String Content Extraction Language (SCOEL)

Goal Directed String Scanning

by John Goettsche
New Mexico Tech

Table of Contents

Table of Contents....2
Introduction....4
String Analysis....5
  Information Extraction....5
  Scanning and Iterating through a String of Characters....6
    Singular Sentinel Iteration....6
    Multiple Sentinel or Pattern Iteration....9
    Unconditional Iteration....10
    Unicon String Scanning....11
    Unicon Pattern Matching....11
    SCOEL Scanning....12
    Scanning Functions and Objects:....13
  Backtracking....14
    Unicon....14
    Prolog....14
  Goal Directed Extraction....15
    Pattern Definitions....17
    Additional Matching Objects available in Patterns only:....18
    Structural Matching Objects available in Patterns only:....20
  Anchoring....22
  Back Scanning....23
Strings and Lists....24
    Collection Methods:....24
    Iteration Methods:....25
Related Research....25
  Goal Directed String Scanning....25
  Website Information Extraction with Goal Directed String Scanning....25
  Finding Partitioned Tokens with SCOEL....25
  Evaluating Context of Extracted Information with SCOEL....25
  Extracting Information from HTML Tables with SCOEL....25
  Other things suggested by others…....26
Bibliography....27

Table of Figures

Figure 1: Scanning a single element based upon a set....7
Figure 2: Scanning a number of elements based upon a set....7
Figure 3: Scanning any number of elements based upon a set....8
Figure 4: Scanning any number of characters until an element matches a member of a set....8
Figure 5: *S*canning elements which match another string of elements....9
Figure 6: Scanning the next element....10
Figure 7: Prolog Predicate Internal Operations....14
Figure 8: SCOEL Matching object and its internal workings....15
Figure 9: Back Matching Object....17

Figure 9: Back Matching Object...17
Figure 10: Pattern Structure..18
Figure 11: Pattern Scan Matching Function..19
Figure 12: Fence Matching Object..20
Figure 13: Or Structural Matching Function..21
Figure 14: Parenthesis Structural Match Function...22
Figure 15: Matching Object..24

Introduction

The problem of extracting information from a string of characters is a problem as old as computers themselves. Computing technology has greatly enhanced the process by increasing its speed, complexity, specificity, and accuracy. It has taken a once tedious task of reading many articles to analyze data and is automating the process. Whether someone is looking to identify names and addresses or determining how many times a company has been referenced in newspaper articles, software is being developed to handle the task. Programming languages are being developed to handle the string analysis in a variety of ways.

The String Content Extraction Language (SCOEL) is designed to allow the user to extract complex distributions of information in strings of data. It expands upon finding tokens, like in parsing problems, to creating patterns to find information which may be spread through out the string. An example would be extracting the digits of a phone number with hyphens and parenthesis, which are typically used to make them more readable for human readers, and are found somewhere in a string of text.

SCOEL is an object oriented language with goal directed evaluation. It is derived from the Unicon programming language by Dr. Clinton Jeffery, et al., also an object oriented programming language with goal directed evaluation.

The implementation of SCOEL will be done by working from the Unicon13 compiler and virtual machine and moving some features to class definitions and objects. It will also require some changes to the lexical, syntactic and semantic structure of the language.

The objective of SCOEL was to make a language which is more strictly object oriented while combining two of Unicon’s string analysis paradigms, Icon’s string scanning environment and SNOBOL4’s pattern matching system, into a single system which uses a common set of built-in matching functions. This requires a review of Unicon's data storage structures, the syntax and semantics of the language, and the main concepts of goal-directed evaluation and backtracking.

String Analysis

The string analysis capabilities of Unicon 13 are in its string scanning environment and its pattern matching system with regular expressions and SNOBOL4 type functions. Both provide a robust method of finding tokens to extract from a particular string.[1], [2], [3] SCOEL will be working with a common primitive set of functions for string and pattern matching. When extracting information from a string or the information being searched may be spread through out the stream and its relevance might be determineable without contextual information or some data may be broken up so it is easier for the human reader to understand. This can lead to more complex searches and normally require multiple searches for patterns and tokens.[4], [5]

Information Extraction

In the area of Information Extraction, the user is attempting to extract information from a stream of data, which can be from any source: a large data base, stored research papers, the internet, or anything else which contains textural data. Searches are generally performed with some keywords or patterns which are searched.[6], [7]

How the information is stored affects how the search is to be performed. If the information is fully structured, it is in a predictable order such as XML documents and CSV (comma separated value) files., then the search algorithm is relatively simple. Semi-structured information, such as a short hand natural language with a fixed grammatical rules or older records which may have evolved over the years, may require more complex systems or pattern definitions to extract the data. As changes in data storage conventions change over time, so do the practices to extract that data changes. Unstructured information, such as the text from lengthy reports or legal documents, require very complex pattern matching schema.[5]

Should a project require batch processing such as billing changes for changes in the contracts of thousands of customers, the system will have to extract information on customer accounts and make the desired changes for each customer. This will requir, extracting all the necessary information from the data stream and relaying the information in a format acceptable to the software handling the database. The data stream most likely be fairly structured and a pattern for each piece of information might be straight forward, some pieces of information may not be consistent depending who entered the data or the software requirements from data entry sources. Some data points may be stored in seperate parts, like a phone number with hyphens and parenthesis. As the project requires the processing of thousands of customer changes, the system needs to be as automated as possible, so the information extraction needs to be complete.

Scanning and Iterating through a String of Characters

There are three schema of iteration through a string of characters. There is the single sentinel method, where each character in the string is tested individually to see if it matches. The multi sentinal or pattern method, where the string or pattern is tested to see if an ordered set of characters or pattern is found. Finally the unrestricted method where a set number of characters or all remaining characters are accepted. Each of these iteration types are further described as follows:

Singular Sentinel Iteration

Sentinel Iteration is where each element of the scan is tested for a sentinel value in order to proceed. Sometimes it is looking for a specific value to advance, sometimes it will advance until the sentinel value if found. Here are some examples of sentinel iteration.

1. The next character if it is in or is not in a set of characters. In this example the position is at index number 2, it checks to see if one of the letters in the set in the box is the next element in the string. If finds an ‘`a`’ which is in the set and advances to index number 3.

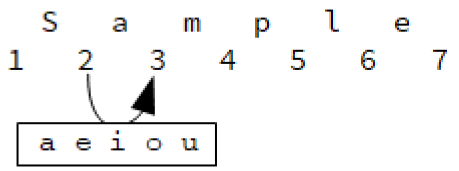

Figure 1: Scanning a single element based upon a set

Unicon string scanning can accomplish this task with the following lines of code:

```
vowels := ‘aeiou’
str := “Sample”
str ? {
   tab(2)
   result := tab(any(vowels))
   }
```

2. The next number of characters if they are in or not in a set of characters. In this example the position is located in index 2 and it is looking for letters over the next three elements. It finds ‘`amp`’ and the position is advanced to index 5.

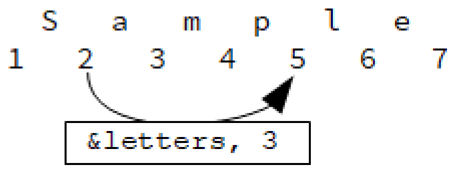

Figure 2: Scanning a number of elements based upon a set

Unicon string scanning can accomplish this task with the following lines of code:

```
str := “Sample”
str ? {
   tab(2)
   result := “”
   while *result <= 3 do {
      result +:= tab(any(&letters))
   }
}
```

3. An undetermined number of characters if they are in or not in a set of characters. In this example the position is located in index 2 and it is looking for any letters. It finds 'ample' and the position is advanced to index 7. The space between indexes 7 and 8 is not a letter therefore it stops at index 7.

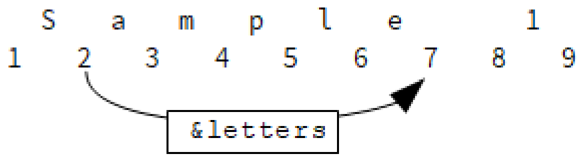

Figure 3: Scanning any number of elements based upon a set

Unicon string scanning can accomplish this task with the following lines of code:

```
str := "Sample 1"
str ? {
   tab(2)
   result +:= tab(many(&letters))
}
```

4. An undetermined number of characters until a character is in or not in a set of characters. In this example the position is at index 3 and it is searching for a vowel to stop at. It advances up to the 'e' and the position ends at index 6 just before the 'e'.

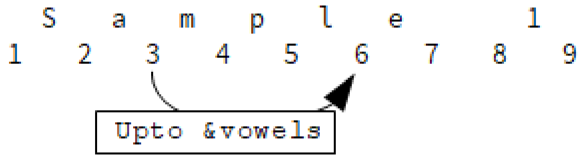

Figure 4: Scanning any number of characters until an element matches a member of a set

Unicon string scanning can accomplish this task with the following lines of code:

```
vowels := 'aeiou'
str := "Sample 1"
str ? {
   tab(3)
   result +:= tab(upto(vowels))
}
```

Multiple Sentinel or Pattern Iteration

Pattern Iteration it is looking for a complex collection of elements which match a pattern definition. Pattern Iteration differs from Single Sentinel iteration in that it is looking for a string of characters to match its definition, while Single Sentinel Iteration is only testing against a single character. This pattern definition can vary in size from a few characters to a very long string of characters with certain criteria defined to match. Here are some examples of Pattern Iteration.

1. The characters which match an ordered set of characters. In this example the position is at the default index 1 and it is looks for the string of characters 'ample'. As it passes through the characters in the subject string it finds an a and then checks to see if all the following elements match the pattern provided. When it does it assigns the value to result. Had there been elements out of order or had some other character not in the pattern it fails and does not make the assignment to result. In this example it finds each of the letters in the pattern and the position is advanced to where the pattern ends at index.

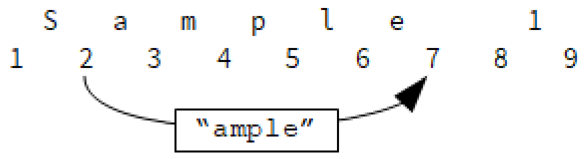

*Figure 5: S*canning elements which match another string of elements

Unicon pattern matching can accomplish this task with the following lines of code:

```
str := "Sample 1"
pat := "ample" -> result
str ?? pat
```

2. The characters which match a predefined pattern.
3. The characters which match an ordered set of characters or a predefined pattern a certain number of times.

4. The characters which match an ordered set of characters or a predefined pattern an indeterminate number of times.
5. The characters before an ordered set of characters or a predefined pattern is found. In this case the system must identify the beginning of the set or pattern and store that location, then it has to test to see if the following elements make the desired set or pattern. If it does it has to go back to the saved index at the beginning of the set or pattern.
6. There is also the issue of parenthesis, Brackets and Quotation marks, which is a pattern of a last found opening character including all intermediate characters until you find your first closing character. You are testing to see if the opening character, say an opening parenthesis is following some where later in the string with a matching closing character.

Unconditional Iteration

1. The next character. In this example the position is at index number 2 and it accepts what ever is located in the next element of the string and advances to index number 3.

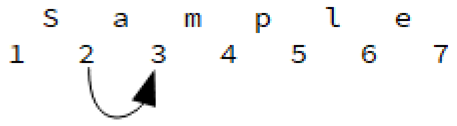

Figure 6: Scanning the next element

2. The next number of characters.
3. All the characters from the current index to the a specified index.
4. All remaining characters.

All of the above is about iterating through a string of characters. What is done with these sub-strings has not been discussed, which will be discussed in the next section.

Unicon String Scanning

The string scanning environment in Unicon was inherited from the Icon programming language. In Unicon a string is put into the string scanning environment by typing:

```
str ? <expressions>
```

The string scanning environment sets the &subject to 'str' and the &pos, or index location, is set at the beginning of the string.

There is a set of string scanning functions which are available for the programmer inside this environment, some are: any(<CSet/string>), bal(), find(<CSet/string>), many(<CSet/string>), move(<int>), tab(<int>), upto(<CSet/string>), and can be used as follows:

```
str := "My phone number is 555-1234"
str ? {
     tab(upto(&digits))
     number := tab(many(&digits))
     tab(upto(&digits))
     number ||:= tab(many(&digits))
}
```

In this case all the characters in the sentence would be passed over until a digit was found, then number would be assigned '5551234'. Then it exits the string scanning environment.

The Unicon string scanning environment is declarative and is designed to progress through the string much like an iterator can move forward and backwards through a list. If the user wants to go backwards through the test it can by using negative values in move() or tab(), or some other combination of functions.

Unicon Pattern Matching

The pattern matching system in Unicon is similar to the pattern matching system in SNOBOL4. A pattern is a primitive data type which can be defined and assigned to a variable. If you wanted to define a pattern to retrieve a phone number from a string you would type:

```
pat := Span(&digits) || '-' || Span(&digits)
```

In this case ‘pat’ is assigned the pattern definition on the right. To find that pattern in ‘str’ above, you would enter:

```
str ?? pat
```

Therefor:

```
number := str ?? pat
```

Results in ‘number’ being assigned ‘555-1234’. If the user wants to perform the match anchored at the current location within the string scanning environment, the ‘=<pattern>’ operator is used.

```
Str ? {
     tab(upto(&digits))
     write(=pat)
}
```

In this example the tab-upto function would move the cursor location forward until a digit is found. Then it would write the results of the pattern match, ‘555-1234’.

SCOEL Scanning

The SCOEL scanning objects which elements are collected is a component of a Matching object which decides what do with the data.

The first option is the simplest, there is no need to keep track of anything other than update the position to the appropriate index number.

The second option can be easily handled by assigning the result to a variable, whether the operation is the result of string scanning tool, a scan or a match. An example would be as follows:

```
foo := str.many(&letters)
```

The variable foo is assigned the result of the string function many() using &letters as the argument. It would start at the current index of the string and would return a sub-string of letters until it reaches a character which is not a letter.

The third option is placing the sub-string on a stack to be combined later. This is what happens when the scan() or match() string functions are used. Each sub-string created by each element in the pattern definition is stored on a stack. When the pattern is completed and it is a success, then each sub-string is popped off the stack to create a new string.

Scanning Functions and Objects:

any() returns the next element, advances the pos, and succeeds.

any(<set>) returns the next element, advances the pos, and succeeds if it matches an element of the set.

many() returns all remaining elements, advances the pos to the end, and succeeds.

many(<int>) returns a string of n elements matching the integer argument, advances the pos, and succeeds if there are enough elements remaining.

many(<set>) returns a string of elements as long as the set contains them, advances the pos, and succeeds if elements are found.

many(<str>) returns a string of elements which match the string argument, advances the pos, and succeeds if the string is found.

many(<pat>) returns a string of elements which match the pattern definition, advances the pos, and succeeds if the pattern is found.

many(<set, int>) returns a string of a certain number of elements which all are contained in a set, advances the pos, and succeeds if a string of elements of an appropriate length are found.

many(<str, int>) returns a string of elements that match the string argument a specified number of times, advances the pos, and succeeds if the repeated string is found the specified number of times.

many(<pat, int>) returns a string of elements that match the pattern definition a specified number of times, advances the pos, and succeeds if the repeated string matches the pattern definition the specified number of times.

upto(<int>) returns a string of elements upto the specified index, advances the pos up to the specified index, and succeeds if any elements are found.

upto(<set>) returns a string of elements upto an element which is a member of the set, advances the pos to the first element of the set is found, and succeeds if members of the set are found.

upto(<str>) returns a string of elements upto a string of elements match the string argument, advances the pos to where the string argument begins, and succeeds if the string argument is found.

upto(<pat>) returns a string of elements upto a string of elements match the pattern definition, advances the pos to where the pattern definition, and succeeds if the pattern definition is found.

Backtracking

Unicon

Unicon's matching functions, `tab()` and `move()`, are generators which suspend results. Backtracking occurs when it fails to get a match from the scanning instructions in their perameters by reseting the &pos in to the scanning environment. This allows alternative matches to be tried from the same position.[1]

```
str ? {
     tab(many(&letters))
     tab(many(&digits))
}
```

In the above example, `str` is a string and a string scanning environment is created with two `tab()` matching functions. If at the beginning of the string there are a collection of letters, then it would succeed and the `&pos` would be advanced to the end of the letters. On the other hand if there weren't any letters at the beginning of the string, it would fail and the `&pos` would not be advaned, had there been any advancement of the `&pos` during the scan for the letters `&pos` would be reset to its orignal location. Then it would perform the next `tab()` matching function where it would look for digits.

Prolog

In the Prolog language, facts and rules are stored as predicates. Backtracking is used in the evaluation of these predicates. As the diagram below shows:

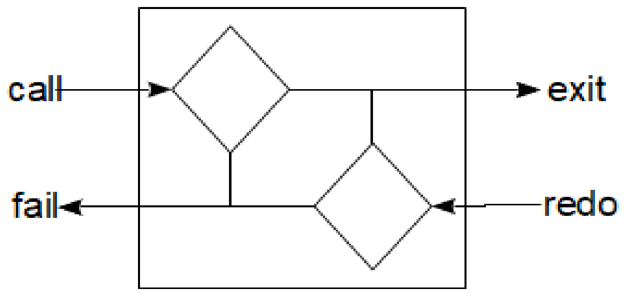

Figure 7: Prolog Predicate Internal Operations

A call is made to a rule or predicate and in the upper left diamond it evaluates its goal. If it succeeds then it exits to the next predicate; if it fails, it returns to the previous predicate with a fail message. Should Prolog reach the end of the set of rules or predicates, it returns back along the executing the instructions for the redo. This allows additional evaluation and commands to be performed as Prolog backtracks past the predicate.[8]

Goal Directed Extraction

SCOEL's string analysis system was designed to take advantage of the backtracking structure of Prolog while having a similar functionality of Unicon's string scanning. This is achieved by creating a linked list of Matching objects where each can contain one or tow Scan objects which determines which elements are under consideration for Matching. The first Scan object, (shown in the top half of the diagram below), is encountered as the string analysis system enters the Matching object. Later, once all the Matching objects have been completed or a scan fails, it backtracks to perform any back scan objects, if there are any, and resolve the results. Back scanning will be described in more detail later.

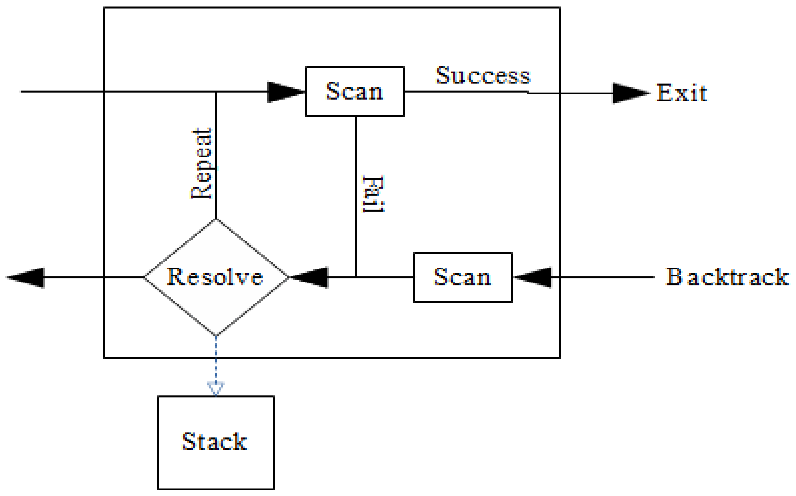

Figure 8: SCOEL Matching object and its internal workings

The internal structure of a Matching object consists of either one or two Scan objects which determines which sub-string is to be identified for the resolution process. If the scan succeeds, the extraction process exits the Matching object and proceeds to the next Matching object. If the scan fails, it begins the backtracking process with a &fail message.

The Matching objects which are used to determine whether the information collected in the scan are stored on the stack, counted or disregarded. The available Matching objects are described as follows:

collate(<scan()>[, scan()]) stores the elements collected from a successful scan to the stack and combines them into a list of tokens.
fetch(<scan()>[, scan()]) anchored match described further below, stores the elements collected from a successful scan to the stack.
Fetch(<scan()>[, scan()]) unanchored match described further below, stores the elements collected from a successful scan to the stack.
pass(<scan()>[, scan()]) anchored match described further below, does not store the elements collected from a successful scan to the stack.
Pass(<scan()>[, scan()]) unanchored match described further below, does not store the elements collected from a successful scan to the stack.
tabulate(<scan()>[, scan()]) returns the number of times it can find the scan successfully.

While back tracking each Matching object determines whether to save the results of the scan, discard them, tabulate them or collate each individual sub-string, in some conditions which will be discussed later it may decide to either fail and repeat, fail and proceed with backtracking, or terminate the operation. It it decides to save the results, they are pushed on to a stack for combining into a final String.

Eventually the chain of Matching objects is capped with a Back Matching object. If one is not included in a pattern definition or a string matching function is used, then a Back Matching object with a &success argument is automatically used. But, when defining a pattern definition if a Back Matching object is added to the end of the definition, it can hold either a &success, &fail or &terminate argument for backtracking. While backtracking if a &success message is passed back to the previous match object, its Back Scan object will be performed. If the Back Scan succeeds, then substrings of the scans will be stored to the stack. When a &fail or &terminate is sent back to the previous Matching object, the Back Scan object is skipped and nothing is stored to the stack.

back(&fail/&success/&terminate) Sets up the necessary components for backtracking and sends a message to identify the status of the string analysis.

Below is a diagram of the Back Matching object, the End box is where the object sets up the required components to track the back tracking process.

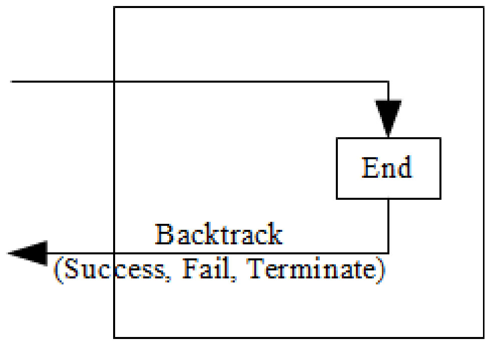

Figure 9: Back Matching Object

Pattern Definitions

When complex sets of instruction are required, those instructions can be combined into a pattern definition. A pattern definition is a series of Matching instructions with their respective scanning operations.

```
pat := pass(many(&letters)) fetch(many(&digits, 3))
fetch(any(&letters))
```

In the above example a pattern definition called `pat` has two Matching instructions; the first passes a set of letters and fetches the next three digits followed by a single letter. When SCOEL comes across this instruction it creates four Matching objects, three being for the `pass,` the two `fetchs` and one for the end of the pattern definition.

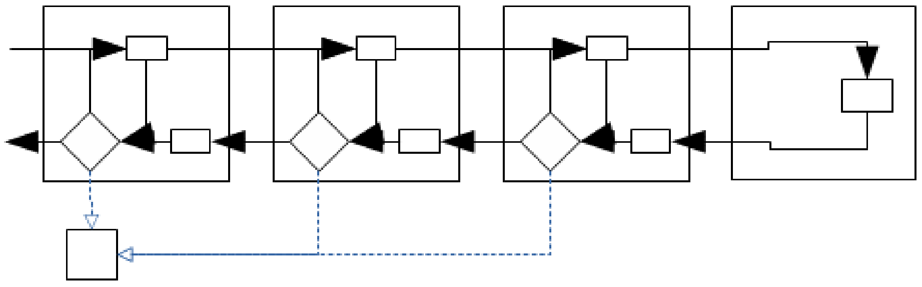

Figure 10: Pattern Structure

It is not necessary to include a Back object in the pattern definition. If one does not appear, it is assumed to it will pass back a success message during backtracking. A Back object can be given either of the &success, &fail, or &terminate keywords.

back(&success|&fail|&terminate) signifies the end of a pattern definition and what message to send when backtracking begins.

The following code illustrates how a fail message can be sent from the Back object:

```
pat := pass(many(&letters)) fetch(many(&digits, 3))
fetch(any(&letters)) back(&fail)
```

Additional Matching Objects available in Patterns only:

When using a pattern argument in a Scan object

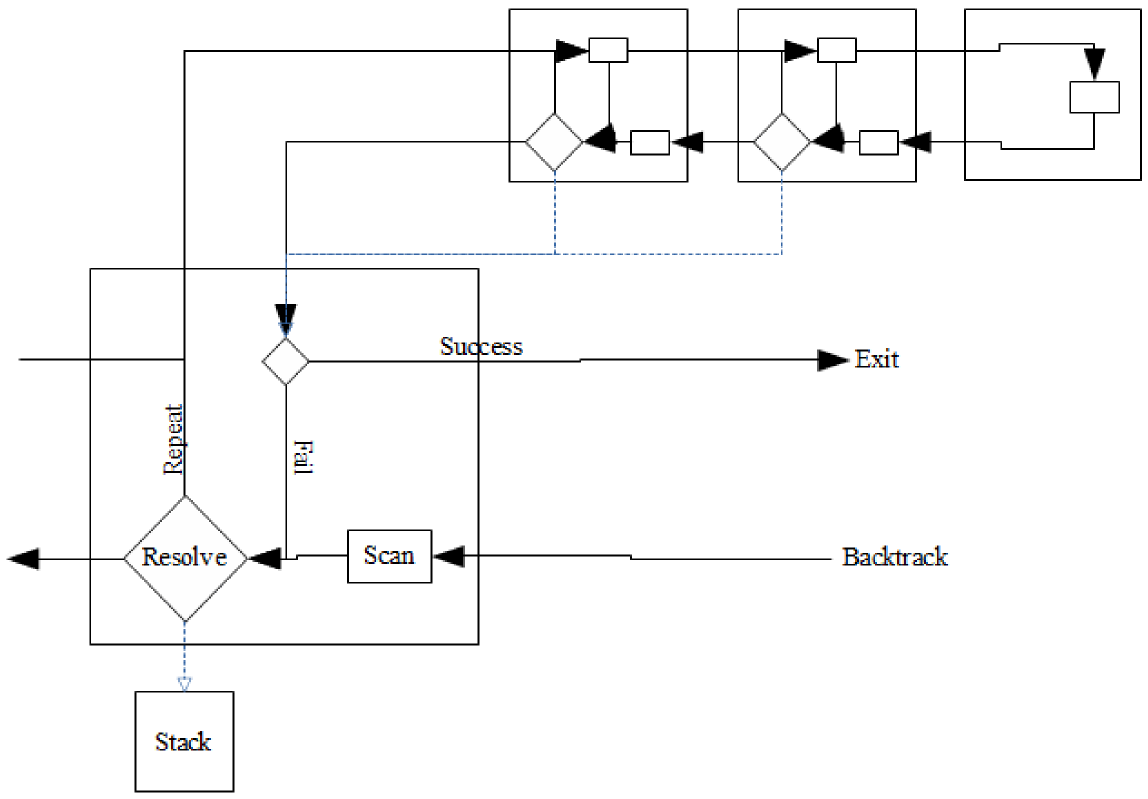

Figure 11: Pattern Scan Matching Function

Should it be necessary to change the message from either success or fail, the Fence object can be used to change the backtrack message to either success, fail or terminate. If SCOEL runs across one these objects during backtracking, it will change the message, but during the forward matching process it will do nothing.

fence(&fail/&success/&terminate) changes the backtracking message at this point if it is either &success or &fail to the message selected.

The following code is an example how this code could be used:

```
pat := pass(many(&letters)) fence(&fail) fetch(many(&digits, 3))
fetch(any(&letters))

str := fetch(subject, pat)
```

In this example, the backtracking process would start by receiving a success message if all pass and fetch objects were successful in their scans. Going back from the end, the first Match object would put a letter on the stack and return a success message. The second Match object would put three digits on the sack. The fence would receive a success message and would send a fail message back. This will result in SCOEL signaling that the string scan failed and str would be assigned to &null

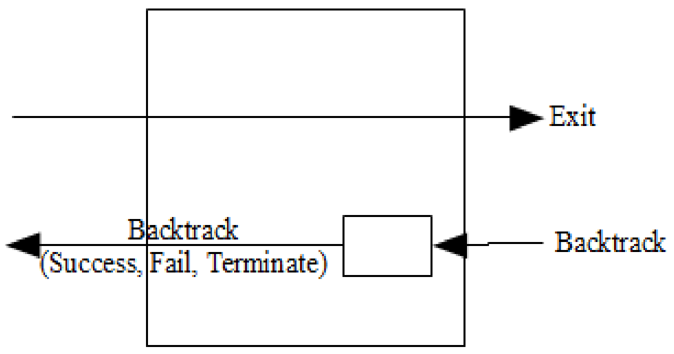

Figure 12: Fence Matching Object

Structural Matching Objects available in Patterns only:

SCOEL uses implicit concatenation. Therefor, it is not necessary to create a Matching class to handle concatenation. By placing a match operation followed by another match operation in a pattern definition SCOEL will combine the results of both match operations if they are both successful.

The Or Matching object is called whenever an or symbol '|' is used in a pattern definition. With an instruction of:

```
fetch(upto(&letters)) | pass(many(&digits))
```

SCOEL will try to fetch all the symbols up to a point where a letter is found. If it is success full in finding a letter, then it will store that information and proceed past the pass() match object. Should it fail, then it will move to the pass match object and attempt read some digits. If it finds any digits, then it will succeed, otherwise it will fail.

The Or '|' Matching object succeeds when either branch of a pattern succeeds. Should the first branch succeeds then it does not try the second branch and reports the success.

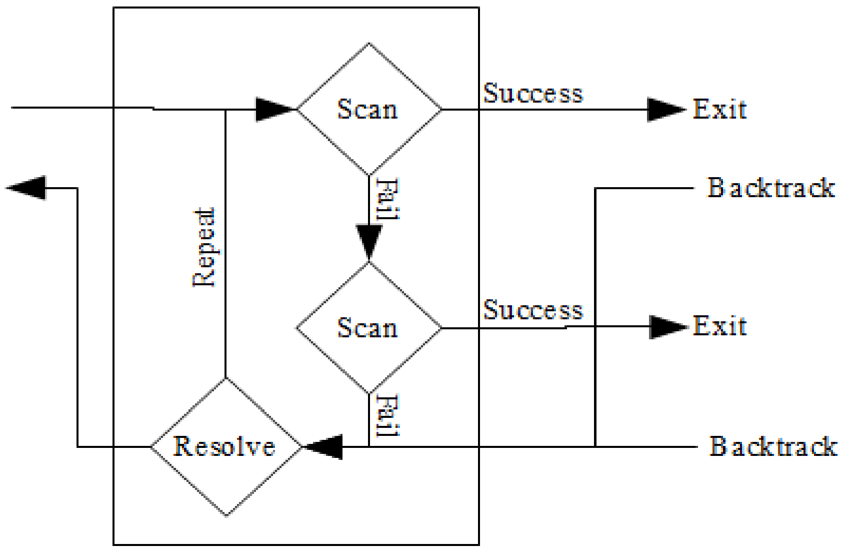

Figure 13: Or Structural Matching Function

To use an Or Matching object it is placed in a pattern definition as follows

```
pat := fetch(any('a')) | fetch(any("bcd"))
str := fetch(subject, pat)
```

In this example, there is a definition for pat in the first line. When the second line executed, the fetch will will first check to see if the next character in subject, if it is an 'a' then it will succeed and push the "a" to the stack. If it does not succeed, then it will try to find the sub-string "bcd".

While defining a pattern it is sometimes necessary and useful to group some Matching objects together with the use of parenthesis. When this is done, a Parenthesis Structural Matching Object is created which controls the flow of the extraction process to keep those items within the parenthesis combined together. The figure below shows the flow of the extraction process when a parenthesis are used. When the extraction process enters the Parenthesis Structural Matching object it is immediately directed to the first Match object in the parenthesis and performs the scans for each object in order until it reaches the last item. It then exits upon success to continue down the pattern definition. When it returns during backtracking it is sent to the last Matching object in the parenthesis and performs any back scanning and resolution as defined.

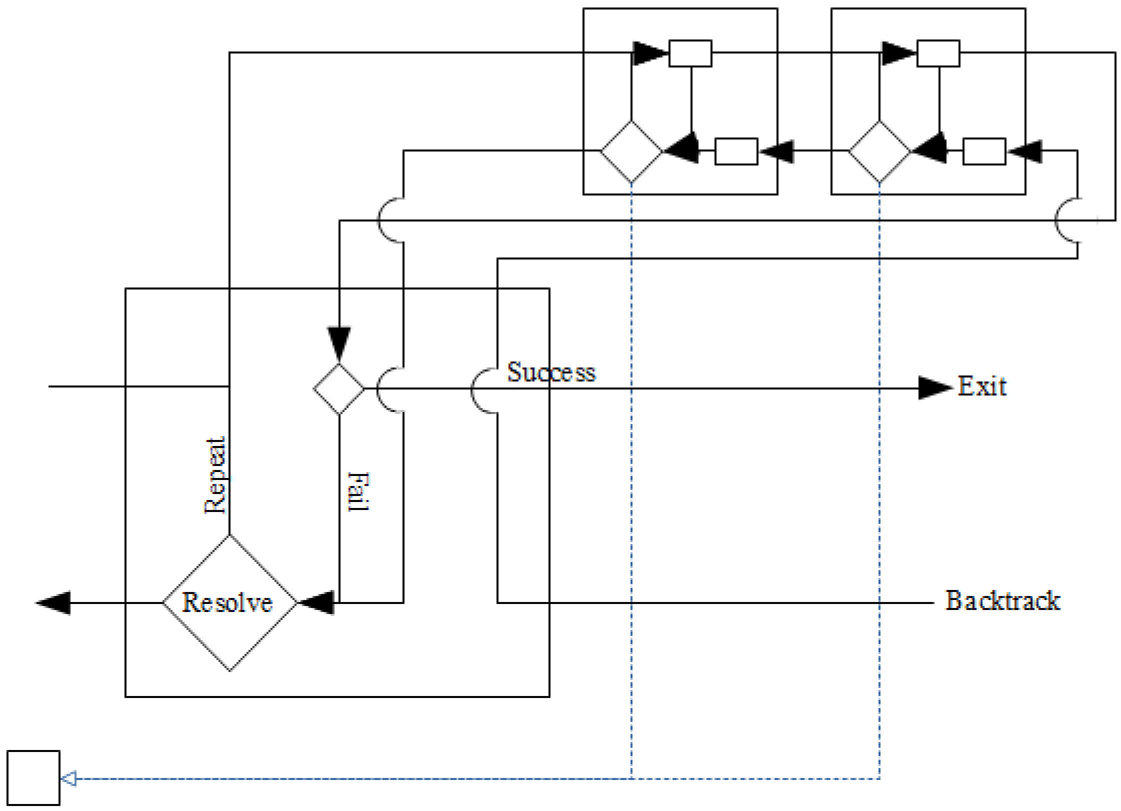

Figure 14: Parenthesis Structural Match Function

In the example below, the pattern will find either a token with ‘abc’ followed by a bunch of letters followed by a bunch of digits or a token which starts with a ‘d’ followed by a bunch of digits. The parenthesis group the fetch and pass Matching objects together.

```
Pat := ( fetch(any("abc")) pass(many(&letters)) ) |
     fetch(any("d")) fetch(many(&digits))
```

Anchoring

The pattern matching system of SNOBOL4 allowed the programmer to determine whether they wanted the pattern match to be tied to the current index location or to progress through the string until it finds a match. This concept is called anchoring. If the pattern match is anchored then it will not progress through the subject string until it finds a match, it will only test from the current location.[2]

In SNOBOL4 the default mode is unanchored, or to scan the string until it finds a match. In SCOEL the default mode is to be anchored, or to test for a match from the current index location only.

Like SNOBOL4, SCOEL allows the programmer to set the anchor mode. SCOEL allows greater flexibility by allowing each match object to be set to either anchored or unanchored mode. To perform an unanchored Match This can be done by using a adding a '#' symbol before the match object:

```
str := #fetch(pat)
```

In this example, SCOEL will look for a string of characters that match the pattern definition. Should it be necessary to differentiate which match objects in a pattern definition need to be anchored and others not it can be done as well:

```
pat := #pass(many(&letters)) fetch(many(&digits))
str := fetch(subject, pat)
```

In this example the pattern definition has an unanchored pass() Match object which will look for a set of letters in the string, then it will try to fetch a series of digits from the end of those letters.

Back Scanning

The pattern matching operations in SNOBOL4 and Unicon13 are limited by delivering a single token at a time. If there are bits of information spread through out a string, then separate pattern matching operations have to be performed.[1], [2], [9] When scanning information like HTML or XML may have opening and closing tags to identify different types of information. With SCOEL's back scanning the opening tag can be scanned then some other information may be analyzed and when the extraction process backtracks back to the Matching object, it can perform a scan for the closing tag.

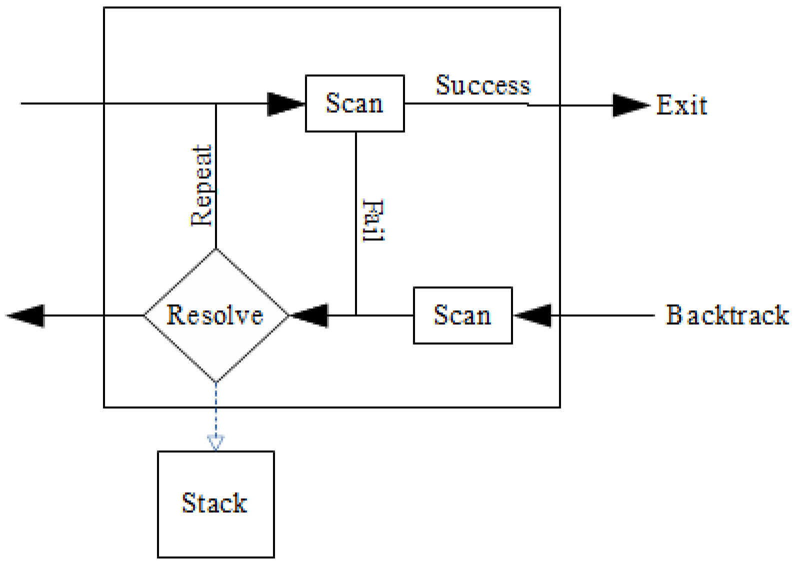

Figure 15: Matching Object

The above Matching Object figure shows the Scan object on the top half and is accessed when the Matching object is encountered. If successful it exits the Matching object. Upon its return it first encounters the Back Scan object where it can perform another scan. Should it succeed, it will push the results of the first scan on to the stack and put the results of the back scan on the stack.

Strings and Lists

Strings and Lists will be constructed as generator objects. They will store a value representing the current index location called a pos. There will be a set of "tools" or methods built-in which may be used to perform a number of tasks on them.

Collection Methods:

clear() removes all elements
member(elem) returns the elem if it is in the collection, otherwise fails
index(elem) returns the index where the element is located
insert(elem, int) inserts the element in the index of the second argument and pushes the remaining elements back one index.
set(elem, int) replaces the element at the index of the second argument with the element of the first argument.
push(elem) adds the element to the head.
pop() returns and removes the element at the head.
put(elem) adds the element to the tail.

put(list/set) adds all the elements of the argument to the tail.
pull() returns and removes the element at the tail.
get(int) returns the element at the index matching the argument.
remove(int) removes the element at the index matching the argument.
remove(elem) removes the element with is the same as the argument.
remove(list/set) removes all elements that are the same and included in the argument
head() returns the element at the head.
tail() returns the element at the tail.
sort()
copy() returns a pointer to the source collection with the index set to the source index.
clone() creates and returns a copy of the source collection with the index set to the source index.

Iteration Methods:

reset() sets the pos to 1
setpos(int) sets the pos to the integer argument.
next()
previous()

Related Research

Goal Directed String Scanning

Website Information Extraction with Goal Directed String Scanning

Finding Partitioned Tokens with SCOEL

Evaluating Context of Extracted Information with SCOEL

Extracting Information from HTML Tables with SCOEL

Other things suggested by others….

Bibliography

[1] C. Jeffery, S. Mohamed, J. Al-Gharaibeh, R. Perreda, and R. Parlett, *Programming with Unicon*. 2012.

[2] R. Griswold, J. F. Poage, and I. P. Polonsky, *The SNOBOL4 Programming Language*. Bell Telephone Laboratories, Inc., 1971.

[3] R. Griswold and M. Griswold, *The Icon Programming Language*, Third. 1996.

[4] H. Cunningham, "Information Extraction, Automatic," *Encyclopedia of Languages & Linguistics*, vol. 5. Elserier, pp. 665–677, 2006.

[5] N. Vlahovic, "Information Retrieval and Information Extraction in Web 2 . 0 environment," *Int. J. Comput.*, vol. 5, no. 1, 2011.

[6] B. B. Seppe vanden Broucke, *Practical Web Scraping for Data Science*. Apress, 2018.

[7] G. L. Hajba, *Website Scraping with Python*. Apress, 2018.

[8] D. Merritt, *Adventure in Prolog*. .

[9] R. Griswold, M. Griswold, K. Walker, and C. L. Jeffery, *The Implentation of Icon and Unicon*. 2014.