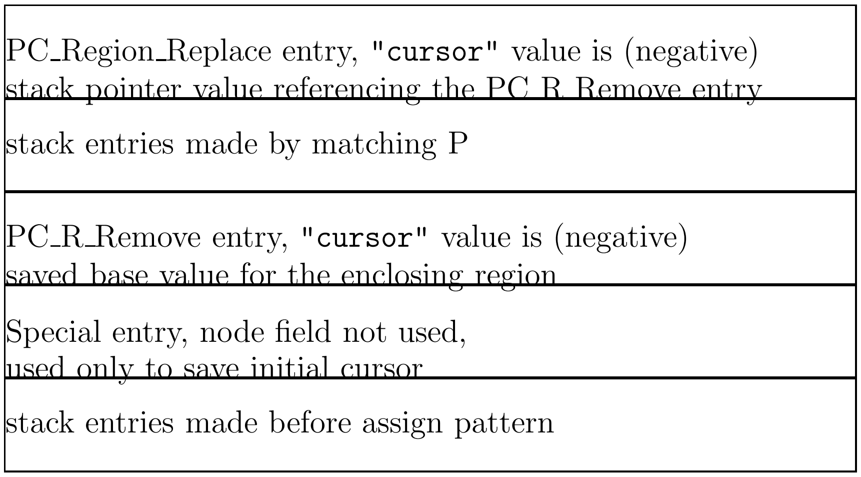

PC_Region_Replace entry, "cursor" value is (negative)
stack pointer value referencing the PC_R_Remove entry
stack entries made by matching P
PC_R_Remove entry, "cursor" value is (negative)
saved base value for the enclosing region
Special entry, node field not used,
used only to save initial cursor
stack entries made before assign pattern

Figure 3.6: Stack structure when P fails

native node, which sets up for another possible match of the PArbno pattern. The stack count (and hence the stack check) for a pattern includes only one iteration of the PArbno pattern. To make sure that multiple iterations do not overflow the stack, the PArbno node saves the stack count required by a single iteration, and the Concat function increments this to include stack entries required by any successor. The `PC_Arbno_Y` node uses this count to ensure that sufficient stack remains before proceeding after matching each new instance.

3.4.3 Implementation of unevaluated expressions

Due to Unicon's lack of a generic eval mechanism, the implementation of unevaluated expressions is a kludge. See chapter2 for the limited expressions that are actually handled. The Unicon compiler and the runtime system interact closely to implement this kludge. The Unicon compiler converts the expression into a form that can be efficiently interpreted by the runtime. Unevaluated expressions are names of functions and variables that are evaluated at runtime. The names are stored either directly or in form of a list in the parameter field of a pattern node containing the unevaluated expression. The following macros are used

- `GetVarFromNodeParameter` This macro is used when the unevaluated expression is reference to a variable or a field member of an object. In case of a simple variable the Unicon compiler stores the name of the variable in the parameter field. The macro uses the function `getvar` to resolve the name. In case of a member field reference the compiler stores a list containing the name of the object as the first element and the field referred to as the second element. The macro uses `getvar` to first resolve the object from the object name. It then performs introspection on the object to determine the location of the field.

- `GetResultFromFuncCall` This macro is used when the unevaluated expression is a function call. The Unicon compiler stores this unevaluated expression in the form of a list. The first element of the list is the name of the function or operator being invoked. The remaining elements of the list are parameters. The macro resolves the names to the actual variable and procedure locations and then uses the function `calliconproc` to perform the actual function invocation. If the function succeeds then the result of the function is stored in the variable `cresult`. If the function fails then it is as if the pattern node also failed to match.

- `GetResultFromMethodCall` This macro is used when the unevaluated expression is a method call. The Unicon compiler stores this unevaluated expression in the form of a list. The first element of the list is the name of the object and the second the method being invoked. The remaining elements are parameters. This macro is very similar to `GetResultFromFuncCall`. The only difference is that first the object name and then the method reference needs to be resolved in manner similar to resolution of field reference performed by `GetVarFromNodeParameter`.

It is not possible to model unevaluated expressions using coexpressions as coexpressions use the environment that was in effect when the coexpression was created. This prevents using the new values of immediately assigned variables in unevaluated expressions.

3.5 Modifications to Unicon's compiler

Unicon's compiler is a source code translator and translates object oriented Unicon code to plain vanilla Icon code. The operators in the new pattern matching feature are actually just functions implemented in Unicon's runtime. The Unicon compiler needed to be modified to translate the operators used in pattern matching code to the respective runtime function.

Operator	Runtime function
&&	pattern_concatenate
.|	pattern_alternate
$->$	pattern_assign_onmatch
$$	pattern_assign_immediate
.$	pattern_setcur

Table 3.1: Pattern Operators and corresponding runtime functions

Unevaluated expression are detected by the compiler as they are always enclosed in either "‘" or "“". The type of the expression can also be determined at compile time. The compiler emits function calls corresponding to the type of unevaluated expression.

Expression	Runtime function
local/global variable	pattern_unevalvar
member field	pattern_unevalvar
string procedure invocation	pattern_stringfunccall
bool procedure invocation	pattern_boolfunccall
string method invocation	pattern_stringmethodcall
bool method invocation	pattern_boolmethodcall

Table 3.2: Unevaluated expressions and corresponding runtime functions

Adding the new operators involved adding the following tokens to the lexical analyser of Unicon.

- ??
- &&
- .|

- ‘
- −>
- $$
- .$

The pattern construction and matching operators have lower precedence than the existing operators in Unicon. See appendix B for the new unicon operator table. The addition of the new operators involved changes to the unicon grammar and addition of new production rules. These rules create specific nodes in the parse tree when pattern matching operators are parsed. The code generation phase treats these nodes as a special case and emits the runtime function corresponding to the pattern operator observed.

CHAPTER 4

BENCHMARKS

To determine the potential value of pattern matching and establish wether it should be made a standard feature of the Unicon programming language, tests were conducted to compare implementations of string processing tasks using both Icon string scanning and the pattern matching facility discussed in this thesis. Programs written in SNOBOL and executed using CSNOBOL4 are also a part of the comparison for the benefit of SNOBOL programmers. The benchmarks compare the conciseness (in terms of Lines Of Code (LOC)) and efficiency of the programs. This chapter also includes the source code of the programs so that readers can draw their own conclusions about readability of programs written in these two different paradigms. The following tests were conducted

- Decomposing phone numbers
- Detecting words with double letters
- Strings of the form $A^nB^nC^n$

4.1 Testing methodology

Programs were written using both string scanning and pattern matching for each of the above tasks. The output of the two programs was compared to ensure that they were identical. Each program was run 10 times using a ruby script and the total real time in seconds is shown in the tables below. The ruby script as well as the details of the machine on which the programs were run are shown in Appendix C.

4.2 Decomposing phone numbers

For the purpose of this test phone numbers are defined by the following regular expression [9].

```
            # don't match beginning of string, number can start anywhere
(\d{3})     # area code is 3 digits (e.g. '800')
\D*         # optional separator is any number of non-digits
(\d{3})     # trunk is 3 digits (e.g. '555')
\D*         # optional separator
(\d{4})     # rest of number is 4 digits (e.g. '1212')
\D*         # optional separator
(\d*)       # extension is optional and can be any number of digits
$           # end of string
```

Some examples of phone numbers that satisfy this regular expression are

- 800-555-1212
- 800 555 1212
- 800.555.1212
- (800) 555-1212
- 1-800-555-1212
- 800-555-1212-1234
- 800-555-1212x1234
- 800-555-1212 ext. 1234
- work 1-(800) 555.1212 #1234

If the input fragment is

```
(800) 598-4668

22100 Gratiot Avenue

East Detroit, MI 48021

www.aarda.org
```

Then the program produces the following output

```
<html>

<body>

<div>(800) 598-4668 </div>

<div style="color: red;">Area Code 800</div>

<div style="color: red;">Trunk 598</div>

<div style="color: red;">Number 4668</div>

<div>22100 Gratiot Avenue </div>

<div>East Detroit, MI 48021 <div>

<div>www.aarda.org </div>

</html>

</body>
```

The input file is processed line by line. If a phone number is detected then the values of the components of phone numbers are written out. If there is no match then the line is output enclosed in a div element. So the program not only detects the phone number, it also converts an unformatted text file to a html file. This color attribute helps in visual inspection for correctness.

The following is the program written using Icon's string scanning features. It is the `if` statement that performs the actual mapping.

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

procedure main()
```

```
   in := open("HealthHotlines.txt", "r")
   out := open("PhoneParseOut.html", "w")
   write(out,"<html>");
   write(out,"<body>" )
   while line := read(in) do
   {
      line ?
      {
         write(out, "<div>",line, "</div>")
         if tab(upto(&digits)) & (areacode := digits(3)) &
         tab(upto(&digits)) & (trunk := digits(3)) &
         tab(upto(&digits)) &  (restnumber := digits(4)) &
         ((tab(upto(&digits)) & tab(many(&digits))) | "" ) &
         pos(0) then
         {
            write(out,"<div␣style=\"color:red\">","AreaCode␣=␣",
                        areacode,"</div>")
            write(out,"<div␣style=\"color:red\">","Trunk␣=␣",
                        trunk,"</div>")
            write(out,"<div␣style=\"color:red\">","Rest␣=␣",
                        restnumber,"</div>")
         }
      }
   }
   write(out,"</body>" )
   write(out,"</html>");
end
```

The following program utilizes the new pattern matching features.

```
procedure main()
    in := open("HealthHotlines.txt", "r")
    out := open("PhonePatternOut.html", "w")
    threedigits := &digits && &digits && &digits
    fourdigits := threedigits && &digits
    phone :=  threedigits -> areacode && PArb() &&
              threedigits -> trunk && PArb() &&
              fourdigits -> restnumber &&
              ((PArb() && PSpan(&digits) -> extension) .| "") &&
              PRpos(0)
    write(out,"<html>");
    write(out,"<body>" )
    while line := read(in) do
     {
         write(out, "<div>",line, "</div>")
         if line ?? phone then
         {
                write(out,"<div␣style=\"color:red\">","AreaCode␣=␣",
                                       areacode,"</div>")
                write(out,"<div␣style=\"color:red\">","Trunk␣=␣",
                                       trunk,"</div>")
                write(out,"<div␣style=\"color:red\">","Rest␣=␣",
                                       restnumber,"</div>")
         }
     }
    write(out,"</body>" )
    write(out,"</html>");
end
```

Here is the same program in SNOBOL

```
    digits = any("0123456789")
    threedigits = digits digits digits
    fourdigits = threedigits digits
    phone =  threedigits . areacode  arb  threedigits . trunk arb
             fourdigits . restnumber  ( (arb span("0123456789") . extension) | '')
             rpos(0)
    output = '<html>'
    output = '<body>'
NLINE     line = INPUT        :F(FINISH)
    output =  '<div>' line  '</div>'
    line phone         :F(NLINE)
    output = '<div␣style="color:red">' 'AreaCode␣=␣'  areacode '</div>'
    output = '<div␣style="color:red">' 'Trunk␣=␣' trunk '</div>'
    output = '<div␣style="color:red">' 'Rest␣=␣' restnumber '</div>' :(NLINE)
FINISH        output = '</body>'
    output = '</html>'
end
```

Summary of the phone benchmarks

String Scanning	
LOC	32
Time	15.09
Pattern matching	
LOC	28
Time	8.28
SNOBOL	
LOC	17
Time	8.28

Table 4.1: Phone Benchmarks

4.3 Detecting words with double letters

For the purpose of this task words were defined as a continuous sequence of letters. The program writes words in which two consecutive letters are the same to a file. Examples of such words are

- moot

- Small

- tomorrow

The input for this test was a 19Mb file.

Consider the following program utilizing string scanning

```
procedure main()
        in := open("mtent12.txt", "r") | stop("open␣failed")
        out := open("mtentscanOut.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
```

Compare it with the following program using pattern matching

```
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
```

Here is the same program in SNOBOL

```
        letters = any("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ")
        double = arbno(letters) letters $ x *x
                (span("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ") | "")
NLINE   line = INPUT                    :F(END)
MATL    line double . word  = ’’        :F(NLINE)
        output = word                   :(MATL)
end
```

Summary of the double letter words benchmarks

String Scanning	
LOC	13
Time	112.054694
Pattern matching	
LOC	7
Time	45.489382
SNOBOL	
LOC	6
Time	637.151224

Table 4.2: Double Letter Words Benchmarks

This pattern uses both immediate assignment as well as unevaluated expressions. Both these operations are among the most inefficient for pattern matching as they involve doing string comparisons with all the names in scope at that time. The next chapter presents ways in which this can be improved.

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

This is a common example of a language that cannot be parsed by a CFG grammar. The Icon program that recognizes strings of this form is from [8]. The Unicon program utilizing string scanning is based on an example in [6].

The Icon string scanning example

```
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 pattern matching example

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

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

A bug in the current CSNOBOL4 program prevents us from presenting an equivalent SNOBOL program.

Summary of the $A^nB^nC^n$ benchmarks

String Scanning	
LOC	11
Time	14.12
Pattern matching	
LOC	11
Time	6.34

Table 4.3: $A^nB^nC^n$ Benchmarks

CHAPTER 5

FUTURE WORK AND CONCLUSION

The pattern matching features implemented in Unicon are still in a pre-release state. This chapter discusses improvements to the pattern matching features, provides an example of how pattern matching and string scanning can cooperate and ends with a summary of the whole project

5.1 Future work

The future work can be broadly divided into two categories.

- Improving the power of pattern matching.
- Improving the efficiency of pattern matching.

5.1.1 Improving the power of pattern matching

A major difference between SNOBOL and Unicon is the lack of a runtime eval mechanism in Unicon. The pattern matching code in SNOBOL often relies on the dynamic evaluation mechanism. For example the pattern for $A^nB^nC^n$ in SNOBOL

```
ANBNCN = POS(0) SPAN('A') @A SPAN('B') @B SPAN('C')
               @C RPOS(0) *((EQ(B - A,A) (EQ(C - B,A) )
```

is more concise and clear than the same in Unicon

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

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

The addition of a generic eval mechanism would remove the arbitary limitations that have been imposed on the unevaluated expressions allowed in patterns. Addition of eval to Unicon will vastly improve the power of the existing pattern matching implementation.

A long standing feature request in Unicon is that the keywords `&output` and `&input` be variables which can be assigned to. This will allow redirection of input and output at runtime in Unicon code. Most code samples contian the following code fragment

```
out := &output
```

Such redundant code will be eliminated if `&output` is a Unicon variable.

5.1.2 Improving the efficiency of pattern matching

It is not possible to resolve variable references for unevaluated expressions and assignment target variables in patterns during pattern construction as patterns are a full fledged data type and can be stored in global variables, return from functions and passed as paramaters functions. In this case the variable references resolved at pattern construction have no valid semantics. Hence instead of the variable reference the pattern stores a string representation of the variable.

The pattern matching implementation handles features such as variable assignment in patterns and unevaluated expressions by searching using string comparison through all the identifiers in scope at that time. This search is performed every time during the pattern match the operation has to be performed. There are two possible methods in consideration currently to improve this.

Instead of performing string comparisons for all the identifiers in scope the identifiers should be kept in hash tables. Then variable resolution will be a quick hash table lookup compared to the many string comparisons that are being performed now. A major benefit of this approach is that such an identifier table will be useful when an eval facility is implemented.

Since the environment does not change in a procedure the variable name resolution should happen only once in each procedure. This can be achieved by creating a table in which a copy of a pattern with all its references resolved can be stored. This table is part of the structure used to represent procedure in Unicon. The first time a pattern A is passed as parameter to the `pattern_match` function a new pattern is constructed which is a copy of pattern A but with the variable references resolved. This copy is stored in the table and associated with the serial number of pattern A. The next time pattern A is passed to `pattern_match` the copy is used.

5.2 Mixing pattern matching and string scanning

Pattern matching and string scanning are two orthogonal ways of performing string processing. Integrating them more closely would pose implementation difficulties nad possibly lead to cognitive dissonance. For example it might be tempting to consider constructs such as

```
 pattern := &PArbno(&digits) && tab(upto(&digits))
```

it must be noted that there is no scanning environment in place during pattern construction by default. If there is a scanning enviroment in effect then the above pattern has the same effect as

```
res := tab(upto(&digits))
 pattern := &PArbno(&digits) && res
```

Since arbitrary Unicon code can be executed during string scanning it is possible to mix pattern matching with string scanning code. For example here is the modified string scanning code for detecting words with double letters

```
procedure main()
    in := open("mtent12.txt", "r") | stop("open␣failed")
    out := open("mtentscanOut.txt", "w")
    double := (PLen(1) $$ 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 Conclusion

The newly added pattern data type provides some benefits in text processing tasks. It allows Unicon programmers to write more succinct and efficient code to perform text processing. Since the new pattern facility closely follows the implementation of patterns in SNOBOL it also provides a path for SNOBOL programmers to migrate their legacy SNOBOL code to Unicon. Translating SNOBOL pattern matching code to Unicon patterns has been simplified because of the nearly one to one matching between SNOBOL operators and functions and Unicon operators and functions. The lack of eval in Unicon prevents the pattern data type from exhibiting all the power of SNOBOL patterns. The real challenge for Unicon patterns is their adoption in the Unicon programmer community and use in real world applications.

APPENDICES

APPENDIX A

ADDING THE PATTERN DATA TYPE TO THE ICONX RUNTIME

Due to Icon's optimizing compiler which performs type inferencing, the procedure for adding new types to Icon is a lengthy and complicated. This Appendix is a detailed line by line explanation of changes that were made to iconx while adding the new pattern type.

The following lines were added to `/common/typespec.txt`

```
pattern{p}:  aggregate(pat_elem)
        return block_pointer
```

Since there are two new structures that are added to Icon's runtime type codes and descriptor codes corresponding to the pattern header structure and pattern element structure have to be added to `rmacros.h`. The following typecodes were added :

```
#define T_Pattern 26    /*  pattern header */
#define T_Pelem   27    /* pattern element */
```

The following descriptor codes were added:

```
#define D_Pattern       (T_Pattern   | D_Typecode | F_Ptr)
#define D_Pelem         (T_Pelem     | D_Typecode | F_Ptr)
```

When a new type is added to Unicon the garbage collector also needs information about the structures added. This information is provided in the following arrays : `bsizes, firstd, firstp, ptrno, blkname`

`bsizes` contains the block size of the newly added structure. The following additions were made to `bsizes`:

```
sizeof(struct b_pattern), /* T_Pattern (26), pattern block */
sizeof(struct b_pelem),   /* T_Pattern (27), pattern element */
```

`firstd` contains the offset (in bytes) to the first descriptor in the block. The following additions were made to `firstd`:

```
0,                /* T_Pattern (26), pattern block */
4*WordSize,       /* T_Pelem (27), pattern element */
```

`firstp` contains the offset of the first pointer in blocks. The following additions were made to `firstp`:

```
3*WordSize, /* T_Pattern(26) pattern block*/
2*WordSize, /* T_Pelem(26) pattern element block*/
```

`ptrno` contains the number of pointers in blocks. The following additions were made to `ptrno`:

```
1, /* T_Pattern (26), pattern block */
1, /* T_Pelem (26), pattern element block */
```

`blkname` contains the block names used by debugging functions. The following additions were made to `blkname`:

```
"pattern",           /* T_Pattern (26) */
"pattern␣element",   /* T_Pelem (27) */
```

New allocation routines must be written for the newly added type. These belong to the file `ralc.r`.

APPENDIX B

MODIFIED OPERATOR TABLE

The newly added pattern construction and pattern matching operators have been inserted in the icon operator table given below.

```
( expr )

{ expr1; expr2;  }

[ expr1, expr2,  ]

expr. f

expr1 [ expr2, expr3,  ]

expr1 [ expr2 : expr3 ]

expr1 [ expr2 +: expr3 ]

expr1 [ expr2 : expr3 ]

expr ( expr1, expr2,  )

expr { expr1, expr2,  }

‘expr‘

‘‘expr‘‘




not expr
```

```
| expr

! expr

* expr

+ expr

 expr

. expr

/ expr

\ expr

= expr

? expr

~ expr

@ expr

^ expr

$$ expr


expr1 \ expr2

expr1 @ expr2

expr1 ! expr2


expr1 ^ expr2 (right associative)
```

```
expr1 * expr2

expr1 / expr2

expr1 % expr2

expr1 ** expr2


expr1 + expr2

expr1  expr2

expr1 ++ expr2

expr1   expr2


expr1 || expr2

expr1 ||| expr2


expr1 < expr2

expr1 <= expr2

expr1 = expr2

expr1 >= expr2

expr1 > expr2

expr1 ~= expr2

expr1 << expr2

expr1 <<= expr2
```

```
expr1 == expr2

expr1 >>= expr2

expr1 >> expr2

expr1 ~== expr2

expr1 === expr2

expr1 ~=== expr2


expr1 .$ expr2

expr1 -> expr2


expr1 && expr2

expr1 | expr2


expr1 .| expr2


expr1 to expr2 by expr3


expr1 ?? expr2


expr1 := expr2 (right associative)

expr1 < expr2 (right associative)
```

```
expr1 :=: expr2 (right associative)

expr1 <> expr2 (right associative)

expr1 op:= expr2 (right associative)


expr1 ? expr2


expr1 & expr2


break expr

case expr of { expr1 : expr2; expr3 : expr4;  }

create expr

every expr1 do expr2

fail

if expr1 then expr2 else expr3

next

repeat expr

return expr

suspend expr1 do expr2

until expr1 do expr2

while expr1 do expr2
```

APPENDIX C

BENCHMARK DETAILS

The ruby script used for the benchmarks

```
require "benchmark"
include Benchmark
bm(12) do |test|
  test.report("Phone␣parse␣String␣scanning:") do
    for i in 1 ... 10
      `./PhoneParse`
    end
  end
  test.report("Phone␣parse␣pattern␣matching:") do
    for i in 1 ... 10
      `./PhPpattern`
    end
  end
  test.report("Double␣letter␣string␣scanning:") do
    for i in 1 ... 10
      `./doubleletter`
    end
  end
  test.report("Double␣letter␣pattern␣matching:") do
    for i in 1 ... 10
      `./doublepattern`
    end
  end
  test.report("ABC␣string␣scanning:") do
    for i in 1 ... 10
      `./ABC <ABCinput >ABCoutput`
    end
  end
  test.report("ABC␣pattern␣matching:") do
    for i in 1 ... 10
      `./ABCss <ABCinput >ABCoutput`
    end
  end

end
```

Operating system on test machine (output of uname -a)

```
Linux 2.6.10-5-386 #1  i686 GNU/Linux
```

CPU information

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 13
model name      : Intel(R) Pentium(R) M processor 1.86GHz
stepping        : 8
cpu MHz         : 797.892
cache size      : 2048 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 2
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep
                  mtrr pge mca cmov pat clflush dts acpi mmx
                  fxsr sse sse2 ss tm pbe nx est tm2
bogomips        : 1576.09
```

REFERENCES

[1] Alfred V. Aho. Nested stack automata. *J. ACM*, 16(3):383–406, 1969.

[2] Philip Budne. Csnobol4 release 1.0. *http://www.snobol4.org/csnobol4/*, 2004.

[3] R. B. K. Dewar and A. P. McCann. MACRO SPITBOL – A SNOBOL4 compiler. *Software – Practice and Experience*, 7(1):95–113, January&February 1977.

[4] Robert K. Dewar. Gnat spitbol patterns. *http://www.iuma.ulpgc.es/users/jmiranda/gnat-rts/rts/g-spipat*$_{adb.htm}$.

[5] James F. Gimpel. A theory of discrete patterns and their implementation in SNOBOL4. *Communications of the ACM*, 16(2):91–100, February 1973.

[6] R. E. Griswold. *String and List Processing in Snobol 4*. Prentice Hall, Englewood Cliffs, 1975.

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

[8] Ralph E. Griswold and Madge T. Griswold. *The Icon programming language*. Peer-to-Peer Communications, San Jose, CA, USA, third edition, 1997.

[9] Mark Piligrim. *Dive Into Python*. Apress, New Jersey,July 2004.

[10] Kenneth Walker. The run-time implementation language for icon. *Department of Computer Science, the University of Arizona*, http://www.cs.arizona.edu/icon/ftp/doc/ipd261.pdf.