CS

COC251

B719235

SQLSQLLSQ - String eQuality Logic for SQL-Like String Queries

by

Liam C. George

Supervisor: Dr D. D. Freydenberger

Department of Computer Science

Loughborough University

April 2020

1 - Acknowledgements

Dr Dominik D. Freydenberger, for general supervision, consultation, expertise, and going above and beyond his duties to ensure my physical and mental wellbeing throughout my time at the University, not just the project, as well as directing me towards relevant research. Aside from patience and explanations, tolerance, care and consideration were shown, the value of which has been insurmountable especially in helping me feel positive about the work.

My family, including Eleanor Kirkham, for support and consistently demonstrating interest in a field unlike their own.

2- Abstract

"The project, titled SQLSQLLSQ (String eQuality Logic in SQL-Like String Queries), is an investigation into a method of querying strings that's more powerful than Regular Expressions. The aim is to emulate some of the more powerful features of SQL querying on large bodied strings, including wildcards and patterns (as in those described in pattern languages - essentially variables, or consistent wildcards) to allow for more effective searching of large documents. This would be done through a variant on First-Order Logic."
-Project Brief submission.

In all, the project intends to develop a more powerful string querying method than Regular Expressions by modelling powerful features from SQL. This is done using a purpose crafted language based on Propositional Logic formulae.

The scope of the project is thus:

- To create and implement a language capable of modelling propositions about the composition of strings
- To create a parser for said language
- To create an evaluator for said parser's result
- To optimise said evaluator to a reasonable extent.

All of this is done done with resource efficiency in mind, and optimisations being aimed at longer strings.

To make the project feasible, a limitation is placed on the strings considered, modelling in part the logic FC [1]. This means that the universe is exactly the set of substrings of the string to be searched, to ensure decidability by making the universe finite.

Two approaches are used to evaluate the parsed queries, referred to as the "top-down" and "bottom-up" approaches. The top-down approach considers all possible combinations of assignments of values to the free variables in a query and uses these to evaluate the query in a top-down manner. The bottom-up approach considers what sets of values for all variables would satisfy said term, and combines these in accordance with the present logical operators and filtering out those which don't satisfy the existential quantifiers before removing the variables quantified (as they are no longer free).

A formula is used to estimate the time taken for each approach and decide which should be used, as both are expensive in both time and memory for longer strings to be searched.

The result of the project should be software that takes a user-entered query and a user-entered body of text to be searched, returning a table of combined values for any free variables for whom the query is satisfied.

Table of Contents

3 - Theory

The usage of Propositional (First-Order) Logic as a base for the model is by design, as it is tailored well for computation and has a system in place to model those more powerful functions we intend to replicate from SQL. Usage of existential quantifiers will substitute well for wildcards, giving users the ability to craft more powerful queries with a higher degree of ease.

Propositional Logic formulae may be easily and perfectly represented through parse trees, lending itself to a model well-tailored for computational evaluation.

I have read into some data structures [2,3,4,5,6,7,8], as well as Parse Trees [9] and Parser Generators [10,11,12,13] in preparation for various parts of the project.

To ensure a finite universe we use that of the logic FC [1], limiting the universe to only the substrings of the search material.

As the number of substrings (or, more formally, ordered sets of *consecutive* characters within a given string, not to be confused with any selection of ordered characters – this clarification has proven necessary in discussion to distinguish between sets of in-order characters from the string and sets of consecutive characters from the string during research) is proportional to the square of the string's length (See 3.1.1 - Return All Substrings Algorithm), it occurred to me that usage of a storage method that lends itself to $log(n)$ search times would negate the polynomial growth of time cost for checking that substrings aren't repeated, lowering the time cost of a substring enumeration algorithm from $O(n^3)$ to $O(n^2 \cdot log(n))$ where n is the length of the string.

This is because the insertion of n^2 items into a data structure whose insertion cost is, on average, belonging to the order of $O(log(n))$ would have a total cost belonging to the order $O(log(p(n)))$, where $p(n)$ is some polynomial function on n. Since no polynomial function on n is not exceeded by some exponent q of n, it follows from the fact that $log(n \cdot m) \equiv m\, log(n)$ that the cost of assembling said data structure would belong to the order $O(log(n))$. This means that most of the cost of enumerating the substrings of the body string will come from the enumeration itself, not the storing of the results.

As said storing is algorithmically similar to (and indeed, often a mere extension of the algorithm for) searching for value in many data structures, it will be easy to streamline the two together in a way that helps to avoid repeated substrings, by including no substrings of a string already present in the structure. Unfortunately, I failed to find a data structure with all the right parameters.

3.1 - Proofs

3.1.1 - Return All Substrings Algorithm

Definition

Input: Any string S.
Output: All substrings of S.

$GetAllSubstrings$ function (Arguments: S):

1. Consider an empty collection of strings C
2. Add S to C
3. If S is not the empty string:
    a. Let $L :=$ S without its final (rightmost) character
    b. Let $R :=$ S without its first (leftmost) character
    c. Add to C all strings who are returned by $GetAllSubstrings(L)$ and are not identical to any string in C
    d. Add to C all strings who are returned by $GetRightSubstrings(L)$ and are not equal to any element of C
4. Return C

$GetRightSubstrings$ function (Arguments: a string S):

1. Consider and empty collection of strings C
2. Add S to C
3. If S is not the empty string
    a. Let $R :=$ S without its first (leftmost) character
    b. Add all strings returned by $GetRightSubstrings(R)$
4. Return C

Proof

First, we consider a method for enumeration of the substrings of a given string.

Let the functions $RemoveFirstCharacter(S)$ and $RemoveFinalCharacter(S)$ return S missing its first or last character respectively, except where S is the empty string, in which case they both return the empty string.

Any substring can be found by placing a pointer in (or to either side of) the string, placing another pointer after or in the same place as the first, and taking the section of the string that lies between them.

It follows that, of a string with N unique characters, as there are N positions for the first pointer to be placed that won't always result in the empty string, and $N - P$ positions for the second pointer to be placed that won't result in the empty string, where P is the number of characters to the left of the first pointer, there are $N + (N - 1) + (N - 2) + \cdots + 1$ substrings, excluding the empty string.

$\sum_{n=1}^{N} n = \frac{1}{2}n(n+1)$ holds for all $N \in \mathbb{N}$. Hence, any string with N unique characters has $\frac{1}{2}n(n+1) + 1$ substrings, and the algorithm has a worst-case runtime and space usage of at least $O(n^2)$.

The only substring to contain both the first and last characters of a string is the string itself. Hence, the set of substrings of a string of N characters contains the string itself, and all substrings of the string's substrings of length $N - 1$.

Hence, the set of all substrings of any string S is equal to the union of:

1. the set of substrings of $RemoveFirstCharacter(S)$,
2. the set of substrings of $RemoveLastCharacter(S)$,
3. and the set containing only S.

$RemoveFirstCharacter(RemoveLastCharacter(S))$ always returns the same result as $RemoveLastCharacter(RemoveFirstCharacter(S))$, as both return the empty string where the string has a length of 2 or less, or S missing both is first and last characters otherwise.

Hence, excepting those substrings that can be found by repeatedly applying $RemoveLastCharacter$ to S, no substrings of $RemoveLastCharacter(S)$ are not also substrings of $RemoveFirstCharacter(S)$.

Therefore, the set of all substrings of any string S is equal to the union of:

1. the set substrings of $RemoveFirstCharacter(S)$,
2. the set of strings that are the result of any number of recursive applications of $RemoveLastCharacter$ to S,
3. and the set containing only S.

The $GetRightSubstrings$ function

Case 1.1: $GetRightSubstrings$ is called, S is "" (the empty string)

- S is added to C
- S is the empty string, so we skip to 4
- C is returned.

This returns {""}, so in this case $RemoveLastCharacter(R)$ returns the set of strings that are the result of any number of recursive applications of $RemoveLastCharacter$ to S in this case.

Case 1.2: $GetRightSubstrings$ is called, S is of length $N + 1, N \in \mathbb{N}_0$

- S is added to C
- S is not the empty string, so we do not skip to 4
- R is $RemoveLastCharacter(S)$.
- $GetRightSubstrings$ is called, S is R, its results are added to C.
- C is returned.

Since R has a length of one less than S (following from the definition of $RemoveLastCharacter$), the recursive call on $GetRightSubstrings$ will terminate.

Supposing $RemoveLastCharacter(R)$ returns all substrings of S except S itself, since S is added to its result and the result is then returned, $RemoveLastCharacter$ gets the set of strings that are the result of any number of recursive applications of $RemoveLastCharacter$ to S.

Therefore, if $RemoveLastCharacter(T)$ gets the set of strings that are the result of any number of recursive applications of $RemoveLastCharacter$ to T for any string T, $RemoveLastCharacter(T + x)$ gets the set of strings that are the result of any number of recursive applications of $RemoveLastCharacter$ to $(T + x)$ where x is any single character string and $+$ is the concatenation function.

Therefore, as all strings can be made by adding a series of characters to the empty string, and $RemoveLastCharacter(S)$ returns the set {“”} where S is the empty string, $RemoveLastCharacter(S)$ always returns the set the set of strings that are the result of any number of recursive applications of $RemoveLastCharacter$ to S.

The $GetAllSubstrings$ function

Case 2.1: S = “” (the empty string)

- S is added to C
- S is the empty string, so we skip to 4
- C is returned

Hence, the result is {“”}, and $GetAllSubstrings$ gets all substrings of S where S is the empty string.

Case 2.2: S is of length $N + 1, N \in \mathbb{N}_0$

- S is added to C
- S is not the empty string, so we do not skip to 4
- L is $RemoveLastCharacter(S)$ and R is $RemoveFirstCharacter(S)$.
- $GetAllSubstrings(L)$ is called and L is of length N. Those elements of the result not already equal to some element of C are added to C
- $GetRightSubstrings(R)$ is called, and those elements of the result not already equal to some element of C are added to C

As C contains all substrings of $RemoveFirstCharacter(S)$, all strings that are the result of any number of recursive applications of $RemoveLastCharacter$ to S, and S itself, $GetAllSubstrings$ gets all substrings of the input string S where S is of length $N + 1$ supposing $GetAllSubstrings(T)$ gets all substring of the input string T where T is of length N.

As $GetAllSubstrings(S)$ gets all substrings of S where S is of length 0 (meaning S is the empty string), then by induction $GetAllSubstrings(T)$ gets all substrings of T where T is of any length $N \in \mathbb{N}$.

Hence, $GetAllSubstrings(S)$ gets all substrings of any string S.

■

3.1.2 - Top-Down FOL Evaluation Algorithm

Definition

Input: Any finite parse tree P following the restrictions set in the section entitled 5.2.1 - The Grammar and Language of Queries[1], a finite set V_{Free} containing the free variables of P and a finite universe of strings U.

Output: Collection of solutions, where solutions are collections of assignments who satisfy P wherever V_{Free} is not empty and assignments are pairs $(v, u), v \in V_{Free}, u \in U$; $true$ wherever P represents a tautological query and V_{Free} is empty, and $false$ wherever P represents an unsatisfiable query, where V_{Free} represents the set of free variables in the query represented by P.

Let:
$Top(P)$ return the top node in P,
$GetChild(P)$ return the child of $Top(P)$ where $Top(P)$ is a quantification,
$LeftChild(P)$ and $RightChild(P)$ get the left child and right child respectively of $Top(P)$ respectively where $Top(P)$ is a binary logical operator,
$GetVar(Top(P))$, return the variable quantified at the top node in P,
and $Compose(phrase, A)$ return a string where for all $(v, u) \in A$ all unquoted occurrences of v are replaced with u.

$EvaluateTopDown$ function (arguments: P, V_{Free}, U)

1. Consider an empty set $Solutions$ and an empty set A
2. If V_{Free} is empty
    a. Return $Evaluate(P, U, A)$
3. Consider each possible solution A – that is, each possible set S such that $\forall v \in V_{Free}: \exists u \in U: (v, u) \in S \ \wedge (\nexists r \in U: u \neq r \ \wedge (v, r) \in S)$
    a. If $Evaluate(P, U, A)$ returns $true$
        i. Add S to $Solutions$
4. Return $Solutions$

$Evaluate$ function (arguments: P, U, A)

1. If $Top(P)$ is a quantification:
    a. Let $V \coloneqq GetVar(Top(P))$
    b. If $Top(P)$ is an existential quantification
        i. If $Evaluate(GetChild(P), U, A \cup \{(V, u)\})$ is satisfied for any value $u \in U$
            1. return $true$
        ii. Return $false$
    c. If $Top(P)$ is a universal quantification

[1] The presently relevant restrictions are that there must be no negations, relations, or functions inside the query, and that quantifiers come first, as in Prenex normal form.

i. If $Evaluate(GetChild(P), U, A \cup \{(V, u)\})$ is satisfied for all values $u \in U$
    1. return $true$
ii. Return $false$

2. If $Top(P)$ is a binary logical operator
    a. Let $\circ$ be the binary operator depicted by $Top(P)$
    b. Return $Evaluate(LeftChild(P), U, A) \circ Evaluate(RightChild(P), U, A)$
3. If $Top(P)$ is a term
    a. Let $\circ$ be the binary comparator of the term, $Left$ be the left-hand-side of the term and $Right$ be the right-hand-side of the term.
    b. Return $Compose(Left, A) \circ Compose(Right, A)$

Proof

Any parse tree subject to these restrictions has, as nodes, only logical operators, quantifications, and terms.

The $Evaluate$ Function

Case 1: $Top(P)$ is a term

- $Top(P)$ is not a quantification or a binary logical operator.
- $Top(P)$ is a term, so $\circ$ is the binary comparator, and Return $Compose(Left, U, A) \circ Compose(Right, U, A)$ is returned.

Hence, where $Top(P)$ is term, the left-hand-side and right-hand-side are compared according to the comparator after being composed. Therefore, $Evaluate(P)$ will always terminate and correctly evaluate terms.

Case 2: $Top(P)$ is a binary logical operator

- $Top(P)$ is not a quantification.
- $Top(P)$ is a binary logical operator, so $\circ$ is the binary operator, and $Evaluate(LeftChild(P), U, A) \circ Evaluate(RightChild(P), U, A)$ is returned.

Hence, as according to the restrictions on the queries that can be given the children of a binary logical operator can only be a term or a binary logical operator, it suffices when proving that these binary logical operators operate correctly to prove inductively that the operators are performed correctly, and the calls made to $Evaluate$ terminate.

Suppose both $LeftChild(P)$ and $RightChild(P)$ are terms (as is the only case where P has a depth of 1)– then both $Evaluate(LeftChild(P), U, A)$ and $Evaluate(RightChild(P), U, A)$ terminate and evaluate correctly, as both are under case 1.

Suppose P has a depth of $N + 1$, and that for all parse trees Q of depth N or less for whom $Top(P)$ is a binary logical operator, $Evaluate(P, U, A)$ terminates and evaluates correctly for a finite set of strings U and a finite set of pairs of strings A:

Then each of $Top(P)$'s children have at most a depth of N.

Then both $Evaluate(LeftChild(P), U, A)$ and $Evaluate(RightChild(P), U, A)$ terminate and evaluate correctly subject to those suppositions.

Then $Evaluate(P, U, A)$ terminates and evaluates correctly, subject to those suppositions.

Then, by induction, as $Evaluate(P, U, A)$ terminates and evaluates correctly wherever $Top(P)$ is a binary logical operator and P has a depth of 1, and does so where P has a depth of $N + 1$ so long as it does so where $Top(P)$ is a binary logical operator and P has a depth of N, $Evaluate(P, U, A)$ terminates and evaluates correctly where P has a depth $n \in \mathbb{N}, n > 1$, ($n = 1$ being here the case where P has no children, which cannot occur) wherever $Top(P)$ is a binary logical operator.

Case 3: $Top(P)$ is a quantification

Let $QuantifierDepth(P)$ denote the depth of the highest non-quantifier in P.

Case 3.1: $Top(P)$ is a universal quantification

- $Top(P)$ is a quantification, so which kind of quantification it is determines what's returned.
- $GetQuantifier\big(Top(P)\big)$ is a universal quantifier, so we return $true$ if $Evaluate(GetChild(P), U, A \cup \{(V, u)\})$ is satisfied for all values $u \in U$, and otherwise return $false$. This follows from the definition of the universal quantifier.

Where $GetChild(P)$ is a Boolean operator or is a term (as is always the case where $QuantifierDepth(P)$ is 0) then $Evaluate(GetChild(P), U, A \cup \{(V, u)\})$ terminates and evaluates correctly as it is in case 1 or is in case 2, and so if all values $u \in U$ have $Evaluate(GetChild(P), U, A \cup \{(V, u)\})$ return $true$, then $Evaluate(P, U, A)$ returns $true$, and $Evaluate(P, U, A)$ returns $false$ otherwise.

Then $Evaluate(P, U, A)$ terminates and evaluates correctly wherever $QuantifierDepth(P)$ returns 0.

Suppose $QuantifierDepth(P)$ is $N + 1$ and $Evaluate(Q, U, A)$ terminates and evaluates correctly wherever $QuantifierDepth(Q) \leq N$:

Then $Evaluate(GetChild(P), U, A \cup \{(V, u)\})$ terminates and evaluates correctly for all values $u \in U$.

Then as U is finite, $Evaluate(P, U, A)$ terminates and evaluates correctly, subject to those suppositions.

Then, by induction, as $Evaluate(P, U, A)$ terminates and evaluates correctly where $QuantifierDepth(P)$ is 0 and $Top(P)$ is a universal quantifier, and does so where $QuantifierDepth(P)$ is $N + 1$ and $Top(P)$ is a universal quantifier so long as it does so wherever $QuantifierDepth(P)$ is N and $Top(P)$ is a universal quantifier,

$Evaluate(P, U, A)$ terminates and evaluates correctly wherever $Top(P)$ is a universal quantifier.

Case 3.2: $Top(P)$ is an existential quantification

- $Top(P)$ is a quantification, so which kind of quantification it is determines what's returned.
- $GetQuantifier\big(Top(P)\big)$ is an existential quantifier, so we return $true$ if $Evaluate(GetChild(P), U, A \cup \{(V, u)\})$ is satisfied for any value $u \in U$, and otherwise return $false$. This follows from the definition of the existential quantifier.

Where $GetChild(P)$ is a Boolean operator or is a term (as is always the case where $QuantifierDepth(P)$ is 0) then $Evaluate(GetChild(P), U, A \cup \{(V, u)\})$ terminates and evaluates correctly as it is in case 1 or is in case 2, and so if any value $u \in U$ have $Evaluate(GetChild(P), U, A \cup \{(V, u)\})$ return $true$, then $Evaluate(P, U, A)$ returns $true$, and $Evaluate(P, U, A)$ returns $false$ otherwise.

Then $Evaluate(P, U, A)$ terminates and evaluates correctly wherever $QuantifierDepth(P)$ returns 0 and $Top(P)$ is an existential quantifier.

Suppose $QuantifierDepth(P)$ is $N + 1$ and $Evaluate(Q, U, A)$ terminates and evaluates correctly wherever $QuantifierDepth(Q) \leq N$:

Then $Evaluate(GetChild(P), U, A \cup \{(V, u)\})$ terminates and evaluates correctly for all values $u \in U$ where $Top(P)$ is an existential quantifier.

Then as U is finite, $Evaluate(P, U, A)$ terminates and evaluates correctly, subject to those suppositions.

Then, by induction, as $Evaluate(P, U, A)$ terminates and evaluates correctly where $QuantifierDepth(P)$ is 0 and $Top(P)$ is an existential quantifier, and does so where $QuantifierDepth(P)$ is $N + 1$ and $Top(P)$ is an existential quantifier so long as it does so wherever $QuantifierDepth(P)$ is N and $Top(P)$ is an existential quantifier, $Evaluate(P, U, A)$ terminates and evaluates correctly wherever $Top(P)$ is an existential quantifier.

Thus, in all cases, $Evaluate$ terminates and evaluates correctly, given appropriate inputs.

The *EvaluateTopDown* function

-Each possible solution A is considered

-If there are no possible solutions, $Evaluate(P, U, \emptyset)$ is returned and the function terminates, and so the returned value represents whether the query is a tautology

-Those who cause $Evaluate(P, U, A)$ to return true are added to the set $Solutions$

-solutions is returned.

Therefore, either a list of all satisfying solutions is returned, or if a Boolean representing whether P represents a tautology.

Therefore, $EvaluateTopDown$ functions as intended.

■

3.1.3 - Bottom-Up FOL Evaluation Algorithm

Definition

Input: Any finite parse tree P following the restrictions set in the section entitled 5.2.1 - The Grammar and Language of Queries[2], a finite set V_{Free} containing the free variables of P and a finite universe of strings U.

Output: Collection of solutions, where solutions are collections of assignments who satisfy P wherever V_{Free} is not empty and assignments are pairs $(v,u), v \in V_{Free}, u \in U$; $true$ wherever P represents a tautological query and V_{Free} is empty, and $false$ wherever P represents an unsatisfiable query, where V_{Free} represents the set of free variables in the query represented by P.

Let:
$EvaluateTopDown$ act as described in Top-Down FOL Evaluation Algorithm,
$Evaluate$ act as described in Top-Down FOL Evaluation Algorithm,
$RemoveVar(S,v)$ removes any assignment $(v,u), u \in U$ from all sets in S.
$Top(P)$ return the top node in P,
$GetChild(P)$ return the child of $Top(P)$ where $Top(P)$ is a quantification or a unary logical operator,
$LeftChild(P)$ and $RightChild(P)$ get the left child and right child respectively of $Top(P)$ respectively where $Top(P)$ is a binary logical operator,
$GetVar(Top(P))$ return the variable quantified at the top node in P,
$GetQuantifier(Top(P))$ return the quantifier at the top node of P,
and $Compose(phrase, A)$ return a string where for all $(v,u) \in A$ all unquoted occurrences of v are replaced with u.

$EvaluateBottomUp$ function (arguments: P, V_{Free}, U)

1. If $Top(P)$ is a term
    a. return $EvaluateTopDown(P, V_{Free}, U)$
2. If $Top(P)$ is a binary logical operator
    a. Let $L := EvaluateBottomUp(LeftChild(P), V_{Free},\ U),\ R := EvaluateBottomUp(RightChild(P), V_{Free},\ U)$
    b. If $Top(P)$ is a disjunction
        i. Return $L \cup R$
    c. If $Top(P)$ is a conjunction
        i. Return $L \cap R$
3. If $Top(P)$ is a quantification
    a. Let $var := GetVar(Top(P))$
    b. Let $S := EvaluateBottomUp(Child(P), V_{Free},\ U)$

[2] The presently relevant restrictions are that there must be no negations, relations, or functions inside the query, and that quantifiers come first, as in Prenex normal form.

c. If $Top(P)$ is an existential quantification
    i. Return $RemoveVar(S, var)$
d. If $Top(P)$ is a universal quantification
    i. Let $Q \coloneqq RemoveVar(S, var)$
    ii. Let $R \coloneqq \{q \mid q \in q, \forall val \in U: (q \cup \{(var, val)\}) \in S\}$
    iii. Return R

Proof

Case 1: $Top(P)$ is a term

- $EvaluateTopDown(P, V_{Free}, U)$ is returned

According to the proof of the Top-Down FOL Evaluation Algorithm, this terminates and evaluates correctly.

Case 2: $Top(P)$ is a binary logical operator

- L is the result of $EvaluateBottomUp(LeftChild(P), V_{Free}, U)$, R is the result of $EvaluateBottomUp(RightChild(P), V_{Free}, U)$

Case 2.1: $Top(P)$ is a disjunction

- The union of L and R is returned

The disjunction should return any solution who satisfies either the query represented by $LeftChild(P)$ or the query represented by $RightChild(P)$.

Suppose P has a depth of 2 (that is, P has no child that has children, and so according to the restrictions on the parse trees who can be entered, both P's children are terms).

Then L is the set of solutions who satisfy $LeftChild(P)$, and R is the set of solutions who satisfy $RightChild(P)$.

Then any solution either from L or R satisfies the disjunction of the query represented by $LeftChild(P)$ and the query represented by $RightChild(P)$.

Then $L \cup R$ is the set of solutions who satisfy P.

Then $EvaluateBottomUp$ terminates and evaluates correctly, subject to these suppositions.

Suppose P has a depth of $N + 1$ and $EvaluateBottomUp(P, V_{Free}, U)$ terminates and evaluates correctly wherever P has a depth less than or equal to N.

Then P has children of depths less than or equal to N.

Then L is the set of solutions who satisfy $LeftChild(P)$, and R is the set of solutions who satisfy $RightChild(P)$.

Then any solution either from L or R satisfies the disjunction of the query represented by $LeftChild(P)$ and the query represented by $RightChild(P)$.

Then $L \cup R$ is the set of solutions who satisfy P.

Then, by induction, as $BottomUpEvaluate(P, U, A)$ terminates and evaluates correctly wherever $Top(P)$ is a disjunction and P has a depth of 1, and does so where P has a depth of $N + 1$ so long as it does so where $Top(P)$ is a disjunction and P has a depth of N, $BottomUpEvaluate(P, U, A)$ terminates and evaluates correctly where P has a depth $n \in \mathbb{N}, n > 1$, ($n = 1$ being here the case where P has no children, which cannot occur) wherever $Top(P)$ is a disjunction.

Case 2.2: $Top(P)$ is a conjunction

- The union of L and R is returned

The disjunction should return any solution who satisfies either the query represented by $LeftChild(P)$ or the query represented by $RightChild(P)$.

Suppose P has a depth of 2 (that is, P has no child that has children, and so according to the restrictions on the parse trees who can be entered, both P's children are terms).

Then L is the set of solutions who satisfy $LeftChild(P)$, and R is the set of solutions who satisfy $RightChild(P)$.

Then any solution from both L and R satisfies the conjunction of the query represented by $LeftChild(P)$ and the query represented by $RightChild(P)$.

Then $L \cap R$ is the set of solutions who satisfy P.

Then $EvaluateBottomUp$ terminates and evaluates correctly, subject to these suppositions.

Suppose P has a depth of $N + 1$ and $EvaluateBottomUp(P, V_{Free}, U)$ terminates and evaluates correctly wherever P has a depth less than or equal to N.

Then P has children of depths less than or equal to N.

Then L is the set of solutions who satisfy $LeftChild(P)$, and R is the set of solutions who satisfy $RightChild(P)$.

Then any solution either from L or R satisfies the disjunction of the query represented by $LeftChild(P)$ and the query represented by $RightChild(P)$.

Then $L \cap R$ is the set of solutions who satisfy P.

Then, by induction, as $BottomUpEvaluate(P, U, A)$ terminates and evaluates correctly wherever $Top(P)$ is a conjunction and P has a depth of 1, and does so where P has a depth of $N + 1$ so long as it does so where $Top(P)$ is a conjunction and P has a depth of N, $BottomUpEvaluate(P, U, A)$ terminates and evaluates correctly where P has a depth $n \in \mathbb{N}, n > 1$, ($n = 1$ being here the case where P has no children, which cannot occur) wherever $Top(P)$ is a conjunction.

Case 3: $Top(P)$ is a quantification

Let $QuantifierDepth(P)$ denote the depth of the highest non-quantifier in P.

- var is the result of $GetVar(Top(P))$ and S is the result of $EvaluateBottomUp(Child(P), V_{Free},\ U)$

Case 3.1: $Top(P)$ is an existential quantification

- $RemoveVar(S, var)$ is returned.

Suppose $Child(P)$ is a term or a binary logical operator, then $QuantifierDepth(P)$ is 1.

S is the result of $EvaluateBottomUp(Child(P), V_{Free},\ U)$.

Then S is the set of solutions who satisfy the query represented by $Child(P)$.

Then S is the set of solutions who satisfy the query quantified by the quantifier $Top(P)$.

All solutions in S contain an assignment $(var, u), u \in U$.

Then all $s \in S$ demonstrate a set of assignments where there exists an assignment $(var, u), u \in U$.

Then $RemoveVar(S, var)$ is the set of solutions who satisfy the query P, subject to those suppositions.

Then $EvaluateBottomUp(P, V_{Free},\ U)$ terminates and evaluates correctly, subject to those suppositions.

Suppose $QuantifierDepth(P)$ is $N + 1$, and $EvaluateBottomUp(P, V_{Free}, U)$ terminates and evaluates correctly wherever $QuantifierDepth(Q) \leq N$.

Then $EvaluateBottomUp(Child(P), V_{Free},\ U)$ terminates and evaluates correctly and S is the result of $EvaluateBottomUp(Child(P), V_{Free},\ U)$.

Then S is the set of solutions who satisfy the query represented by $Child(P)$.

Then all solutions $s \in S$ satisfy $s == p \cup \{(var, val)\}$ for some $val \in U$ and some $p \in RemoveVar(S, var)$, and no solution t satisfies the query represented by P such that $t \notin S$.

Then $RemoveVar(S, var)$ is the set of solutions who satisfy the query P, subject to those assumptions.

Then $EvaluateBottomUp(P, V_{Free},\ U)$ terminates and evaluates correctly, subject to those suppositions.

Then, by induction, as $EvaluateBottomUp(P, U, A)$ terminates and evaluates correctly where $QuantifierDepth(P)$ is 0 and $Top(P)$ is an existential quantifier, and does so where $QuantifierDepth(P)$ is $N + 1$ and $Top(P)$ is an existential quantifier so long as it does so wherever $QuantifierDepth(P)$ is N and $Top(P)$ is an existential quantifier,

$EvaluateBottomUp(P, U, A)$ terminates and evaluates correctly wherever $Top(P)$ is an existential quantifier.

Case 3.2: $Top(P)$ is a universal quantification

- Q is the result of $RemoveVar(S, var)$
- R is the set of solutions q in Q where $\forall val \in U: (q \cup \{(var, val)\}) \in S$
- R is returned

Suppose $Child(P)$ is a term or a binary logical operator, then $QuantifierDepth(P)$ is 1.

S is the result of $EvaluateBottomUp(Child(P), V_{Free},\ U)$.

Then S is the set of solutions who satisfy the query represented by $Child(P)$.

Then S is the set of solutions who satisfy the query quantified by the quantification $Top(P)$.

Then R is the set of solutions $q \in Q$ where $q \cup \{(var, val)\} \in S$ for all $val \in U$.

Then R is the set of solutions who, for all possible values of val, satisfy $Child(P)$.

Then R is the set of solutions who satisfy the query P, subject to those suppositions.

Then $EvaluateBottomUp(P, V_{Free},\ U)$ terminates and evaluates correctly, subject to those suppositions.

Suppose $QuantifierDepth(P)$ is $N + 1$, and $EvaluateBottomUp(P, V_{Free}, U)$ terminates and evaluates correctly wherever $QuantifierDepth(Q) \leq N$.

Then $EvaluateBottomUp(Child(P), V_{Free},\ U)$ terminates and evaluates correctly and S is the result of $EvaluateBottomUp(Child(P), V_{Free},\ U)$.

Then S is the set of solutions who satisfy the query represented by $Child(P)$.

Then R is the set of solutions $q \in Q$ where $q \cup \{(var, val)\} \in S$ for all $val \in U$.

Then R is the set of solutions who, for all possible values of val, satisfy $Child(P)$.

Then R is the set of solutions who satisfy P.

Then $EvaluateBottomUp(P, V_{Free},\ U)$ terminates and evaluates correctly, subject to those suppositions.

Then, by induction, as $EvaluateBottomUp(P, U, A)$ terminates and evaluates correctly where $QuantifierDepth(P)$ is 0 and $Top(P)$ is an universal quantifier, and does so where $QuantifierDepth(P)$ is $N + 1$ and $Top(P)$ is an universal quantifier so long as it does so wherever $QuantifierDepth(P)$ is N and $Top(P)$ is an universal quantifier, $EvaluateBottomUp(P, U, A)$ terminates and evaluates correctly wherever $Top(P)$ is an universal quantifier.

Therefore, $EvaluateBottomUp$ functions as intended.

■

4 - Planning

As mentioned in the project description, the scope of the project is thus:

- To create and implement a language capable of modelling propositions about the composition of strings
- To create a parser for said language
- To create an evaluator for said parser's result
- To optimise said evaluator to a reasonable extent.

Creating such a language should prove easy – simply taking the form of Propositional Formulae and tweaking it for easier computation by inserting tokens to separate different parts of the statement. It follows from a discussion with my project supervisor that Prenex Normal Form is an appropriate standard to aide in ease of parsing but could lead to reckless expense of computational resources. Despite this drawback, I consider it to be a satisfactory starting point for now.

During the preliminary phases of planning the project I developed an algorithm that efficiently enumerates all sets of consecutive characters within a given string (consecutive substrings), which would be used to define the set of possible strings that any variables within queries could take.

The following is the plan presented in the project brief:

- Build formula representation/model
- Write text-to-model formula parser
- Write substring enumerator (with formal proof)
- Write top-down evaluator (with formal proof)
- Write bottom-up evaluator (with formal proof)
- Compare top-down and bottom-up evaluators' runtime and space efficiencies in different scenarios
- Have final evaluator run on user-entered queries
- Final functional testing and code clean-up

5 - Implementation

5.1 - Substring Enumerator

I began by coding the substring enumeration algorithm, since it seemed to be an easy way to make a dent in the project. This progressed quickly and easily, with only minor errors which were easily and quickly fixed.

Said enumerator works by considering the string as a whole, before dismissing either the leftmost or the rightmost character. If it dismisses the leftmost character, it does so repeatedly until no characters remain. If it dismisses the rightmost character, it repeats the algorithm recursively, treating the new truncated string as it did the original. Finally, it considers the empty string.

Currently, these results are stored in a list since lists are effective for the purpose of testing the algorithm. I intended to change this so results are stored in a tree, since it would be possible to make the enumeration algorithm give a maximally correct result faster than using a list and searching it for repetitions, wherever any part of the string to be searched is repeated (such as in the string "abcdefbcdgh", in which the phrase "bcd" is repeated despite the fact that it and its substrings need only be considered once). Since this is a minor optimisation, it is not implemented.

5.2 - Parser

5.2.1 - The Grammar and Language of Queries

Queries are in a modified Prenex normal form, designed for ease of parsing, without hindering the possibility of modification to a grammar designed for ease of user input.

Queries start with a function identifier and list of free variables, in the form $f(a, b, c \ldots)$ where f is the name of the function (though recursive functions are not allowed, making the function name currently useless) and a, b, c and so on are free variables.

This is always followed by a definition marker of the form $:=$ to distinctly separate the query from the function identifier and free variables.

The query should consist of non-negated quantifications (where "E" represents the existential quantifier and "A" represents the universal quantifier) followed by colons, either grouped (as in $Ex, y, z:$) or ungrouped (as in $Ex:Ey:Ez:$), before a final colon separating the query's prefix from its matrix. This is followed by conjunctions and disjunctions (where "*" represents the conjunction operator and "+" represents the disjunction operator) of any non-zero number of terms, where brackets are not necessary except to indicate that a disjunction should take precedence over a conjunction but may be included wherever the user finds them useful. Terms should be wrapped in square brackets (as in $[S\,!= A.B]$) and should consist of concatenations of any nonzero number of strings and variable names to both the left and the right of one comparator. Some example queries follow:

$$f(a, b) := Ex, y: : [S == x.a.y] * [b == a.a]$$

$$f(a) := Az : Ey : : ([S == z.y] + [S == y.z]) * [a >= z]$$

It should be noted that spaces may be placed anywhere in the query outside of strings to ease writing the query, but will be immediately removed by the parser before parsing – that is, spaces outside of speech marks make no difference to the query. Two examples follow that will give identical results in all cases:

$$f \quad (\quad a \quad) \quad := \quad [S = \quad = " \ " \quad . \quad a \quad]$$

$$f(a) := [S == " \ ".a]$$

To allow for this, variable names cannot contain spaces.

It should also be noted that queries contain no negations, and so any need for negations should be handled by using logical identities to push the negation into a term and using the opposite comparator in said term to do so.

As such, the negation of the query $f() := Ex : [S == " \ ".x]$ could be represented as $f() := Ax : [S \ = ! = " \ ".x]$, or equivalent.

To include a backslash or a speech-mark in a string, place a backslash before it, as in the following example query:

$$f(abcdefg) := Eabc : : [abcdefg == "\\\"\\apple".abc]$$

Here, the string being concatenated with the variable abc is parsed as "\"\$apple$".

Additionally, brackets should not surround any quantifiers. Hence, $f() := Ex, y : [S == x."a".y]$ is a valid query, but $f() := Ex : (y : [S == x."a".y])$ is not.

Finally, variables should not be named "S", nor should their names contain the following characters:

- . (period)
- , (comma)
- " (speech marks)
- ' (quote mark/apostrophe)
- = (equals sign)
- > (greater-than sign)
- < (less-than sign)
- [(open square-bracket)
-] (close square-bracket)
- + (plus sign)
- * (asterisk)
- ((open bracket)
-) (close bracket)
- : (colon)

- (space character)
- \ (backslash)

Other unmentioned special characters may cause issue.

5.2.2 - Parsing Queries

I decided upon a good starting model for a language for queries – essentially the form of Propositional Formulae with some minor tweaks for ease of parsing, including the limitation that all formulae should be in Prenex normal form, which separates existential quantifiers from the rest of the statement. This restriction should be easy enough to lift retroactively by widening the recursion of the parsing algorithm to include the section which parses quantifiers but is thus far a satisfactory start.

Though I attempted to utilise a parser generator [10] [11] [12] to ease writing a parser and modifying its grammar, I found this to be unfortunately difficult since most parser generators I found met my needs well were difficult to utilise within c# - the language I'm writing the project in. After some deliberation, I erred on the side of caution with the January deadline in mind and changed plans to write a parser manually instead. Though this proved time consuming, in this case clearer and more definitive progress proved a harbinger of success.

With control of the language used to specify the queries, I could require useful marker characters to denote the edges of various parts of the code, most useful in parsing terms. Due to the Prenex normal form, I found I was able to split the queries into three parts – the signature, the quantifiers and the proposition – which helped chunk the problem, and I decided that the representation of the queries would reflect this until such a time that the Prenex normal form restriction was removed.

Later, I would come back to have the parser decompose the terms of a query into a datatype involving two sides (arrays of strings containing atoms) and a comparator.

The parser, after lexing the query, replaces quantifications that are grouped together as a shorthand into distinctly separate quantifications (a parallel to "$\exists x, y$:" being replaced with "$\exists x: \exists y$:"). The parser then moves on to handle the side of the query that represents the matrix [14], leaving the prefix of quantifiers to handle later.

The parser recurses on the query, splitting it at any logical conjunctions or disjunctions that exist outside of any brackets. If brackets surround the whole remaining query, they are removed. If the remaining query is only one lexicon, it is concluded to be a term and is parsed as such.

When parsing a term, the first unquoted index of a comparator's symbol and split the string into those characters before the comparator and after it, calling these the right-hand-side and left-hand-side respectively. The term is then defined as a left-hand-side string, a comparator, and a right-hand-side string, is wrapped in an otherwise null operator so it can be placed in the parse tree.

Once the matrix is parsed, it in turn is wrapped inside an otherwise null quantifier, and placed in the parse tree below the series of quantifiers, and the parse tree is returned.

5.3 - Evaluator

5.3.1 - Term Evaluation

A specific static class was built to handle the evaluation of terms, which would give a Boolean result given a term and a list of variable assignments. It would do this by composing the strings on the left and right sides of the term, before comparing them according to the term's comparator.

The comparators are restricted to the following list, and custom relations cannot be defined:

Name	Symbol	Description
Identical	"=="	returns true if and only if the left string reads the same as the right string
Not Identical	"!==" or "==!" or "=!="	returns true if and only i f the left string reads differently to the right string
Lesser	"<"	returns true if and only if the left string has strictly fewer characters than the right string
Lesser Or Equal	"<=" or "=<"	returns true if and only if the left string has the same or fewer characters than the right string
Greater	">"	returns true if and only if the left string has strictly more characters than the right string
Greater Or Equal	">=" or "=>"	returns true if and only if the left string has the same or more characters than the right string
Equal	"="	returns true if and only if the left string has the same number of characters as the right string

The evaluator models the comparators on a case-by-case basis, and the string composer allows for string concatenation and has escape characters to allow for quote marks in strings.

5.3.2 - Top-Down Approach

I began work on the top-down evaluator without regard to free variables, with an intent to build them in later. The evaluator would take a quantified expression and consider the quantifier used and the variable quantified. It would loop through all possible values for the variable, as determined by the substring generator, stopping when it found one that gives an answer to the quantifier (that is, a value that satisfies the quantified expression if the quantifier is existential, but is a value that does not satisfy the quantified expression if the

quantifier is universal), and returns the appropriate truth value. If none are found, then the opposite truth value is returned.

Propositional logic operators were restricted to * (representing logical AND) and + (representing logical OR) whose symbols were decided on with regards to softly typed languages' approach to mathematical operations applied to Booleans and Boolean evaluation of integers. The evaluator simply considered the left operand and the right operand separately and returned the logical operator's result when applied to the two.

Terms were evaluated using the Term Evaluator, substituting all the variables with their values as provided by the top-down evaluator's enumeration through the substrings of the body of text to be searched, both for quantified and free variables in their own manners. A variety of functions were included for a term to provide, including reference to a text's length using mathematical comparators, and reference to its content using a double equals sign (==). To aide with the ability for the user to build a fixed string into the query and the use of special characters in them, the terms were later divided into atoms, which involved developing a method of escaping special characters and finding the first "unquoted" instance of particular characters in strings with escaped quotes being taken into consideration.

5.3.3 - Bottom-Up Approach

After some initial testing to make sure the top-down quantifier was working in at least the basic use cases, I worked on the bottom-up quantifier. This would work by exploring the parse tree with a depth-first approach and evaluating terms as though their variables were all free (with the exception of the variable representing the body of text to be searched, denoted S), calling the top-down evaluator's method to do so, and returning the resulting set of solutions. It would consider logical operators very differently – asking simply if a solution exists in the results from both the left-hand and the right-hand sides for a conjunction, but potentially creating a new solution for every pair of left and right solutions for a disjunction. In this way, a conjunction acts like a filter for solutions where a disjunction creates more.

Quantifiers work differently from each other also – the existential quantifier simply filtering out solutions without values for a variable, where the universal quantifier performs a much more complicated procedure: starting by creating a copy of the list of solutions, stripping the quantified variable from their assignments and removing any duplicates, then accepting those solutions for whom adding an assignment for the quantified variable results in a solution from the original list from all values.

To reduce memory usage, variables aren't included in solutions until they're relevant (that is, the variables in the solution at a node in the parse tree are those who are free variables in its subquery). This means that the user-inputted list of free variables is only considered at the end of evaluation, where those free variables not used in the results are finally included before the evaluator returns the result.

5.4 - User Interface

With the evaluators coded and tested to ensure they work correctly in basic cases, I made a UI to circumvent the need to run the program every time I wanted to test something, to ease both query and search material entry for both me (during testing) and the end user (during usage), and to display the results in an easily readable manner.

The query entry textbox is a simple, single-line textbox to prevent the user from entering new-lines and keep the interface clean. The search material entry textbox is a rich text box, to allow for multiline entry of text.

A button for top-down evaluation is present, and a button for bottom-up evaluation is present. When clicked, the evaluator specified by the button will query the material as specified. A message box will appear displaying the time it took to parse and evaluate everything before the results are displayed.

If the query has free variables, the resulting solutions are detailed in a data grid [15] to the right of the text boxes. Otherwise, a message box will appear telling the user if the query is a tautology or is unsatisfiable.

Finally, a message box will appear displaying the time it took to display the results.

The user is free to repeat the process as they like.

Should the user enter search material or a query that can't be parsed or evaluated for whatever reason, a message box will appear advising how to remediate the problem should it be one I've predicted.

5.6 - Development Diary

A table of bugs encountered, their solutions and iterative modifications to the program detailing its development follows.

Bug Description	Solution
Iteration 1: Development of Parser	
Infinite loop parsing expressions found	Incrementation of counter variable used to iterate through the lexed expression symbols.
Iteration 2: Initial development of Top-Down Evaluator	
Left-hand-side of terms is a character too short, leading to incorrect evaluations	Changed the length of substring getting left-hand-side of terms.
Identical and Equal operators not successfully distinguished by parser	Changed the conditions of the selection statement selecting the comparator for a term
Iteration 3: Free variables accounted for in Top-Down Evaluator	
Infinite loop occurring in Top-Down Evaluator where values for free variables are considered	Process for iteration through values for variables corrected.
Free variables incorrectly parsed when more than one is present	Parser's splitting of the string containing the free variables corrected.
Parser fails when query contains no quantifiers	Parser changes to default to parsing the logical expression when no quantifier is present.
Priority of operations in parse tree is mismatched with query	Fixed parsing priority order of operations.
Iteration 4: Term class developed	
Parser not parsing terms into classes correctly.	Created separate function for parsing terms in parser. Cleaner code was more robust.
Iteration 5: Escape character added	
Infinite loop occurs while identifying escaped speech marks	Incrementor in loop added.
Iteration 6: Modified Term class (required escape character functionality)	
Terms' left and right sides missing first atom	Parser corrected.
Iteration 7: Assignment class and Bottom-Up Evaluator developed	
Method to get free variables from solution fails when solution is empty	Made an edge case to deal with this.
Only bound variables shown in solution	Changed "not equal" to "equal" in method that removed constrained variables based on names.
Iteration 8: UI Implemented	
Bottom-up proving disproportionately slower	Prevented "S" (the variable representing the search material) from appearing in free variable list.

than Top-Down	
Solution comparison is giving false negatives	Overrode Equals() function for Assignment class.
Universal quantifier is incorrectly modelled in bottom-up evaluation.	Completely revamped how universal quantifiers are handled by the bottom-up evaluator to a model more closely depicting the Bottom-Up FOL Evaluation Algorithm depicted in the *Proofs* section.
Lists of solutions were not correctly being cloned, and instead were being referenced	Used Linq's ConvertAll() function to simulate making an entirely different list, as no explicit deep-clone function was inbuilt.
Linq's Distinct() operator not filtering out replica	Wrote a different function (and an overload) to make sure solutions and assignments were distinct [16] [17] [18].
Comparators of length 3 (eg. "=!=") leading to incorrect parsing of right-hand-side of terms	Corrected substring start index to be 1 greater in these cases and cleaned up comparator selection statement.

7 - Evaluation

The following is a table containing the goals depicted in the section 4 - Planning, detailing their completeness.

- Build formula representation/model
- Write text-to-model formula parser
- Write substring enumerator (with formal proof)
- Write top-down evaluator (with formal proof)
- Write bottom-up evaluator (with formal proof)
- Compare top-down and bottom-up evaluators' runtime and space efficiencies in different scenarios
- Have final evaluator run on user-entered queries
- Final functional testing and code clean-up

Goal	Completeness
Build formula representation/model	✓
Write text-to-model formula parser	✓
Write substring enumerator (with formal proof)	✓
Write top-down evaluator (with formal proof)	✓
Write bottom-up evaluation (with formal proof)	✓
Compare top-down and bottom-up evaluators' runtime and space efficiencies in different scenarios	✗
Have final evaluator run on user-entered queries.	✓
Final functional testing and code clean-up	✓

Further development would likely involve minor refactoring before the implementation of a method that estimates which of the two evaluators would be more efficient when dealing with the current query and search material (according to the user's preference of time or memory efficiency), alongside further testing.

8 - Bibliography

[1] D. D. Freydenberger and L. Peterfreund, “The theory of concatenation over finite models,” 18 February 2020. [Online]. Available: arXiv:1912.06110 [cs.LO]. [Accessed 15 April 2020].

[2] T. Tarczynski, “Heap class in .NET [duplicate],” Stack Overflow, 23 May 2017. [Online]. Available: https://stackoverflow.com/questions/2231796/heap-class-in-net.

[3] M. Cochran, “C# Heap(ing) Vs Stack(ing) In .NET - Part One,” C# Corner, 2019. [Online]. Available: https://www.c-sharpcorner.com/article/C-Sharp-heaping-vs-stacking-in-net-part-i/.

[4] “Binary Heap,” Wikipedia, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Binary_heap.

[5] “Binary Search Tree,” Wikipedia, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Binary_search_tree.

[6] “Heap (data structure),” Wikipedia, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Heap_(data_structure).

[7] “Hash Table,” Wikipedia, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Hash_table.

[8] “B-tree,” Wikipedia, 2019. [Online]. Available: https://en.wikipedia.org/wiki/B-tree.

[9] “7.6. Parse Tree,” Runestone Academy, 2015. [Online]. Available: https://runestone.academy/runestone/books/published/pythonds/Trees/ParseTree.html.

[10] J. HasaniH, “What is a good C# compiler-compiler/parser generator?,” 3 August 2015. [Online]. Available: https://stackoverflow.com/questions/1194584/what-is-a-good-c-sharp-compiler-compiler-parser-generator.

[11] F. Tomassetti, “The ANTLR Mega Tutorial,” 2017. [Online]. Available: https://convertkit.s3.amazonaws.com/landing_pages/incentives/000/375/965/original/ANTLR_Mega_Tutorial.pdf?1566483883.

[12] G. Tomassetti, “The ANTLR Mega Tutorial,” tomassetti.me, 8 March 2017. [Online]. Available: https://tomassetti.me/antlr-mega-tutorial/.

[13] T. Parr, “Antlr,” Antlr, 2014. [Online]. Available: https://www.antlr.org/.

[14] “Prenex normal form,” Wikipedia, 15 March 2020. [Online]. Available: https://en.wikipedia.org/wiki/Prenex_normal_form. [Accessed 25 April 2020].

[15] “DataGridView.Rows Property,” Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.datagridview.rows?redirectedfrom=MSDN&view=netframework-4.8#System_Windows_Forms_DataGridView_Rows. [Accessed 04 2020].

[16] “Enumerable.Distinct Method,” Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.distinct?view=netframework-4.8. [Accessed 04 2020].

[17] C. D. Souza, “How to Remove Duplicates from a C# List,” 06 07 2018. [Online]. Available: https://carldesouza.com/how-to-remove-duplicates-from-a-c-list/. [Accessed 04 2020].

[18] “How create a new deep copy (clone) of a List<T>?,” Stack Overflow, 22 December 2012. [Online]. Available: https://stackoverflow.com/questions/14007405/how-create-a-new-deep-copy-clone-of-a-listt. [Accessed 04 2020].