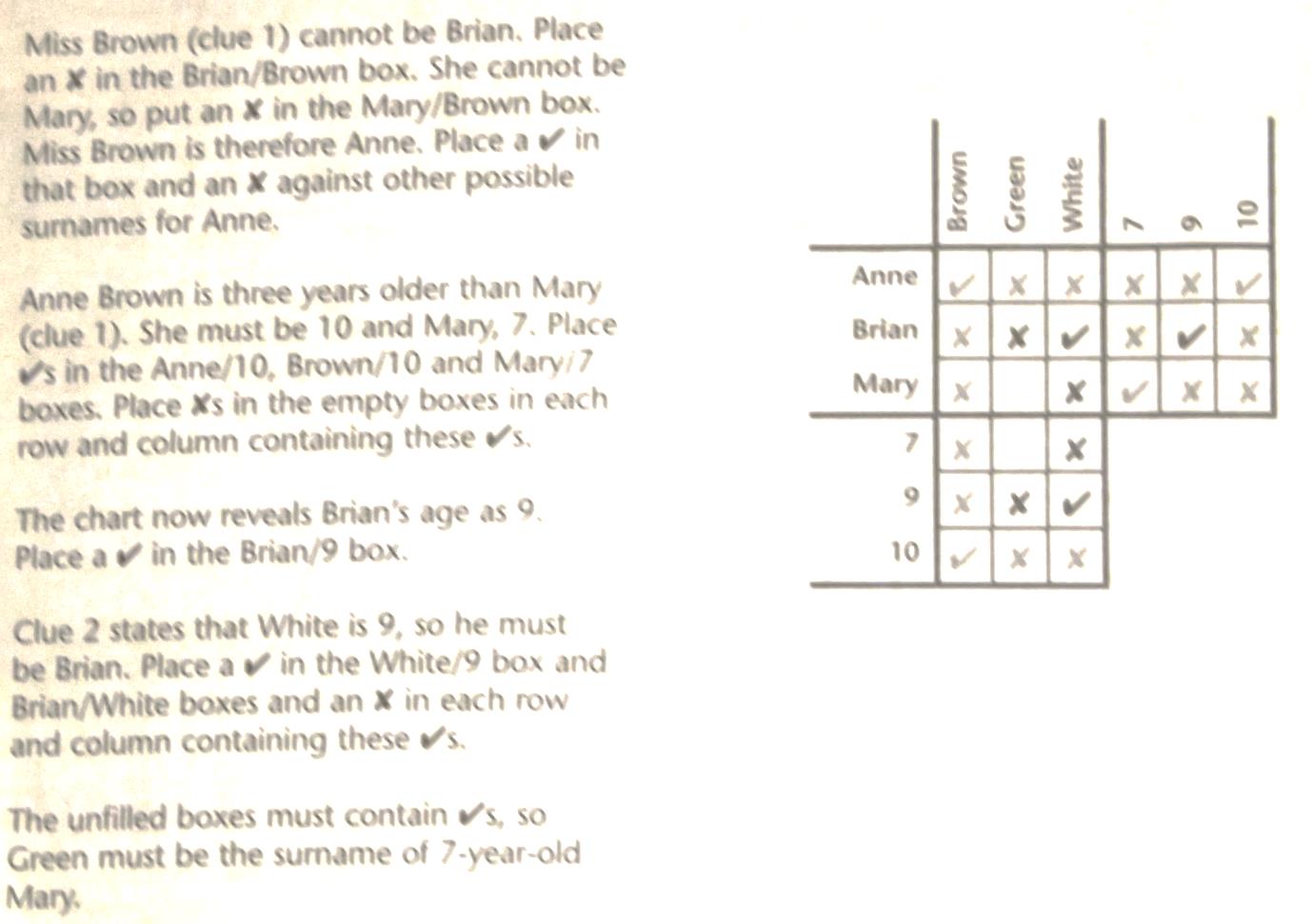
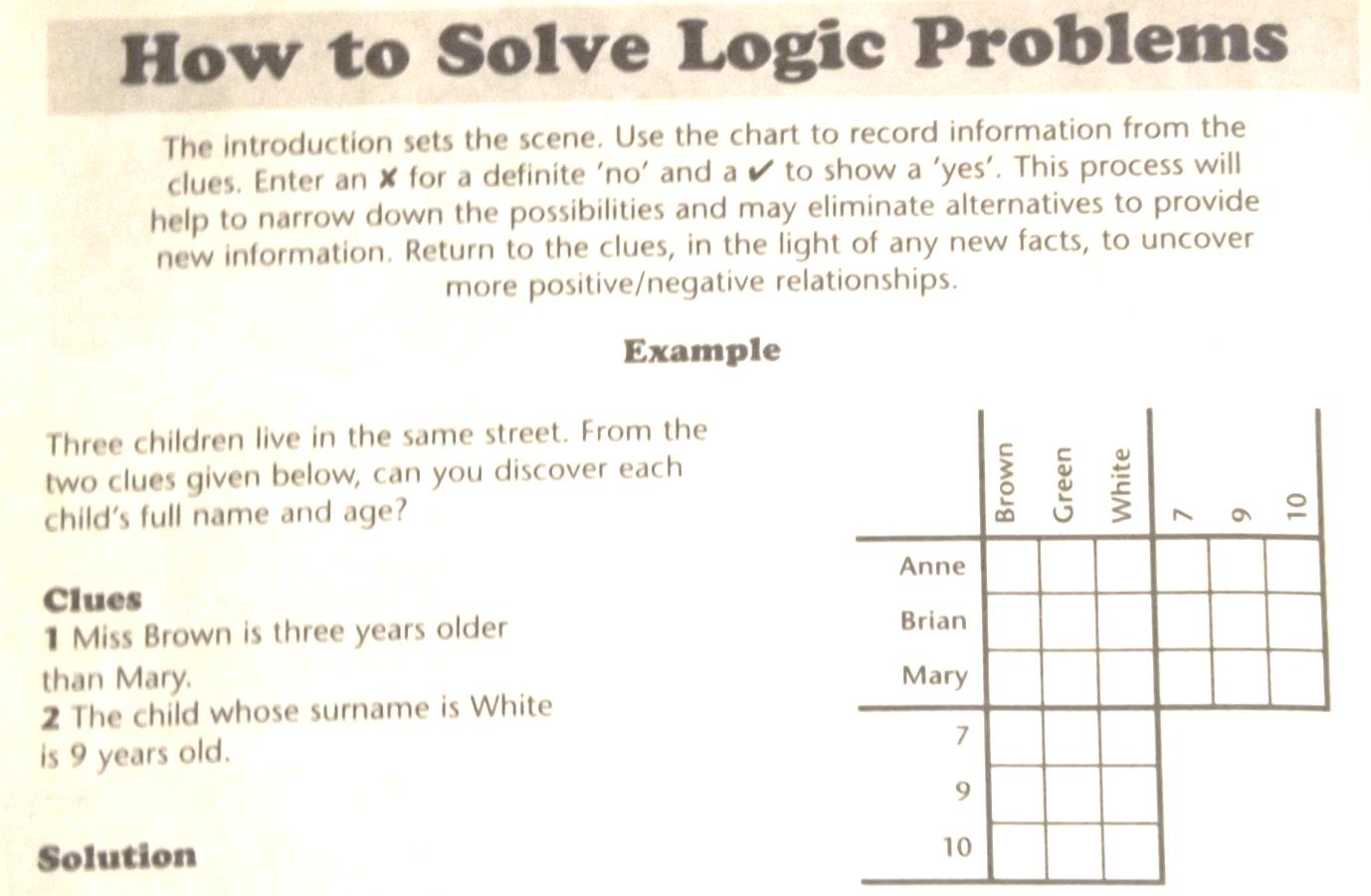
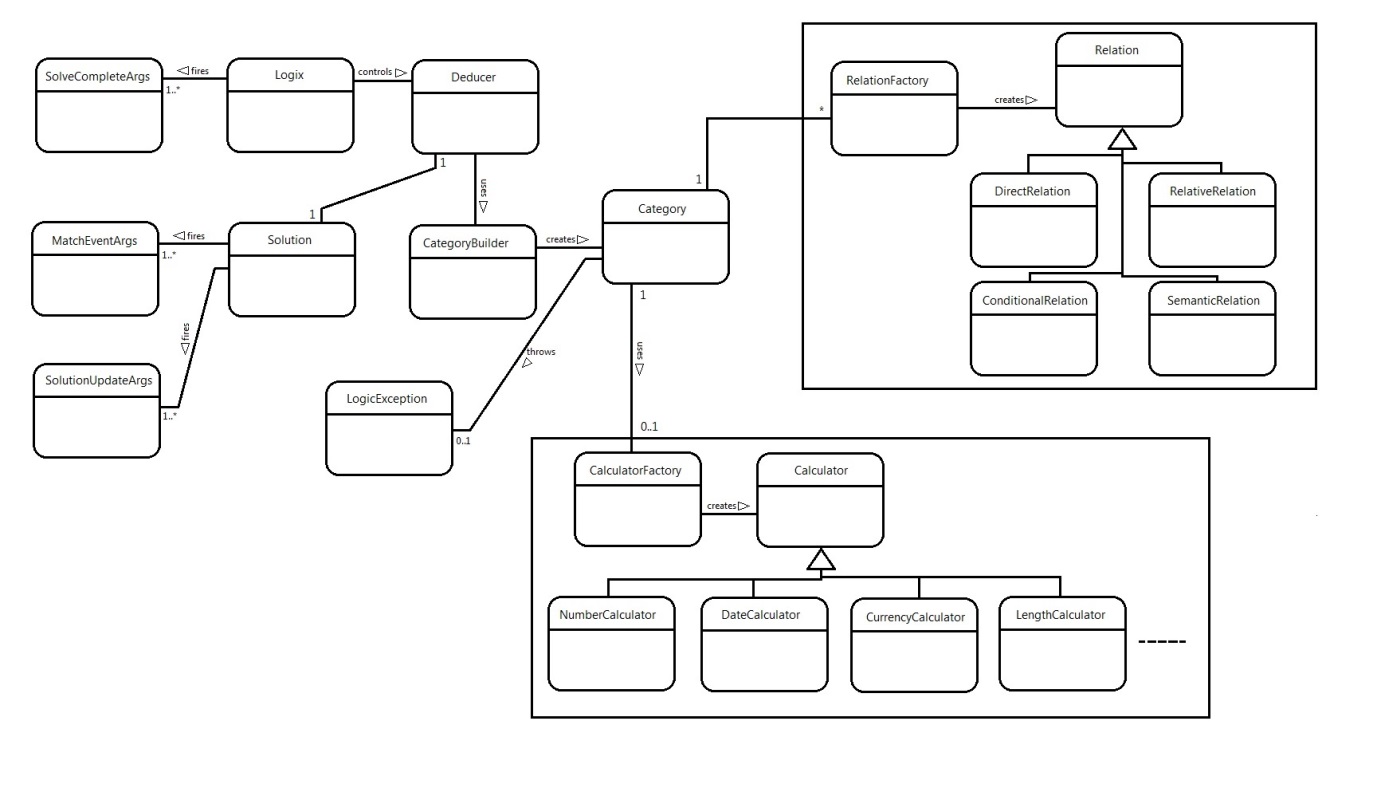
# Appendices

Appendix A – Original *Logic Problems* teaching problem taken from May edition [6]



Appendix B – Code Structure

1. Logix Class Diagram



1. Parser – TermsDictionary

using System;

using System.Collections.Generic;

using System.Linq;

namespace CZParser

{

public class TermsDictionary

{

internal List<string> disassociatives;

//e.g. "not"; signifies negative relationship.

internal List<string> numbers;

internal List<List<string>> quantifiers;

//e.g. "days"; signifies the unit in a comparative relationship.

internal List<string> prepositions;

//e.g. "before"; signifies direction of comparative relationship.

Stored in opposite pairs, with - before +.

internal List<string> superlatives;

internal char? currency;

private static string OF = "To";

private static string WITH = "Tw";

private static string AND = "Ta";

private static string EITHER = "Te";

private static string BUT = "Tb";

private static string NEGATIVE = "Td";

private static string SUPER = "Ts";

private static string FORMER = "Tf";

private static string LATTER = "Tl";

private static string THIS = "Tt";

private static string THAN = "Th";

public TermsDictionary(string[] keywords, char? currency = null) {

this.currency = currency;

disassociatives = setupStandardDisassociatives();

quantifiers = new List<List<string>>();

numbers = new List<string>();

prepositions = new List<string>();

for (int i = 0; i < keywords.Count(); i++) {

addQuantifiers(keywords[i]);

addSuperlatives(keywords[i]);

if (!string.IsNullOrEmpty(keywords[i])) {

numbers.AddRange(getNumericTermsForKey(keywords[i]));

setStandardPrepositions();

prepositions.AddRange(getPrepositionsForKey(keywords[i])

?? new List<string>());

}

}

}

private void setStandardPrepositions() {

if (prepositions.Count == 0) {

prepositions.AddRange(new List<string> { "before", "after", "less", "greater",

"fewer", "more", "lower", "higher", "behind", "ahead", "rear",

"front", "begins", "ends", "beginning", "ending" });

}

}

private void addQuantifiers(string key) {

quantifiers.Add(getQuantifiersForKey(key));

}

private List<string> setupStandardSuperlatives() {

return new List<string>();

}

private List<string> setupStandardDisassociatives() {

return new List<string> {"neither", "nor", "not", "isn't", "doesn't", "wasn't",

"didn't", "never" };

}

private List<string> getQuantifiersForKey(string key)

{

if (string.IsNullOrEmpty(key)) {

return new List<string>() { };

}

switch (key) {

case "left-right": return new List<string> { "left", "right" };

case "days": return new List<string> { "day", "days", "night", "nights" };

case "months": return new List<string> { "month", "months" };

case "years": return new List<string> { "year", "years" };

case "numeric": return new List<string> { "times", "twice", "half", "double",

"quarter" };

case "currency": return new List<string> { "pounds", "dollars", "euros",

"money", "cash", "wealth", "half", "double", "quarter",

"twice" };

case "dates": return new List<string> { "day", "days", "week", "weeks" };

case "time": return new List<string> { "hour", "hours" };

case "ordinals": return new List<string>() { };

case "alphabet":

case "length" :

case "words": return new List<string> { "letter", "word" };

default:

throw new ArgumentException("Keyword not recognised: " + key);

}

}

private void addSuperlatives(string key) {

if (string.IsNullOrEmpty(key)) {

return;

}

if (superlatives == null) {

superlatives = setupStandardSuperlatives();

}

switch (key) {

case "time":

case "days":

case "months":

case "years":

case "ordinals": superlatives.AddRange(new List<string> { "earliest",

"latest", "first", "last" }); return;

case "numeric": superlatives.AddRange(new List<string> { "lowest",

"highest" }); return;

}

}

private List<string> getPrepositionsForKey(string key) {

switch (key) {

case "dates":

case "time":

case "days":

case "months":

case "ordinals":

case "alphabet":

case "years": return new List<string> { "earlier", "later" };

case "currency": return new List<string> { "cheap", "expensive", "cheaper",

"dearer", "economical", "costly" };

case "numeric":

case "left-right": return null;

case "length":

case "words": return new List<string> { "shorter", "longer" };

default:

throw new ArgumentException("Keyword not recognised: " + key);

}

}

private List<string> getNumericTermsForKey(string key) {

switch (key) {

case "left-right":

case "days":

case "months":

case "years":

case "currency": return new List<string> { "1", "2", "3", "4", "5", "6", "7",

"8", "9", "10", "15", "20", "50", "100", "one", "two", "three",

"four", "five", "six", "seven", "eight", "nine", "ten", "eleven",

"twelve", "thirteen", "fourteen", "fifteen", "hundred",

"thousand", "million", "billion" };

case "words":

case "length":

case "numeric":

case "dates":

case "ordinals":

case "alphabet":

case "time": return new List<string> { "one", "two", "three", "four", "five",

"six", "seven", "eight", "nine", "ten", "eleven", "twelve",

"thirteen", "fourteen", "fifteen", "sixteen", "seventeen",

"eighteen", "nineteen", "twenty" };

default:

throw new ArgumentException("Keyword not recognised: " + key);

}

}

public List<string> getQuantifiers() {

List<string> quants = new List<string>();

foreach (List<string> list in quantifiers) {

quants.AddRange(list);

}

return quants;

}

internal string defineItem(string word) {

try {

double result;

if (double.TryParse(word, out result)) {

return word; //All numbers should be kept.

}

if (word == "of") {

return OF;

}

if (word == "with") {

return WITH;

}

if (word == "and") {

return AND;

}

if (word == "either") {

return EITHER;

}

if (word == "or") {

return EITHER;

}

if (word == "but") {

return BUT;

}

if (word == "this") {

return THIS;

}

if (word == "than") {

return THAN;

}

for (int i = 0; i < disassociatives.Count; i++) {

if (word.ToLower() == disassociatives[i]) {

return NEGATIVE;

}

}

if (superlatives != null) {

for (int i = 0; i < superlatives.Count; i++) {

if (word.ToLower() == superlatives[i]) {

return i % 2 == 0 ? SUPER + "(-)" : SUPER + "(+)";

}

}

}

if (isFormerReferencer(word.ToLower())) {

return FORMER;

}

else if (isLatterReferencer(word.ToLower())) {

return LATTER;

}

for (int i = 0; i < numbers.Count; i++) {

if (word.ToLower() == numbers[i]) {

return "Tx(" + makeNumber(word) + ")";

}

}

for (int i = 0; i < quantifiers.Count; i++) {

if (quantifiers[i].Contains(word.ToLower())) {

return "Tq(" + Convert.ToChar('A' + i) + ")";

}

}

if (currency.HasValue) {

if (word[0] == currency.Value) {

return "Tx(" + word + ")";

}

}

for (int i = 0; i < prepositions.Count; i++) {

if (word.ToLower() == prepositions[i]) {

return "Tp(" + getDirection(i) + ")";

}

}

return string.Empty;

}

catch (Exception e) {

throw e;

}

}

private string makeNumber(string word) {

word = word.ToLower();

int result = 0;

string digits = "";

if (Int32.TryParse(word, out result)) {

return result.ToString();

}

foreach (char c in word) {

if (Int32.TryParse(c.ToString(), out result)) {

digits += c;

}

else {

break;

}

}

if (!string.IsNullOrEmpty(digits)) {

return digits;

}

List<string> units = new List<string>() { "zero", "one", "two", "three", "four",

"five", "six", "seven", "eight", "nine", "ten", "eleven",

"twelve", "thirteen", "fourteen", "fifteen", "sixteen",

"seventeen", "eighteen", "nineteen", "twenty" };

for (int i = 0; i <= 20; i++) {

if (word == units[i]) {

return i.ToString();

}

}

List<string> bigNums = new List<string>() { "hundred", "100", "thousand",

"1000", "million", "1000000", "billion", "1000000000" };

for (int i = 0; i < bigNums.Count; i++) {

//only numbers as words can get here

if (word == bigNums[i]) {

return bigNums[i+1];

}

}

throw new ParserException("Can't numberify this yet: " + word);

}

private string getDirection(int index) {

if (index % 2 == 0) {

return "-";

}

return "+";

}

internal static bool isOf(string tag) {

return tag == OF;

}

internal static bool isWith(string tag) {

return tag == WITH;

}

internal static bool isAnd(string tag) {

return tag == AND;

}

internal static bool isBut(string tag) {

return tag == BUT;

}

internal static bool isSingleTermItem(string tag) {

return tag == THIS || tag == BUT || tag.Contains(SUPER) || tag == EITHER;

}

internal static bool isNegative(string p) {

return p == NEGATIVE;

}

private bool isFormerReferencer(string word) {

return word == "former" || word == "first";

}

private bool isLatterReferencer(string word) {

return word == "latter" || word == "second";

}

}

}

1. Parser – CategoryDictionary

using System;

using System.Collections.Generic;

namespace CZParser

{

public class CategoryDictionary

{

internal List<string> codes;

internal List<string> words;

private List<string> categoryTitles;

private static List<string> neutrals = new List<string> {"of", "the", "in",

"out", "and", "a", "on", "went", "to" };

public CategoryDictionary(List<string> categories, List<string> items) {

categoryTitles = new List<string>(categories);

codes = new List<string>();

words = new List<string>();

int catCount = categories.Count;

int catHeight = items.Count / categories.Count;

char catIdentifier = 'A';

for (int i = 0; i < catCount; i++, catIdentifier++) {

for (int j = 0; j < catHeight; j++) {

if (items[(i \* catHeight) + j].Trim().Contains(" ")) {

//multiple words

var splitItem = items[(i \* catHeight) + j].Split(' ');

foreach (string itemWord in splitItem) {

if (shouldBeConsidered(itemWord, categories)) {

codes.Add(catIdentifier.ToString() + (j + 1));

words.Add(itemWord.ToLower().Trim());

}

}

}

else {

codes.Add(catIdentifier.ToString() + (j + 1));

words.Add(items[(i \* catHeight) + j].ToLower().Trim());

}

}

}

}

private bool shouldBeConsidered(string s, List<string> categories) {

if (neutrals.Contains(s.ToLower())) {

return false;

}

return true;

}

public List<string> getItems() {

return words;

}

internal string findItemMatches(string word) {

string result = "";

if (word[0] == '\'') {

word = word.Substring(1);

}

for (int i = 0; i < codes.Count; i++) {

if (words[i] == word.ToLower() || words[i] + "'s" == word.ToLower()

|| words[i] == word.ToLower() + "s"

|| words[i] + "'" == word.ToLower()) {

if (result.Length > 0) {

result += ",";

}

result += codes[i];

}

}

for (int i = 0; i < categoryTitles.Count; i++) {

if (categoryTitles[i].ToLower() == word.ToLower()) {

if (result.Length > 0) {

result += ",";

}

result += (Convert.ToChar('A' + i)).ToString();

}

}

return result;

}

}

}

1. Logix - Category

using System;

using System.Linq;

using Logix.Calculators;

using Representation;

using System.Collections.Generic;

namespace Logix

{

public delegate void MatchEventHandler (Category sender, MatchEventArgs e);

/\* Category class

\* Creates a matrix that will store information on one category. The matrix is always four rows deep with one labelling column

\* and then one column per value within the category. The top row holds the index, the second the value (if necessary) and the

\* third/fourth any known positive/negative relational data so that logical deductions can be drawn.

\* Rules keeps a copy of keywords relevant to the category.

\* The Identifier is the [A|B|C|D|E...] puzzle-scope character for this category.

\*/

public class Category

{

public event MatchEventHandler Matched;

public enum Rows

{

Index = 0,

Values = 1,

Positives = 2,

Negatives = 3

};

public char identifier {get; internal set;}

public int size {get; internal set;}

private string keyword;

private object[][] innerArray;

private Calculator calculator;

internal Category() { size = 0; identifier = 'Z'; }

private void OnMatch(MatchEventArgs e) {

if (Matched != null) {

Matched(this, e);

}

}

internal void createInnerArray()

{

innerArray = createArray(identifier, size);

}

private object[][] createArray(char ident, int size) {

object[][] newArray = { new object[size + 1], new object[size + 1], new

string[size + 1], new string[size+1] };

newArray[1][0] = ident;

for (int i = 0; i <= size; i++) {

newArray[(int)Rows.Index][i] = i;

newArray[(int)Rows.Positives][i] = "";

newArray[(int)Rows.Negatives][i] = "";

}

return newArray;

}

internal void enterValues(object[] list) {

if (list.Length > this.size) {

throw new ArgumentException("List contains too many arguements for

this Line. Line: " + this.identifier + " Size: " + this.size + "

List length: " + list.Length);

}

if (list.Length < this.size) {

throw new ArgumentException("List contains too few arguements for this

Line. Line: " + this.identifier + " Size: " + this.size + " List

length: " + list.Length);

}

if (keyword.ToLower() == "numeric") {

//keep just number

string aux = "";

int n = 0;

for (int i = 0; i < size; i++) {

foreach (string s in list[i].ToString().Split(new char[] {' '})) {

if (int.TryParse(s, out n)) {

aux += n.ToString();

}

}

this.innerArray[1][i+1] = aux;

aux = "";

}

}

else {

for (int i = 1; i <= size; i++) {

this.innerArray[1][i] = list[i - 1];

}

}

}

internal void setKeyword(string key) {

this.keyword = key;

calculator = CalculatorFactory.getInstance().createCalculator(key);

}

public object retrieveValue(string item) {

if (item[0] != this.identifier) {

throw new ArgumentException("Identifier does not match Index provided:

" + item);

}

int column = Convert.ToInt32(item.Substring(1));

if (this.size < column) {

throw new IndexOutOfRangeException("Index (" + item + ") not within

Line " + this.identifier);

}

return this.innerArray[(int)Category.Rows.Values][column];

}

public void addRelation(string p1, string p2, Rows row = Rows.Positives) {

if (p1[0] != identifier) {

throw new ArgumentException("Identifier does not match target

location: " + p1);

}

int column = Convert.ToInt32(p1[1].ToString());

if (this.size < column) {

throw new ArgumentException("Target is out of bounds: " + p1);

}

if (!this.innerArray[(int)row][column].ToString().Contains(p2)) {

this.innerArray[(int)row][column] += p2;

if (row == Rows.Positives) {

OnMatch(new MatchEventArgs(p1, p2));

createNegativeLinkForOthers(p2, column);

}

}

}

public string checkForMatch(string p, Rows row = Rows.Positives) {

if (string.IsNullOrEmpty(p)) return null;

//return category item that relates to item passed.

int finds = 0;

char category = p[0];

string result = null;

if (row == Rows.Positives) {

for (int i = 1; i <= size; i++) {

if (this.innerArray[(int)row][i].ToString().Contains(p)) {

return this.identifier + (i).ToString();

}

if (!this.innerArray[(int)row][i].ToString().Contains(category)) {

finds++;

result = this.identifier + (i).ToString();

}

}

return finds < 2 ? result : null;

}

for (int i = 1; i <= size; i++) {

if (this.innerArray[(int)Rows.Positives][i].ToString().Contains(p))

return null;

if (this.innerArray[(int)row][i].ToString().Contains(p)) {

finds++;

}

else {

result = this.identifier + (i).ToString();

}

}

return finds == size - 1 ? result : null;

}

internal string findTarget(object knownValue, string comparative) {

object targetValue = calculator.calculateValue(knownValue, comparative);

return findItem(targetValue);

}

internal string findItem(object targetValue) {

for (int i = 1; i <= this.size; i++) {

if (innerArray[(int)Rows.Values][i].ToString() ==

targetValue.ToString()) {

return this.identifier + i.ToString();

}

if (innerArray[(int)Rows.Values][i].ToString().Replace(",", "") ==

targetValue.ToString()) {

return this.identifier + i.ToString();

}

}

return null;

}

internal static string[] getMatchedItems(Category c, int index) {

string positivesList = c.innerArray[(int)Rows.Positives][index].ToString();

return splitItems(positivesList);

}

internal static string[] getUnmatchedItems(Category c, int index) {

string negativesList = c.innerArray[(int)Rows.Negatives][index].ToString();

return splitItems(negativesList);

}

private static string[] splitItems(string list) {

if (string.IsNullOrEmpty(list)) return null;

List<string> items = new List<string>();

string item = list[0].ToString();

for (int i = 1; i < list.Length; i++) {

if (Char.IsLetter(list[i])) {

items.Add(item);

item = list[i].ToString();

}

else item += list[i];

}

items.Add(item);

return items.ToArray();

}

internal List<Relation> checkDeductibles() {

List<Relation> relations = new List<Relation>();

string[] negatives = getAllListedItems(Rows.Negatives);

if (negatives != null) {

foreach (string item in negatives.Distinct<string>()) {

//check if it is listed against all but one

string item2 = checkForMatch(item, Rows.Negatives);

if (!string.IsNullOrEmpty(item2)) {

relations.Add(RelationFactory.getInstance()

.createRelation(item, item2, true));

}

}

}

return relations;

}

private string[] getAllListedItems(Rows row) {

string list = "";

for (int i = 1; i <= size; i++) {

list += innerArray[(int)row][i].ToString();

}

return splitItems(list);

}

public List<Relation> considerRelationToCategory(Relation r, bool alreadySeen) {

try {

if (!r.getRule().Contains(identifier)) {

//cannot use this relation

return new List<Relation> { r };

}

if (r.isConditional()) {

if (((ConditionalRelation)r).getRule().Contains(identifier)) {

bool? conditionMet = checkDeterminability(r);

if (conditionMet.HasValue) {

if (conditionMet.Value) {

return new List<Relation> { RelationFactory.getInstance()

.createRelation(((ConditionalRelation)r).getIfTrueStatement())};

}

return new List<Relation> { RelationFactory.getInstance()

.createRelation(((ConditionalRelation)r).getIfFalseStatement())};

}

}

return new List<Relation> { r };

}

if (r.isDirect()) {

if (!alreadySeen) {

Category.Rows row = r.isPositive() ? Category.Rows.Positives :

Category.Rows.Negatives;

addRelation(r.getBaseItem(identifier),

r.getRelatedItem(identifier), row);

return new List<Relation> { r };

}

return null;

}

if (r.isRelative()) {

string[] items = { r.getBaseItem(identifier,

Relations.Sides.Left), r.getBaseItem(identifier,

Relations.Sides.Right) };

if (items[0] == null && items[1] == null) {

return new List<Relation> { r };

}

if (items[0] == items[1]) {

/\*//Single-item Relative

if (items[0] != identifier.ToString()) {

//cannot discover anything within this category

return new List<Relation> { r };

}\*/

string relatedItem = r.getRelatedItem(identifier);

return createImpossiblesForItem(relatedItem,

Relations.getComparator(r.getRule()),

Relations.getComparativeAmount(r.getRule(), false));

}

string leftMatch = checkForMatch(items[0]);

string rightMatch = checkForMatch(items[1]);

List<Relation> results = new List<Relation>();

if (!string.IsNullOrEmpty(leftMatch) &&

!string.IsNullOrEmpty(rightMatch)) {

//both sides already matched

return null;

}

else if (!string.IsNullOrEmpty(leftMatch) ||

!string.IsNullOrEmpty(rightMatch)) {

//if either of the two items has a value, then something can

be learnt for the other, if not a complete match

if (Representation.Relations.isQuantified(r.getRule())) {

string unknownItem = leftMatch == null ? items[0] :

items[1];

object knownValue = retrieveValue(leftMatch ??

rightMatch);

bool inverse = leftMatch == null ? true : false;

string targetItem = findTarget(knownValue,

Relations.getComparativeAmount(r.getRule(), inverse));

addRelation(targetItem, unknownItem, r.isPositive() ?

Category.Rows.Positives : Category.Rows.Negatives);

results.Add(RelationFactory.getInstance()

.createRelation(targetItem, unknownItem, r.isPositive()));

}

else {

results = considerComparative(leftMatch ?? items[0],

Relations.getComparator(r.getRule())

, rightMatch ?? items[1]);

}

return results;

}

else if (items[0] != null && items[1] != null) {

//create negative relations from comparatives

results = (createNegativeRelationsToBounds(items[0], items[1],

r.getComparator(), Relations.getComparativeAmount(r.getRule()

, false), alreadySeen));

if (results == null) results = new List<Relation> { r };

else results.Add(r);

}

return results;

}

if (r.isSemantic()) {

throw new NotImplementedException();

}

throw new ArgumentException("Relation type not recognised: " +

r.GetType().ToString());

}

catch (Exception e) {

throw e;

}

}

/// <summary>

/// For single-item Relative, returns items more/less than bound amount.

/// </summary>

/// <param name="relatedItem"></param>

/// <param name="comparator"></param>

/// <param name="bound"></param>

/// <returns></returns>

private List<Relation> createImpossiblesForItem(string relatedItem

, string comparator, string bound) {

try {

List<Relation> results = new List<Relation>();

for (int i = 0; i < size; i++) {

if (calculator.checkPredicate(innerArray[(int)Rows.Values][i]

, comparator, bound)) {

continue;

}

else {

results.Add(RelationFactory.getInstance()

.createRelation(relatedItem, identifier.ToString() + i, false));

}

}

return results;

}

catch (LogicException l) {

throw l;

}

}

private bool? checkDeterminability(Relation r) {

Relation testRelation = ((ConditionalRelation)r).conditional;

if (testRelation.isRelative()) {

//not yet implemented

return null;

}

else {

string baseItem = testRelation.getBaseItem(this.identifier);

string relatedItem = testRelation.getRelatedItem(this.identifier);

if (innerArray[(int)Rows.Positives][Convert.ToInt32

(baseItem[1].ToString())].ToString().Contains(relatedItem)) {

if (testRelation.isPositive()) {

return true; //Positive rule, positive match

}

return false; //Negative rule, positive match

}

else if (innerArray[(int)Rows.Negatives][Convert.ToInt32

(baseItem[1].ToString())].ToString().Contains(relatedItem)) {

if (testRelation.isPositive()) {

return false; //Positive rule, negative match

}

return true; //Negative rule, negative match

}

return null;

}

}

internal List<Relation> considerComparative(string p1, string comparator

, string p2) {

string matchedIndex = "";

string itemToRelate = "";

List<Relation> results = new List<Relation>();

matchedIndex = p1[0] == identifier ? p1 : p2;

itemToRelate = p1[0] == identifier ? p2 : p1;

comparator = p1[0] == identifier ? comparator : Relations.getInverse(

comparator);

List<int> indexesToCheck = this.calculator.getImpossibles

(Convert.ToInt32(matchedIndex.Substring(1)), comparator, this.size);

foreach (int i in indexesToCheck) {

if (innerArray[(int)Rows.Positives][i].ToString().Contains(itemToRelate[0])) {

continue;

}

if (innerArray[(int)Rows.Negatives][i].ToString().Contains(itemToRelate[0])) {

continue;

}

this.addRelation(identifier + i.ToString(), itemToRelate, Rows.Negatives);

results.Add(RelationFactory.getInstance().createRelation(identifier +

i.ToString(), itemToRelate, false));

}

return results;

}

private void createNegativeLinkForOthers(string p2, int y) {

for (int i = 1; i <= size; i++) {

if (i == y) {

continue;

}

addRelation(identifier.ToString() + i, p2, Rows.Negatives);

}

}

private List<Relation> createNegativeRelationsToBounds(string leftItem, string

rightItem, string comparator, string difference, bool alreadySeen) {

RelationFactory relationBuilder = RelationFactory.getInstance();

List<Relation> relations = new List<Relation>();

if (difference == null) { //not a Quantified Relation

string inverseComparator = Relations.getInverse(comparator);

for (int i = 1; i <= size; i++) {

if (hasNoPossibleOpposite(leftItem, i, comparator, rightItem, alreadySeen)) {

relations.Add(relationBuilder.createRelation(leftItem,

identifier.ToString() + i, false));

}

if (hasNoPossibleOpposite(rightItem, i, inverseComparator, leftItem,

alreadySeen)) {

relations.Add(relationBuilder.createRelation(rightItem,

identifier.ToString() + i, false));

}

}

}

else {

string calculatedItem = "";

string inverseDifference = Relations.getInverse(difference[0].ToString()) +

difference.Substring(1);

for (int i = 1; i <= size; i++) {

calculatedItem = findTarget(innerArray[(int)Rows.Values][i], difference);

if (calculatedItem == null) {

relations.Add(relationBuilder.createRelation(leftItem, identifier +

i.ToString(), false));

}

calculatedItem = findTarget(innerArray[(int)Rows.Values][i],

inverseDifference);

if (calculatedItem == null) {

relations.Add(relationBuilder.createRelation(rightItem, identifier +

i.ToString(), false));

}

if (isNegativelyRelated(leftItem, i)) {

Relation aux = formNegativeToNearbyItem(rightItem, i, difference);

if (aux != null) relations.Add(aux);

}

if (isNegativelyRelated(rightItem, i)) {

Relation aux = formNegativeToNearbyItem(leftItem, i, inverseDifference);

if (aux != null) relations.Add(aux);

}

}

}

relations.AddRange(checkDeductibles());

return relations;

}

private Relation formNegativeToNearbyItem(string leftItem, int i, string

difference) {

RelationFactory relationBuilder = RelationFactory.getInstance();

string calculatedItem = findTarget(innerArray[(int)Rows.Values][i], difference);

if (calculatedItem != null) {

return relationBuilder.createRelation(leftItem, calculatedItem, false);

}

return null;

}

private bool isNegativelyRelated(string leftItem, int i) {

return innerArray[(int)Rows.Negatives][i].ToString().Contains(leftItem);

}

private bool hasNoPossibleOpposite(string item1, int index, string comparator,

string item2, bool alreadySeen) {

if (index == 1 && Relations.checkDirection(comparator) ==

Relations.Directions.Higher) {

return !alreadySeen;

}

if (index == size && Relations.checkDirection(comparator) ==

Relations.Directions.Lower) {

return !alreadySeen;

}

if (innerArray[(int)Rows.Negatives][index].ToString().Contains(item1)) {

return false;

}

List<int> indexesToCheck = this.calculator.getImpossibles(index, comparator,

this.size);

foreach (int i in indexesToCheck) {

if (!innerArray[(int)Rows.Negatives][i].ToString().Contains(item2)) {

return false;

}

}

return true;

}

internal bool isInnerArraySet() {

return innerArray != null;

}

}

}