String Content Extraction Language (SCOEL)

Goal Directed String Scanning and Logical Programming Objects

by John Goettsche, PLS
New Mexico Tech

Table of Contents

Table of Figures

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, references to other legal or research documents, or determining how many times a company has been referenced in newspaper articles; software is being developed to handle the task.

Background

Several years ago, while I was working as a land surveyor, I was tasked with doing some legal research on a particularly large project. It was a redevelopment project for the Mission Bay District in San Francisco. The Mission Bay District was condemned as a result of the destruction of the 1989 earthquake. It was my task to map all the known and recoverable right-of-ways and easements in the district and prepare a boundary and topographic survey of the district. The boundary was not the difficult part, the City and County of San Francisco had condemned all the properties in the district and purchased the properties by imminent domain. It was all owned by the City and County of San Francisco.

The difficult task was to identify the rights-of-way and easements of the various entities that had them and do my best to locate them on the ground. A developer does not want to build a forty story building over an easement a utility company owns, because the utility company has a right to access it and a forty story building gets in the way of that. So it is in the interest of the developer to have the unity company abandon some easements while getting new ones that are more amenable to the development.

When I started the project, as is standard operating procedure for a land survey, I received all the pertinent deeds that were recorded at the courthouse along with any supplemental information which may affect property rights on the site. The initial set of documents arrived in seven boxes, enough to fill one and a half filing cabinets full of papers. It was my job to read them and identify any easement or right-of-way which may have been granted as some time. It was a majorly tedious task. It would have been a major savings in my time, which was being billed out at over a hundred dollars and hour, if this task could have been automated in some way. Unfortunately, I was did not have the benefits of automated information extraction at the time. I read it all, much of it several times to confirm references from several documents.

Typically a deed will contain a property description and its exceptions, easements, and so forth. Both the property description and its exceptions and easements make references to other documents to identify sources of others' parties interest in the property or adjoining property. It was these references which were of interest to me.

The problem with San Francisco was that there had been an earlier, much more destructive earthquake in 1906 with a fire that destroyed much of the public records. People still had their property rights and ownership was determined at that time through recovering personal documents and a series of state and federal acts to establish who owned what.

There as this utility company in the area which had several easements in the area, the San Francisco Gas Light Company. It had easements along all the roads, prior to 1906, and across several parcels of land. Unfortunately, the records specifically describing these easements were destroyed in 1906, yet the San Francisco Gas Light Company still owned these easements, at least on paper. As it turns out, we quit using gas lights and switched to electric lights at about the time of the 1906 earthquake and the San Francisco Gas Light Company no longer existed, other than as a paper entity. The developer was able to get those easements abandoned.

This project took months to perform, an information extraction system would have saved tens, possibly more than a hundred thousand dollars in billable hours. A programming language designed to extract information and make associations between various elements of information would save the software developer time and money in developing the software I needed on the Mission Bay project.

Goals of SCOEL

Many programming languages and software have been developed to handle problems associated with information extraction. These systems are designed to identify terms, relationships between terms, relationships between other relationships and collections of terms and relationships.

The String Content Extraction Language (SCOEL) will be designed to allow the user to extract complex distributions of information in strings of data. It will include a set of objects for logical programming which will enhance of identifying relationships between entities and other relationships.

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 will be an object oriented language with goal directed evaluation and some elements of logical programming. It will be derived from the Unicon programming language by Dr. Clinton Jeffery, et al., also an object oriented programming language with goal directed evaluation.

There are some differences in SCOEL. The objects in SCOEL are either defined by a class or a record, a record being a class without any methods and an implied constructor. There are global and constant variables, while variables in classes and methods must be declared to be either: local, read only or public. Class methods can be either a method, like those in Unicon; a fact, which is any object; a query; a rule; or a function which succeeds by returning or suspending a value, or it fails.

Since there are different environments for string extraction and logical processing, there is an invocable assignment which identifies the elements as being processed in their appropriate processing environment. Predicates will be processed in the logical processing environment and patterns will be processed in the extraction processing environment. Both of these environments are goal directed in that they seek a result or fail.

The implementation of SCOEL will be implemented in Unicon13. The lexicon and syntax is derived from Unicon and has been revised. Some of the terms have changed and some operators have been combined for simplicity for the programmer.

Object Oriented Programming

Object Oriented programming is a programming paradigm where classes are designed to hold the data and for its use, or behavior. The relationships between the objects determines how the program operates.[1] There are many different object-oriented programming languages including: Java, C++, Smalltalk, Unicon, PHP, Python, Visual Basic, and many others.

The advantages of object-oriented programming is its scalability and efficiency. Its main features are abstraction, encapsulation, inheritance, and polymorphism.

Unicon

Programming with Unicon describes Unicon's object-oriented programming as starting "with encapsulation, inheritance, and polymorphism."[2]

The Object Oriented aspects of Unicon were designed for simplicity and recognizes there are some situations where putting all the information into a class may not be the most desirable solution.[2] Therefore it is not entirely object-oriented.

Encapsulation

The principle of encapsulation is the hiding of variables and code in a class. Methods are used to access and manipulate the data or perform specific tasks. This is supposed to help protect the data from unintentional changes to your data. In languages like Java, it is encouraged the class variables be made private, or invisible to the outside classes. Having differing accessibility standings requires the system to check not only does the variable exists and is in scope, but also whether it is accessible.[1][2]

In Unicon it was decided to make the object variables available, or public, so they can be accessed from anywhere. This eliminated the need to check the visibility of class variables when referenced.[2]

Inheritance

A class can inherit the qualities of other classes. This helps reduce redundant code in larger programs. The class which other classes inherit is called the super class and the inheriting classes are sub-classes. If you create a primate class as a super class, with all kinds of data like hair color, eye color and teeth. The sub-classes chimp, gorilla and monkey, inherit the qualities of the primate class. Each would not have to contain their own code where they are common. Each could have their own code which is relevant to chimps gorillas or monkeys. You may want to add a tail attribute to the monkey class.

Unicon allows classes to inherit from multiple classes, in java multiple inheritance is only allowed with interfaces. An override of methods defined in the super class is done when a sub-class defines a method with the same name as the super class method. Since Unicon allows multiple inheritance, the super class must be identified when invoking an overridden method.[2]

```
object$superclass.method(parameters)
```

Polymorphism

Polymorphism is a property as a result of inheritance and encapsulation. A sub-class has the properties of the super class and can interact with other classes in the same way as the super class. If a class want to call any method which is defined in the super class, it will have to problem doing so, but if the method has been overwritten in the sub-class, then it will run the sub-class method.

Packages

In large programs, programmers may find themselves writing global variables which have matching names. This can be big trouble. A programmer will have difficulty finding out why a variable is changing its value when it is not supposed to. Packages allow for the partitioning of the global data so they will not interfere with one another. Declaring a package in Unicon tells the system where a piece of data or namespace where it exists and that it is not part of other packages.

In Unicon the contents of a package is the code that follows its declaration. To declare a package you use the keyword package followed by either a package name or a filename. To import a package it done similarly with the keyword import followed by the package name or a filename.

Class definitions

Classes are data structures defined by the user. In Unicon they are defined as follows:

```
class classname(attribute1, attribute2, …)
     method methodname(param1, param2, …)
          <method body>
     end
     . . .
initially(paramA, paramB, …)
     <method body>
end
```

Classes are instantiated by invoking its class name and some arguments like below:

```
object := classname(arg1, arg2, …)
```

To invoke a class method it would look something like this:

```
object.methodname(arg1, arg2, …)
```

Variable Scope

SCOEL

Like Unicon, Scoel is also an object oriented language. Although it is derived from Unicon, there are many differences in Scoel.

Encapsulation

Scoel requires accessibility to be part of the declaration of all its variables. The accessibility ranges include, global, const (constant), local, read (read only), and public. Global and const variables can only be declared outside of a class definition. The other three can only be declared inside a class definition, either inside or outside a method.

Inheritance

Scoel is like Unicon in its inheritance. Other than some notational differences, they are pretty much the same.

Packages

In Scoel, every text file containing code is a package. There is no need to declare a package because it is assumed. Classes, records and other information are segregated into the package where they are defined. Each package can any number of class and record definitions; facts, queries, rules and functions; and global and constant variables.

Classes

Classes in Scoel are similar to classes in Unicon, except the attributes are declared inside the class, not in the class header; also the constructor is called constructor:

```
class classname
     method methodname(param1, param2, …)
          <method body>
     end
     . . .
constructor(paramA, paramB, …)
     <method body>
end
```

Abstract Classes

Like interfaces, can have the signature of methods to be defined later, but can also have method definitions and variable declarations.

Variable Scope and Permissions

All variables in Scoel must have their accessibility declared. Topical constant and global variables can be declared outside of the class definitions, while private, read only and public accessibility can only be used inside a class definition.

- Topical variables are only accessible inside their package and may be modified from within its package.
- Constants are accessible anywhere but may not be modified.
- Global variables are accessible anywhere and can be modified anywhere.
- Private variables are only accessible inside their class or method.
- Read only variables are accessible anywhere, but may not be modified.
- Public variables are accessible anywhere and can be modified anywhere.

Goal Directed Evaluation

I first encountered Goal Directed Evaluation in the Unicon programming language. Unicon is a language created by Dr. Clinton Jeffery which is a derivative of the Icon programming language by Ralph Griswald. I found this for of evaluation to be very helpful in developing a program, in that it does not crash the program every time it cannot make a comparison or evaluate an expression because the data types do not match.

In Goal Directed Evaluation, the goal of an expression is to find a result. It succeeds when it finds a result or fails to do so. If it succeeds, it executes as intended. If it does not find a solution it does not perform the expression.{programming with Unicon}

Icon & Unicon

In Icon's and Unicons' Goal Directed Evaluation an expression succeeds if a value is produced. Therefore, the expression a < b

is evaluated if `a` has a value less than `b` then it returns `b`. When the values are both integer values, it is pretty straight forward, since the < operator is a numeric comparison. If one or both are not numeric values, Unicon will try to convert a and b to be numeric values for the comparison. Should it fail to do so, then the expression is not evaluated and fails. A string value comparison can be made with the << operator.

There are times when a programmer wants to have an expression fail in order to determine what to do next. For example you might want to test whether a variable has been assigned a value. The following if statement can be used:

```
if \someVariable then {
     do something
} else {
     do something else
}
```

Another common situation is when you want to do something with a variable, such as writing it, it does not produce an error and crash your system, it just does not perform the `write` expression.

Prolog

Prolog's Goal directed Evaluation is very similar to Icon and Unicon in that it is trying to find a result. When performing a query, Prolog is more analogous to searching for records in a database. It will return all the records that match the query, when it no longer can find any matches, it stops.{Adventures in Prolog}

Assignments

Scoel has three assignment operators, one for regular assignments, the second for conditional assignments and a third for invocable assignments. The first two control when how the assignment is made and the third is for pattern definitions, logical expressions and functions. They are:

- := Immediate Assignment, the expression on the right side of the operator is resolved and assigned to the variable on the left.

- :> Conditional Assignment, the expression on the right side of the operator is resolved and if it succeeds then it is assigned to the variable on the left.
- :– Invocable Assignment, used for assigning the logical predicate, logical rule, pattern definition or function on the right side of the operator to the variable on the left.

An example to illustrate how the first two act differently, lets start with the given:

```
local value
value := 5
```

Now consider the following expressions:

```
value := (some failing expression or &null)
value :> (some failing expression or &null)
```

The former value would be assigned &null as the expression failed, while the latter value would not be assigned any value and would remain 5. This allows the programmer to decide if the expression on the right is to be assigned in all cases or only when it succeeds. Originally, it was my intention only use this distinction in pattern definitions, but since Scoel will be using a pattern wrapper for expressions that are not extraction or structural objects.

Invocable assignments are designed to be used with logical expressions, pattern definitions, or functions. All may be invoked and evaluated during during run time with the current values of their elements. The structure of the invocable assignment determines what kind of processing paradigm is to be used when it is invoked.

Logical definitions

The logical programming elements of Scoel are similar to Prolog. There are atoms which are analogous to a data unit in a data base and can be stored as variables; a fact which is analogous to a record in a database and can be stored as a record or an object; a query which is similar to a query in a database; and a rule which is a collection of facts queries and expressions. The latter three definitions have their own structure. When Logical definitions are invoked, they use the logical processing paradigm. Scoel's logical programming will be discussed in detail later.

Pattern definitions

Pattern definitions are used in Scoel's string extraction processing, which will be discussed in detail later. Pattern definitions use the invocable assignment and must defined in the following format:

```
[patternName] :- {
     <pattern object or expression>
     <pattern object or expression>
     . . .
     <pattern object or expression>
}
```

The code block to the right of the invocable assignment can have any number of pattern objects and expressions as are needed. Eventually, I plan to make any expression work, presently it will only handle non structural expressions, any expression which creates a fork in the program flow.

Patterns can only be invoked as an argument in a string extraction invocation. There will be more on this later.

Functions

Same as Pattern definitions, uses the invocable assignment. They are proceesed in the standard processing paradigm as methods, classes and the like; and therefore must adhere to the structure below:

```
[functionName] (<parameters>) :- {
     <expression>
     <expression>
     . . .
     <expression>
}
```

Any number of expressions can be used in a function as necessary. Functions can be defined globally or as a part of a class and require a return or suspend expression.

Functions are invoked in the same manner as methods:

```
[functionName] (<parameters>)
```

Logical expressions

Logical expressions work with the object table, which contains all the objects created in the program. The table has lists for each type of object for faster referencing.

Facts

All objects created in a program is considered a fact are predicates in the logical paradigm. The object variables are its atoms. Every object created during run time is put into the system object table, which allows the program to access these facts and analyze them with user defined queries and rules.

Consider the research problem described in the introduction. I had several thousand deeds with legal descriptions where each contained at least one description describing the land it pertains to. In those legal descriptions, each property line was described giving a direction and a distance and may have made some references to other boundaries described in other deeds or boundaries. The deed could also describe a some easements and exceptions, each containing legal descriptions with multiple references to any number of deeds or other boundaries. It is possible to write some class definitions to handle all these items: legal descriptions, easements, exceptions and all the mentioned references. The search algorithm for all the references which can be contained in the deed could be difficult. They are referenced in many ways for many possible situations. Each deed would have to be stored in a deed object and would have to include the recording information for that deed, even the recording information could have references. Lets assume we have a created deed class and several records and classes for its components as well as a record definition for the references which looks something like this:

```
record deedReference(sourceDeed, deedItem, referncedDeed)
```

To create deed references all you would need to do is create a record applying the appropriate information. The deed object with has the reference call would be the first argument, the item in the deed where the call is made would be the second argument, and finally, the deed being referenced would be the third argument.

The Deed class will need variables to store the recording information, legal descriptions, exceptions and easements.

```
class Deed
     public
          instNum,
          recordingDate,
          legalDescriptions,
          exceptions,
          easements
     . . .
constructor(instNum, date, ld, ex, ease)
     . . .
end
```

To assign a fact to a variable you can use the invocable assignment or the immediate assignment, for example:

```
     thisDeed :- Deed(326458, December 14, 1979, LegalDescriptions_1,
     Exceptions_1, Easements_1)
```

or

```
     thisDeed :- Deed(326458, December 14, 1979, LegalDescriptions_1,
     Exceptions_1, Easements_1)
```

In both examples, a Deed object is created and stored in the system object table and it is assigned to the thisDeed variable. Assuming we made a deed object with an easement description in we want to reference and assigned it to the referenceDeed variable, then we could make a reference as follows:

```
     deedReference(thisDeed, legalDescriptions[2], referncеDeed)
```

This line of code would create a deedReference record containing three pieces of as was described above and store it in the system object table in a list with a keyword of `deedReference`.

Queries

Queries are predicates that perform a search of all objects which match their given values. Lets say we had created several objects for various Deeds we are working with and we wanted to get a list of all the Deeds which make a reference to thisDeed. A query could be defined as follows:

```
     referenceThisDeed :- (deed, item)
          deedReference(deed, item, thisDeed)
```

deedReference objects in the list. Notice the deed, and item parameters, these are local variables to temporrarily hold the sourceDeed and deedItem of the matching deedReference. These parameters identify the objects that can hold any value, or what we are searching for.

Rules

Rules are ordered lists of predicates, rules and other expressions to carry out some action. Lets say we wanted to print out all the instrument numbers and recording dates of all the deed that reference thisDeed. It could be written as :

```
printReferenceThisDeed :- (deed, item){
     deedReference(deed, item, thisDeed)
     write("Inst. No.: ", deed.instNum, " dated ",
               deed.recordingDate)
}
```

In this example, a rule was created with two elements, the first being a query definition and the second being a write expression. When this rule is invoked, it will iterate through all the deedReference objects and suspends the deedReference object and assigns the sourceDeed to deed, and the deedItem to item. It writes the instrument number and recording date of deed which is suspended from the query. In this case deed.instNum would equal 326458.

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.[2], [3], [4] 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 determinable 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.[5], [6]

The relationship between tokens or terms are can be difficult to find than the terms themselves. This is getting into the problem of what do the terms mean, rather than what they are. In my past work as a Professional Land Surveyor, I dealt with legal descriptions on a regular basis. There are legal definitions and centuries of court precedence establishing the meaning of many of the terms used in a property description; along with a hierarchy of which terms hold more weight in determining a boundary location.[7] As these are established relationships, they can be identified and analyzed. The goal of SCOEL is to make is to build a language with a more comprehensive system to identify terms and the relationships between those terms.

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'.

Iteration through strings

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 sentinel 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.

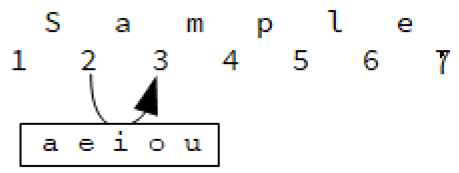

Figure 1: Scanning a single element based upon a set

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.

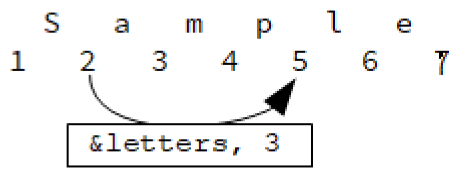

Figure 2: Scanning a number of elements based upon a set

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.

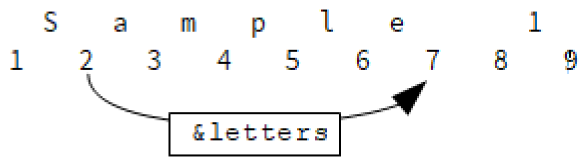

Figure 3: Scanning any number of elements based upon a set

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

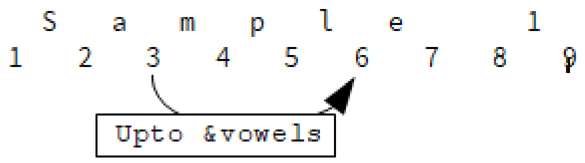

Figure 4: Scanning any number of characters until an element matches a member of a set

```
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.

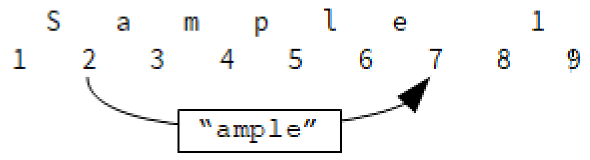

*Figure 5: S*canning elements which match another string of elements

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. Unlike matching a string of characters, the predefined pattern may or may not result in the same result every time. For example a pattern may start with either a ‘b’ or an ‘r’ followed by an ‘e’ or an ‘ea’ then concludes with a ‘d’. This can result in four possible results: ‘bed’, ‘red’, ‘bead’, or ‘read’. Finding any of these words in the string would be a successful result. In Unicon the code would look something like this

```
str := “read”
pat := (“b” .| “r” || “e” .| “ea” || “d”) -> result
str ?? pat
```

A diagram of the pattern is shown in Figure 6 below.

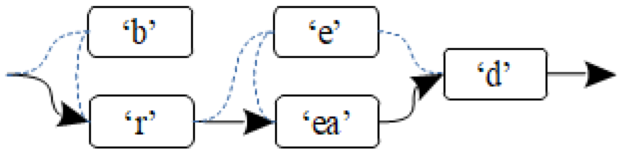

Figure 6: Pattern Example

When the pattern match is run, it first tries to find a ‘b’ in the string. It does not find one so it tries to find an ‘r’, which it does. Being successful, it moves to the next piece of the pattern definition, where it needs to find either an ‘e’ or an ‘ea’. It tries the ‘e’ and succeeds, then it tries to find a ‘d’ and fails. So, it tries the second argument in the or ‘ea’ and succeeds. Then it tries to find an ‘d’ and succeeds. The pattern match being a success, it assigns the value it found to the variable result.

3. The characters which match an ordered set of characters or a predefined pattern a certain number of times. A pattern or string similar to the examples in 1 and 2, which repeats a given number of times. For example you want to check to see if the string ‘ssi’ appears at least twice in ‘Mississippi’. A pattern can be defined to make that search and it would return ‘ssissi’. A Unicon patter definition would look something like this:

```
pat := “ssi” || “ssi”
```

4. The characters which match an ordered set of characters or a predefined pattern an indeterminate number of times. A pattern or string similar to the examples in 1 and 2, which is found repeating at least zero times. A Unicon patter definition would look something like this:

```
pat := Arbno("ssi")
```

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, where the system has to pair up the opening and closing elements of each type. It has to find the opening character and a matching closing.

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.

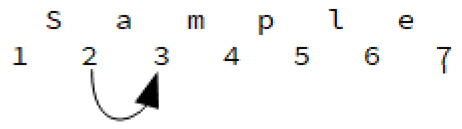

Figure 7: Scanning the next element

2. The next number of characters. If the next three characters of a string are needed then a Unicon pattern definition would be:

   ```
   pat := Len(3)
   ```

3. All the characters from the current index to the a specified index. An example of such a pattern definition would be

   ```
   pat := Arb() || Pos(25)
   ```

   In this case the pattern would match all the elements from the current cursor position to cursor position 25.

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.

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. The internal structure of a string extraction object contains a scan object which determines which sub-string is to be identified for the closure process. If the scan succeeds, the extraction process exits the extraction object and proceeds to the next extraction object. If the scan fails, it begins the closure process with a &fail message.

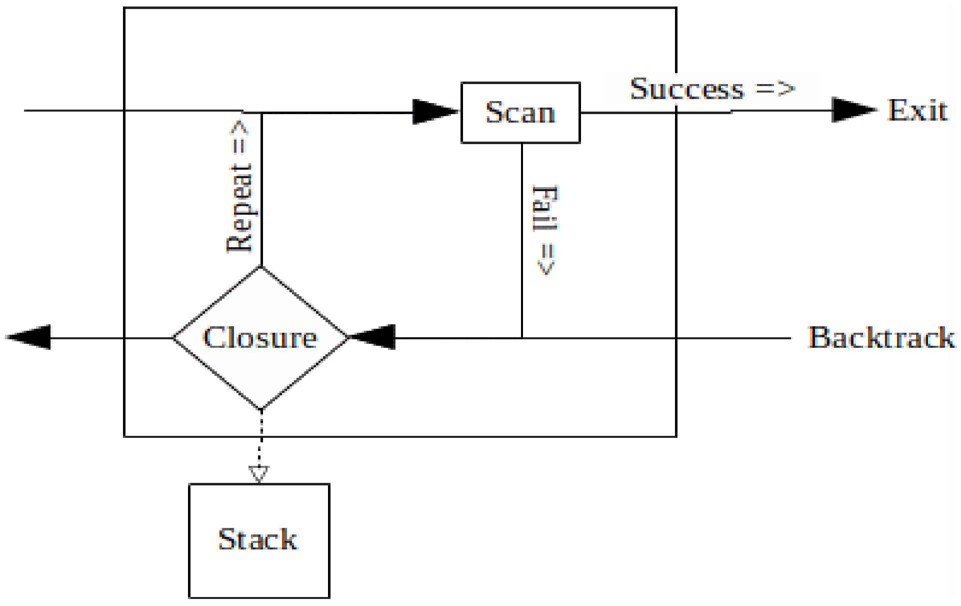

Figure 8: SCOEL Extraction object and its internal workings

The extraction objects which are used to determine whether the information collected in the scan are stored on the stack, counted or disregarded. It uses a prefix system to identify which operation will be performed during closure:

- **c_** (collate) stores the elements collected from a successful scan to the stack and combines them into a list of tokens.
- **f_** (anchored fetch) anchored match described further below, stores the elements collected from a successful scan to the stack.
- **F_** (unanchored fetch) unanchored match described further below, stores the elements collected from a successful scan to the stack.
- **p_** (anchored pass) anchored match described further below, does not store the elements collected from a successful scan to the stack.
- **P_** (unanchored pass) unanchored match described further below, does not store the elements collected from a successful scan to the stack.
- **t_** (tabulate) returns the number of times it can find the scan successfully.

Eventually the chain of extraction objects is capped with a closure object. If one is not included in a pattern definition or a string extraction function is used, then a closure object with a &success argument is automatically used. But, when defining a pattern definition with a closure object, it can hold either a &success, &fail or &terminate argument for closure. While performing closure if &success message is passed back to the previous extraction object, it will push the results onto the stack. When a &fail or &terminate is sent back to the previous Matching object, the extraction object is skipped and nothing is stored to the stack.

- **closure**(&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 closure object, the scan object does not perform a scan, it is where it sets up the required components for the closure process and backtracks through the previous extraction objects.

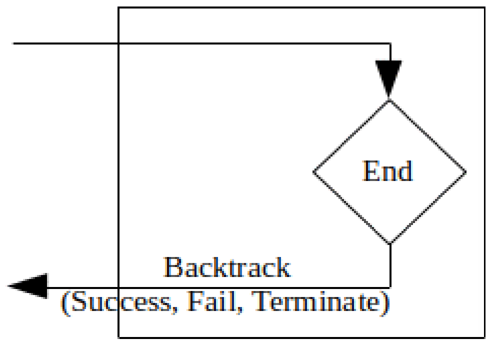

Figure 9: Closure object

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 :- {
     p_many(&letters)
     f_many(&digits, 3)
     f_any(&letters)
     closure(&success)
}
```

The following diagram shows how the pattern definition above would be structured.

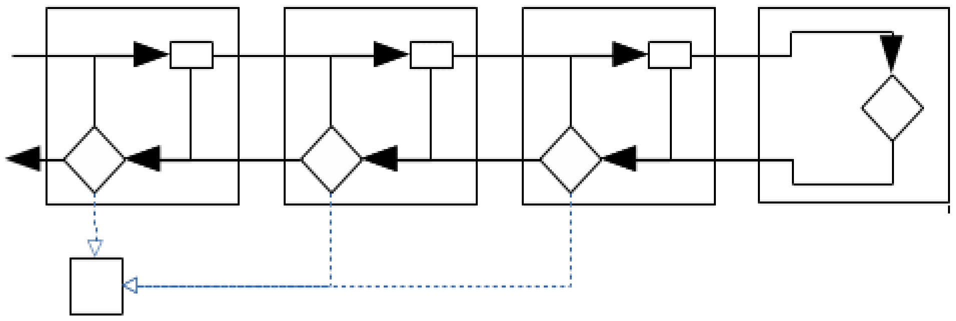
Figure 10: Pattern definition structure

A pattern definition called pat has three extraction objects; the first passes a set of letters, second fetches the next three digits and then it fetches a letter, finally it encounters a closure object. Had there not been a closure object it would by default place a closure(&success) object on the end. A closure object can be given either of the &success, &fail, or &terminate keywords.

- **closure**(&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 closure object:

  ```
  pat := {
       p_pass(&letters)
       f_many(&digits, 3)
       f_any(&letters)
       closure(&fail)
  }
  ```

Running this example will result in nothing being returned, because a fail message would be sent back during the closure operation and none of the closure methods will push their results on to the stack.

Additional Extraction Objects

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.

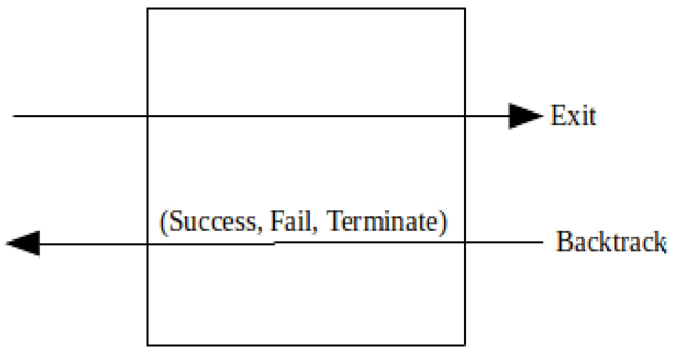

Figure 11: Fence Matching Object

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

```
pat :- {
     p_many(&letters)
     f_many(&digits, 3)
     fence(&success)
     f_any(&letters)
     closure(&fail)
}

str.f_any(pat)
```

This example differs from the previous, in a fence(&success) has been inserted between the f_many(&digits) and the f_any(&letters). The closure process will not add to the stack the results from the f_any(&letters) extraction object as it is sending a &fail message back until it encounters the fence where a &success message is sent back. Then it will push the results from the f_many(&digits, 3), the p_many(&letters) would not as it is a pass extraction 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:

```
f_upto(&letters) | p_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.

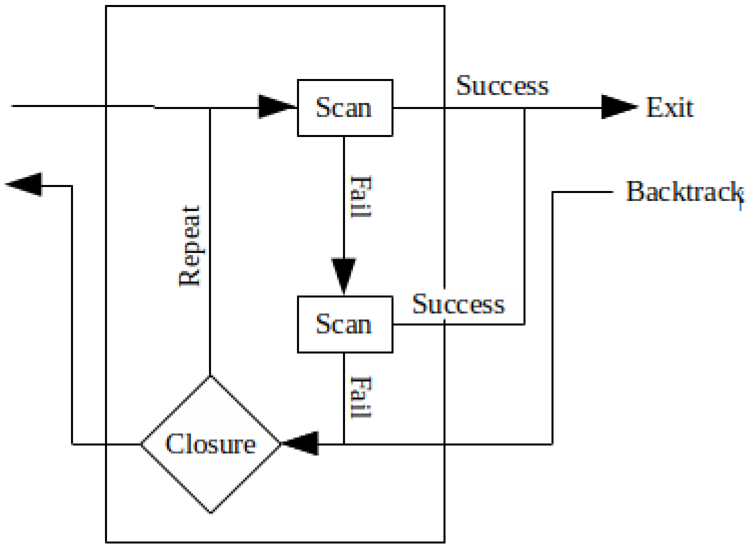

Figure 12: Or extraction object

The Or '|' extraction 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. To use an Or extraction object it is placed in a pattern definition as follows

```
pat := fetch(any('a')) | fetch(any("bcd"))
str.f_any(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.

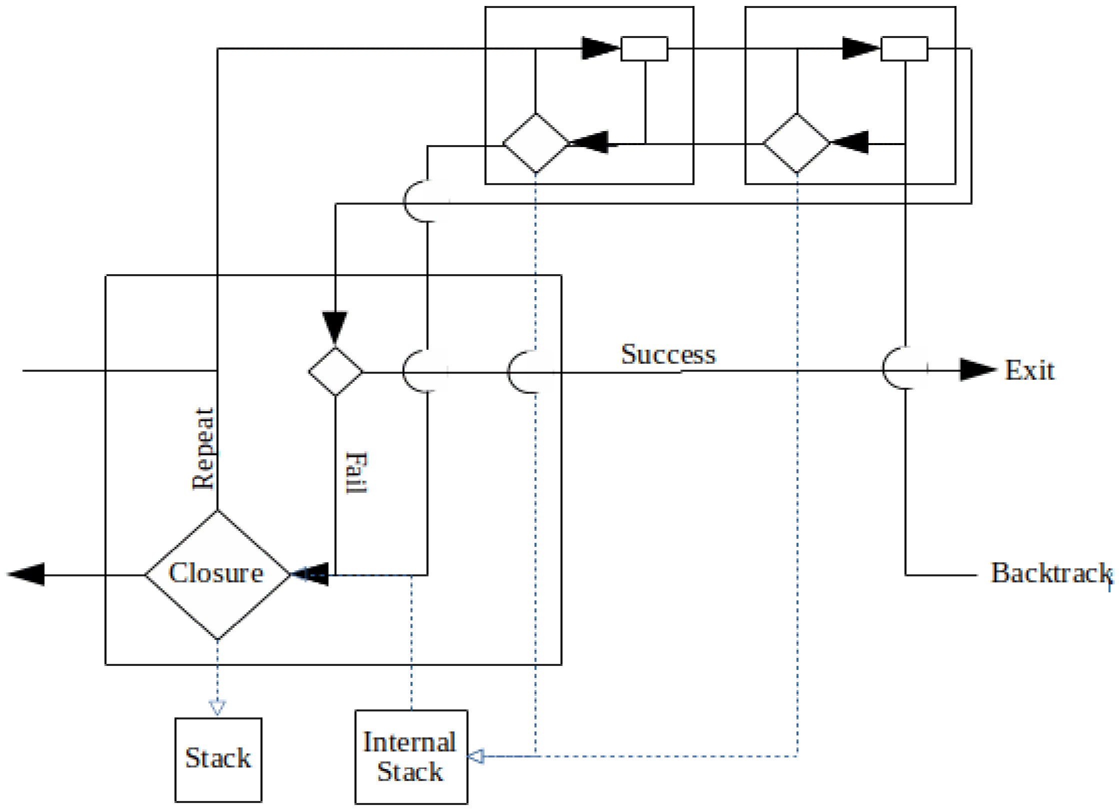

Figure 13: Parenthesis extraction object

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 := {
     (    f_any("abc")
          p_many(&letters)
     ) |  f_any("d")
          f_many(&digits)
}
```

Assignments during extraction

The ability to assign the results of internal scans of a pattern is useful in being able to control the flow of pattern recognition. For example, you scan an integer value which tells you how many characters that follow, it can be assigned to a variable and then passed as an argument later in a scan later in the pattern. The three types of pattern assignments are the immediate assignment, the conditional assignment and the index assignment; similar to what is available in SNOBOL4.

Immediate Assignment

An immediate assignment is made when a pattern contains the following expressions:

```
pat :- {
     p_many(&letters)
     pos := self.pos()
     closure(&fail)
}
```

or

```
pat :- {
     p_many(&letters)
     str := f_many(&digits)
     f_many(&letters)
     closure(&success)
}
```

This tells the information extraction system to assign the results of a match object to a variable called variable. It will make this assignment regardless to weather the complete matching operation is successful or whether the matching function stores the information on to the stack. The scan in the previous matching object must be successful.

Conditional Assignment

The conditional assignment is structured slightly different than the other two. It will only make the assignment during backtracking if a 'success' message is sent back to the assignment matching object. The conditional assignment operator is '->'.

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.[3]

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 pattern element to be set to either anchored or unanchored mode. To perform an unanchored extraction, use a capital letter at the beginning instead:

```
str.F_any(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 := {
     P_many(&letters)
     f_many(&digits)
}
str.f_(pat)
```

In this example the pattern definition has an unanchored P_many() pattern element which will look for a set of letters in the string, then it will try to fetch a series of digits immediately after those letters.

Lists

Lists are collections where the elements are linked in a consecutive order. Lists will not have the logical and extraction processes available to them. It will be considered at some later date. They use the same index scheme and store their present index location. Lists are objects which have at least the following methods:

List Methods:

- **clear**() removes all elements
- **member**(elem) returns the elem if it is in the collection, otherwise fails
- **index**(elem) returns the first 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.

List iteration methods:

- **reset**() sets the pos to 1
- **setPos**(int) sets the pos to the integer argument.
- **getPos**() returns the current pos
- **incrument**() increase the pos by one
- **incrument**(int) increase the pos by the integer value
- **decrument**() decrease the pos by one
- **decrument**(int) decrease the pos by the integer value

Conclusions

SCOEL has its origins in Icon and Unicon, but it is a strictly object oriented language and it is designed to expand upon Icon and Unicon string scanning and pattern matching operations and provide a set of logical programming operations and object. The Mission Bay District of San Francisco project was a project I labored over for many months and needed an automated system to identify references to other deeds. Having a language like Scoel, which combines object oriented programming with features of logical programming would improve the productivity of developers to handle problems like the Mission Bay District of San Francisco.

Related Research

Future research as a result of the development of SCOEL would include: Goal Directed String Scanning, Finding Partitioned Tokens with SCOEL, and Extracting information from collections of property descriptions with SCOEL.

[1] C. Horstmann, *Java for Everyone*. John Wiley & Sons, 2010.

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

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

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

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

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

[7] W. Robillard, D. Wilson, and C. Brown, *Evidence and Procedures for Boundary Location*, Fourth Edi. John Wiley & Sons, 2002.