Study Project Report

Rajath S
2012B5A7589P

Matching Strings generated by a Context Free Grammar

The language generated by a Context Free Grammar G (∑, V, P, S), denoted by L(G) is a set of all strings (using alphabet ∑), that have a valid derivation (restricted by production rules P), from the
start symbol S. The set of context free languages is a proper superset of the set of regular languages. Every regular expression can be expressed as a left-linear context free grammar. Note that, Context Free Grammars have significant practical applications, including programming languages, data representation, natural languages, etc.

The process of verifying whether a string is generated by a CFG, is known as parsing. A parser produces a unique parse tree for each string generated by the CFG (if the grammar is unambiguous), which is a graphical representation of the derivation of the string using production rules specified by the CFG. The worst case asymptotic complexity of Parsing is Θ(x^3 * |G|) using dynamic programming. However, for a restrictive class of CFGs (sufficient for most practical pursposes),
a linear time parsing algorithm can be constructed [LALR parsing is the most preffered parsing method for practical use].

The problem of syntactically matching two strings generated by the same CFG is trivial. Two strings P and Q are equal if and only if, |P| = |Q| and P[i] = Q[i] for all i=1..|P|. If the two strings are equal, then one can easily verify if the string can be generated from the CFG using parsing techniques discussed above. The objective of this project was to encode semantic meaning in to this matching process. Based on the requirements of a domain, two syntactically unequal strings may be semantically equivalent. However, if two strings are syntactically equal, they are semantically equivalent by virtue of the strict definition of the notion of syntactic equivalence. For instance, if we consider the simple expression grammar, strings “x*y” and “y*x” are not equal syntactically. If the domain under consideration is natural numbers, then the strings are semantically equivalent. In case of Matrix algebra, this is in general not true as multiplication is a non-commutative operation. User input is required to define the notion of semantic equivalence, as it is clearly dependent on the domain and cannot be inferred from the grammar.

The semantic rules can be captured in terms of a relationship between non-terminals of the grammar. To clarify, semantic rules can be expressed as a pattern in the parse tree and a rewrite rule that can produce another semantically equivalent parse tree. Note that, application of the rewrite rule must not produce a parse tree that cannot be generated by the given context free grammar. As an example, consider the expression grammar G (figure 1):

$$E \rightarrow E + T \;|\; E - T \;|\; T$$
$$T \rightarrow T * F \;|\; T / F \;|\; F$$
$$F \rightarrow \text{<id>} \;|\; (E)$$

Commutativity of addition can be expressed as the following rewrite rule :

```
      E                                           E
     /|\                                         /|\
    / | \                                       / | \
   E  +  T₂              ====>                                  E  + T₁
  /|\                                               /|\
 / | \                                             / | \
E  +  T₁                                          E  + T₂
```

* Note : numbers are only to differentiate more than one appearance of the same non-terminal.

Parse tree being a naturally recursive structure, the rules can be repeatedly applied to obtain several semantically equivalent strings. Also note that, associativity of +, –, *, / , distributivity of * over +, etc can be expressed with similar tree rewrite rules.

Tree rewrite rules are not the most general way to express semantic rules. There are certain semantics that cannot be captured using tree-rewrite rules. Another problem with tree rewrite rules is that the number of such rules required to express the semantics may become very large due to combinatorial explosion.

Most often, the non-terminals of a Context Free Grammar, do not directly represent domain specific constructs. Sometimes, grammars are altered to fit the requirements of LL(k) or LR(k) parsing. This modification requires additional non-terminals without any meaning in the domain. For instance, the grammar shown in (figure 1) is left recrusive and left-factoring has to be performed to convert it into a LL(1) grammar. These additional non-terminals, further hamper the expressiveness of the rewrite rules.

Provided these rewrite rules, a backtracking based algorithm can be developed to match the two strings applying the rewrite rules recursively. This algorithm is naturally exponential in the worst case because at each rule match, two possiblities have to be explored and in the worst case all the nodes in the tree match the rule. Although, a theoretical proof to prove that a polynomial time algorithm to match the strings does not exist is not provided, the above informal explanation makes it clear. However, the search space for backtracking can be pruned by using the second string as the guide.

Exponential complexity is unacceptable for practical applications. To attack this problem from a pratical perspective, a randomization approach is considered. A monte carlo algorithm, with one-sided error probability is developed using the idea of hashing. The idea is to develop a hash function, to hash parse trees and match the hash values. The semantic rules are encoded in the hash function. Care has to be taken to ensure that the calculation of hash value must not take exponential time.

Hash function for tree structures can be defined recursively. The hash value of a node is calculated by calculating the hash value of each of its children independently and combining the hash values using using a combination function. For example, the tree rewrite rule for commutativity of addition can be captured by using a commutative combination function (like XOR). Capturing associativity is tricky, as the operands can span over multiple levels in the tree. In this case, two different approaches can be taken. The first approach is to use canonicalization – transforming parse tree into a canonical form (like converting a boolean expession into CNF) and then applying the hash function on the canonical parse tree. The second approach is to use level information in hash function. By treating operands of an associative operator at the same level, the hash function can encode associativity. This is much like the syntax directed translation phase in compilers.

The scope of development of hash functions was restricted to encoding commutativity and associativity. This approach was tested on XML grammar. The semantic input was provided through the XML Schema defintion. Schema provides three ways to express order amongst the elements : choice, sequence and all. Choice is a selector between multiple rules, sequence specifies an ordered list of elements and all specifies an unordered list of elements. Nesting of all withing choice/selector and vice-versa is not allowed.

This was tested using the JAVAX DOM Parser. A single pass over the schema file is sufficient to determine if an element has ordered/unordered children. This information is aggregated into a hash table. The hash function uses commutative/non-commutative hash as per the ordered/unordered flag status of an element. (Schema representation was chosen over the DTD representation because Schema is also a well-formed XML document and DOM like traversal can be performed on it).

This XML matching can be converted into a useful software for domain experts, to provide semantic input and data to find matches on large datasets efficiently. However, the domain experts may not be aware of XML Schema definition and the data may not be available in the XML format. To aid the domain experts in creating the XML schema, a UI design is presented below to interactively create the
XML Schema –

UI design -

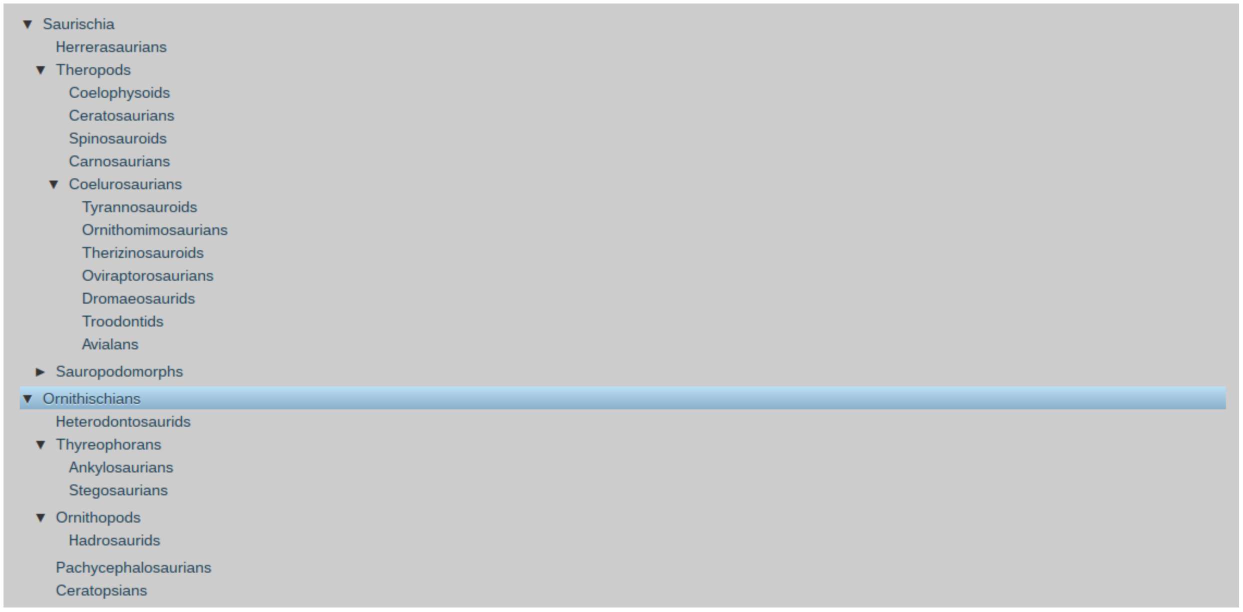
Window is split (vertically) into two panes -

Pane 1 – is used to manage the tree structure.

Pane 2 – is used to set properties of nodes in the trees.

Pane 1

Displays the tree structure in the form of foldable nodes.



- Nodes are selectable.
- Nodes can be re-ordered, moved up/down the hierarchy using the drag-and-drop interface.
- When a node is double clicked, a context menu is opened in Pane – 2 (details provided in the next section).

Adding new nodes to the hierarchy:

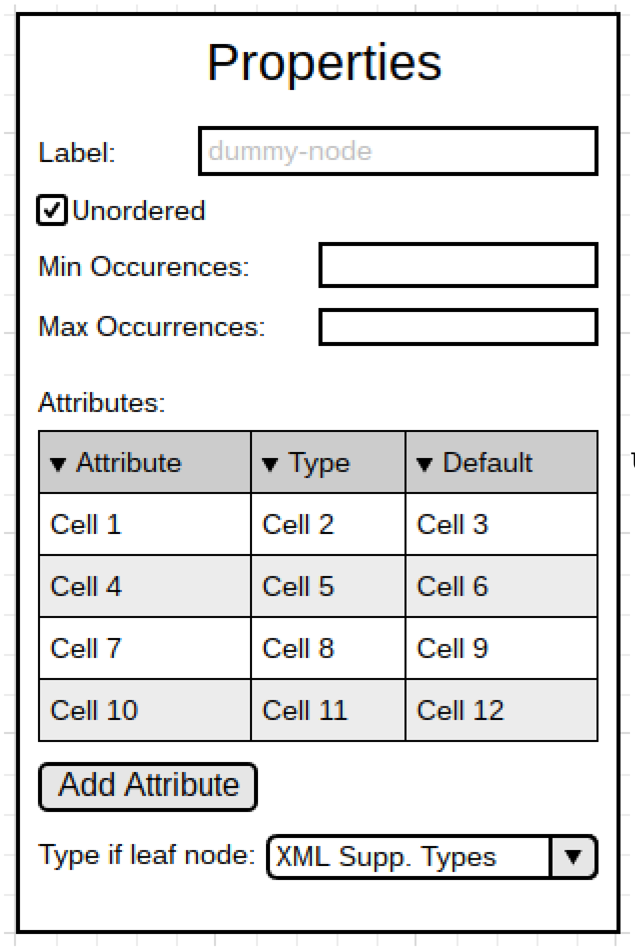
1. Select a parent node (by single click) and click the add button (placed in the status bar in pane – 1). A new node will be added as the last child. Properties of the node (including name) can be edited using Pane – 2.
2. Right click on a node to open a menu with options to add a node above/below the selected node.

A node can be marked as an “Choice” node. This will be a dummy node which behaves just like the XML “Choice” selector. When a node is marked as a choice node, any properties marked in Pane 2 is ignored.

(Nodes can be rearranged using drag-and-drop).

Pane 2

Interface to manage properties of a node.



▼ Attribute	▼ Type	▼ Default
Cell 1	Cell 2	Cell 3
Cell 4	Cell 5	Cell 6
Cell 7	Cell 8	Cell 9
Cell 10	Cell 11	Cell 12

- Element label
- Ordered/Unordered
- Num Occurences
- Attributes
- Type if leaf node

The above properties can be modified using the interface.

Implementation :
The program can be implemented using the HTML, CSS and JS web frontent technologies. A new platform called "Electron" based on NodeJS and Chome V8 JS engine for supporting development of desktop applications using HTML, CSS and JS is available as free & opensource software. This platform can be utilized to create responsive and asynchronous UI using Jquery related packages. The tree structure can be displayed using the "Jqtree" (http://mbraak.github.io/jqTree/) package. Jqtree supports event listeners for actions such as "nodeSelected", "nodeDoubleClicked", "nodeDragged", etc.

An intermediate representation is not required as the event listeners can be used to directly manipulate the XML DOM object in the JS. DOM object manipulation provides getNextChild(), getPrevChild(), getParent(), getChildAfter(), etc. methods that can be called when drag and drop is used to modify the tree representation.

The properties of the nodes have direct representation in the XML Schema

specification –
The attributes can be represented by – <xs:attribute> tag. The default value can be specified in the same tag. The choice nodes can be specified using the <xs:choice> tag. The nesting in the tree has to be translated to nesting of XML tags. Children of unordered nodes can be enclosed in <xs:all> tag and ordered nodes can be enclosed in <xs:sequence> tag. The element tag has minOccurance and maxOccurance attributes that can be inserted from the input.

The UI also has to generate an interface to input data interactively to output XML. The generated interface will be similar to the above UI except that the tree structure will be locked for modification and the data for leaf nodes can be filled using the Pane-2.

In conclusion, the project provides an efficient scheme to semantically match strings generated by a context free grammar, demonstrates the approach by applying it to XML domain and a design to support domain experts by UI is illustrated.