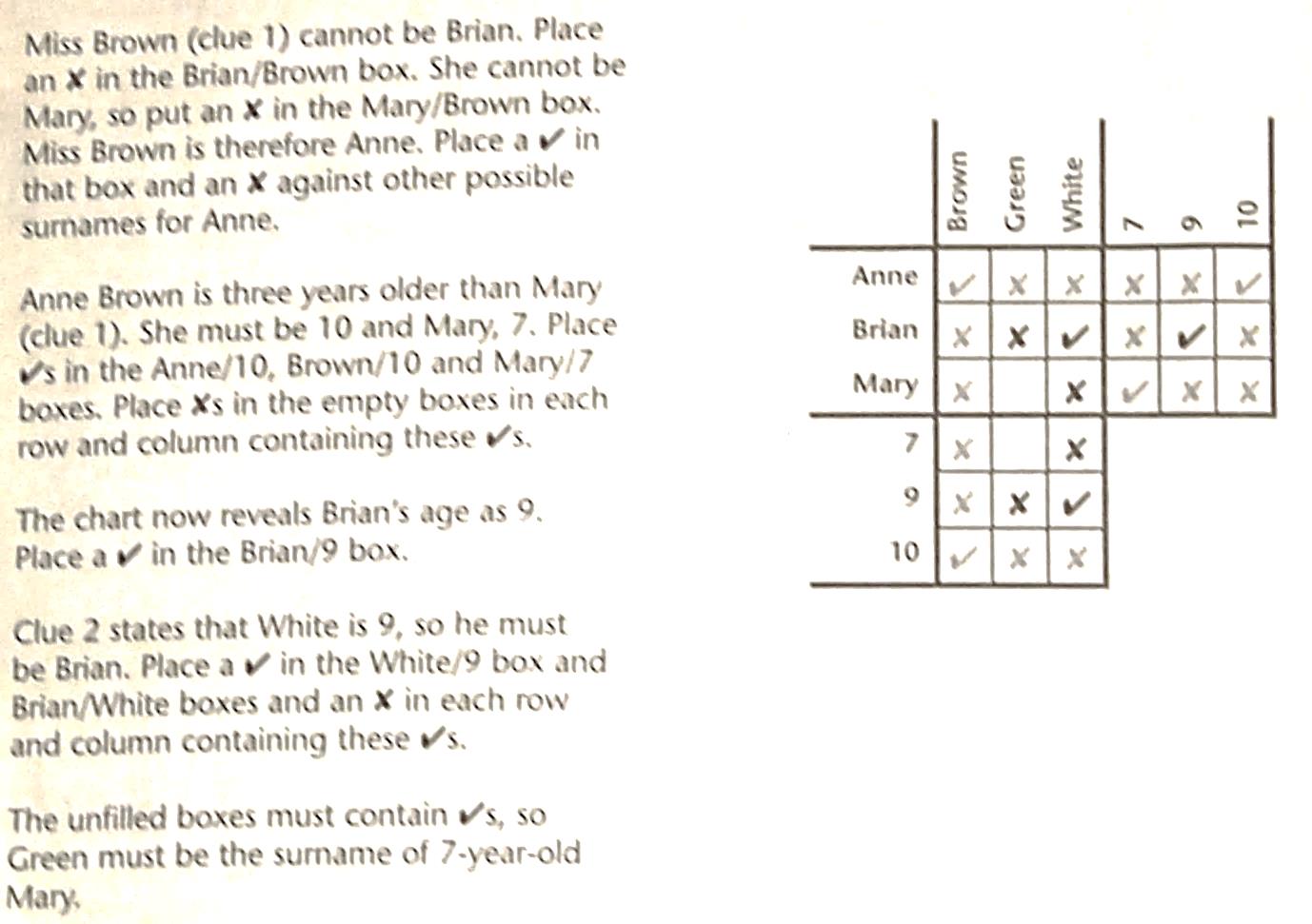
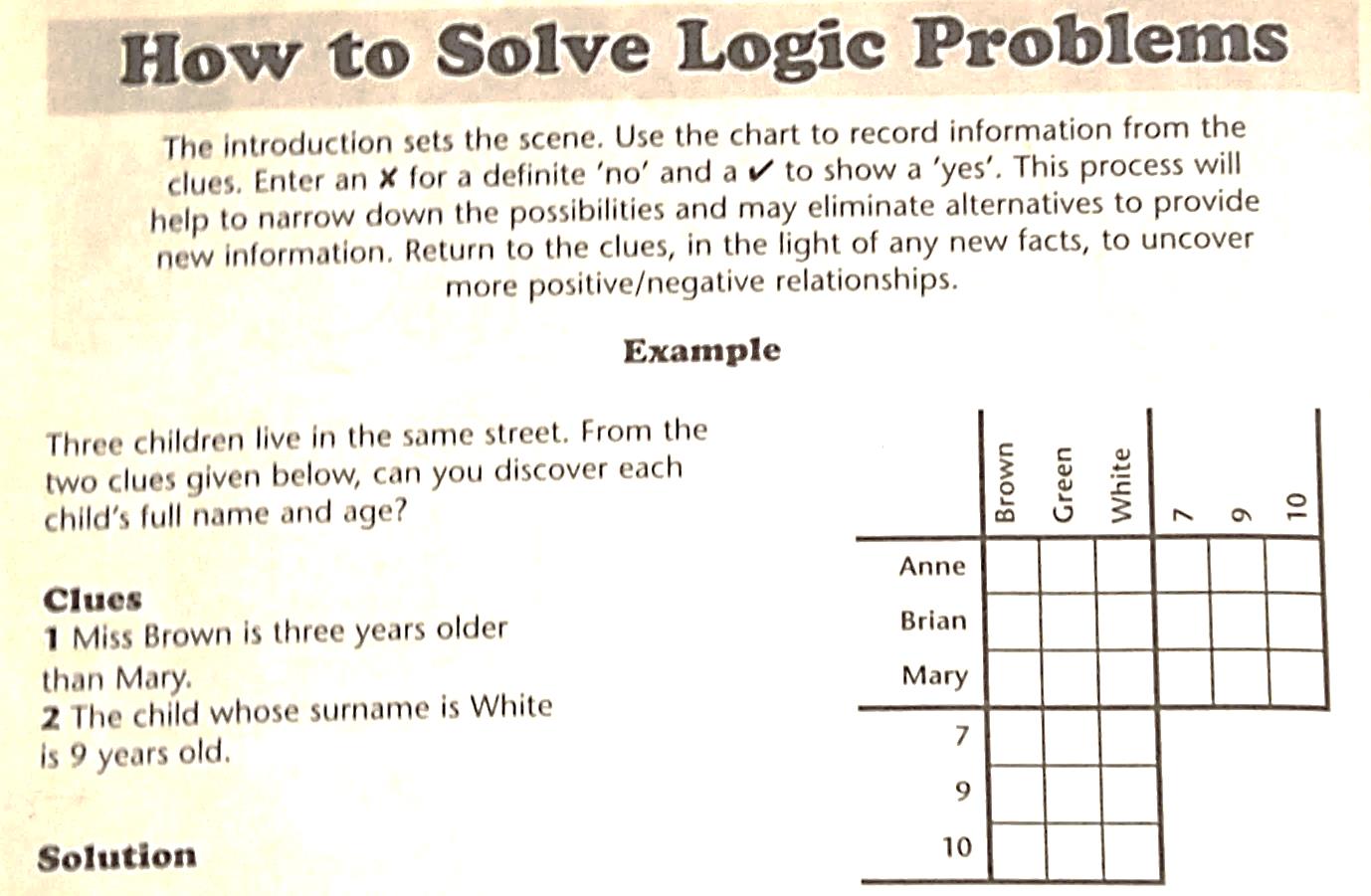
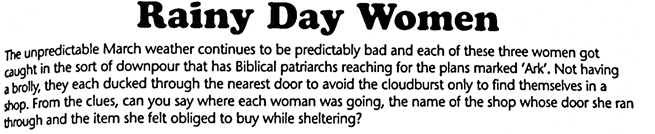
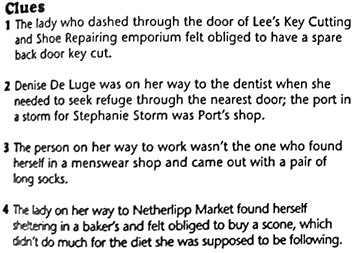
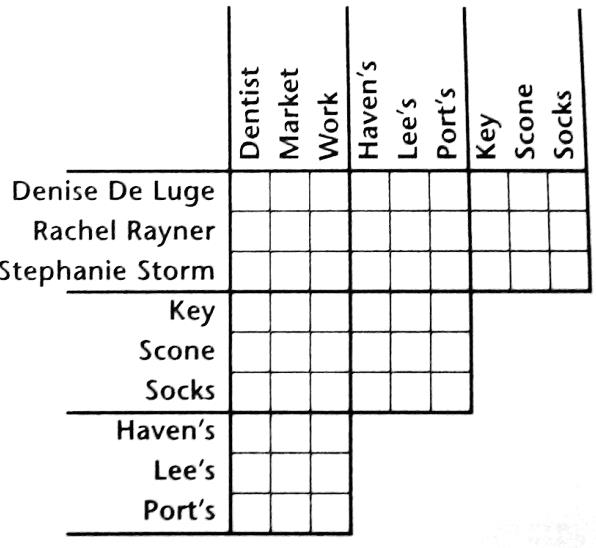
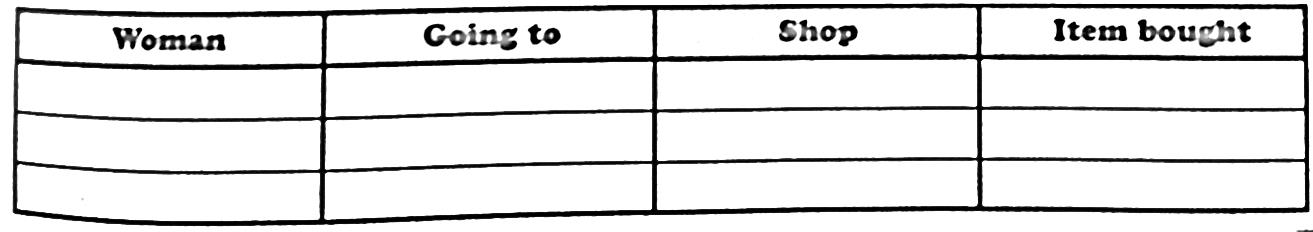
# Appendices

Appendix A1 – Original *Logic Problems* teaching problem taken from May edition [6]



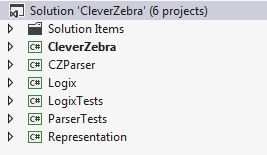
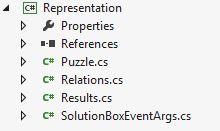
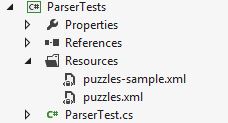
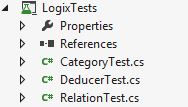
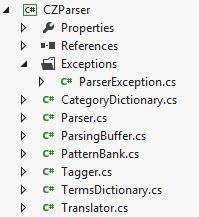
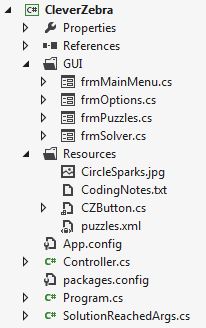
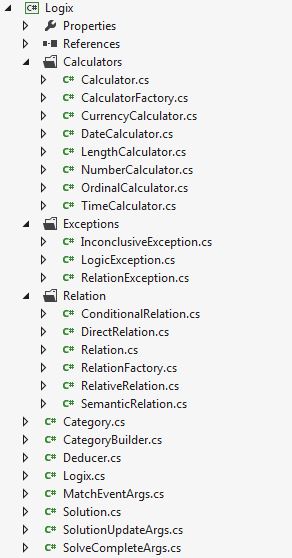
Appendix A2 – Full Sample Problem

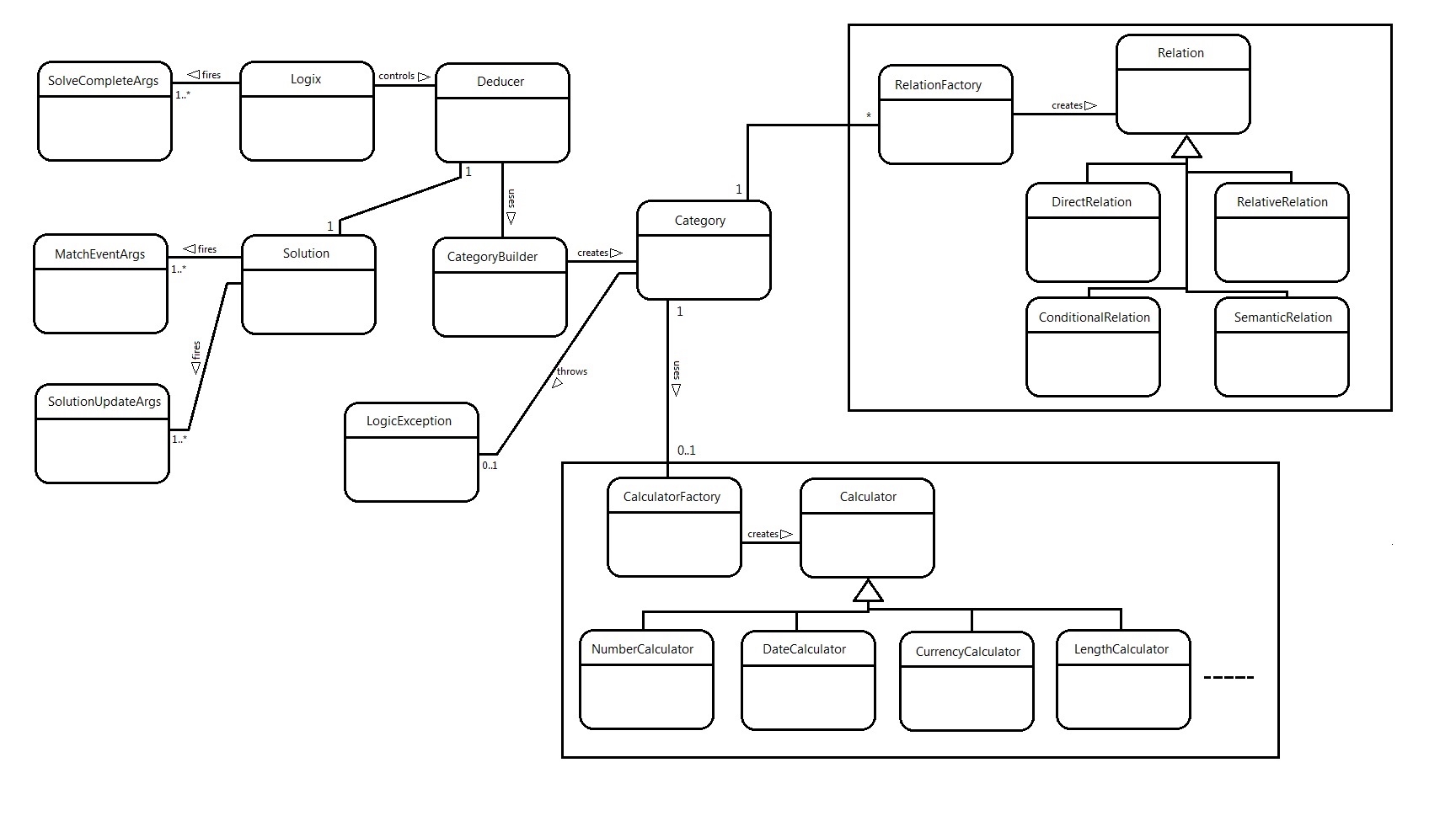


Puzzle “Rainy Day Women” taken from Logic Problems magazine, May 2013 edition [6].

Appendix B – Code Structure

1. Solution Structure



1. Logix Class Diagram
2. 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;

}

}

}

5a. Logix – Logix.cs

using System;

using System.Collections.Generic;

using Representation;

namespace Logix {

public delegate void LogixUpdateHandler(Logix sender, SolutionBoxEventArgs e);

public class Logix {

public SolveCompleteHandler solveHandler;

public LogixUpdateHandler updater;

private Representation.Results latestResults;

private Puzzle p;

internal bool goSlow;

int[,] solutionMatrix;

public Logix(bool slow) { goSlow = slow; }

public List<List<string>> Solve(Puzzle p) {

this.p = p;

Deducer brains = new Deducer(p, goSlow );

brains.Concluded += brains\_solveComplete;

brains.Update += brains\_Update;

brains.setClues(p.getRules());

this.solutionMatrix = new int[p.height, p.width];

try {

solutionMatrix = brains.Go();

}

catch (LogicException l) {

if (l.problemItem != null) {

string friendlyName = p.getNameAt(l.problemItem[0] - 'A',

Convert.ToInt32(l.problemItem.Substring(1)));

throw new LogicException(l.Message + friendlyName + ".\nReasoning failed."

, l.problemItem);

}

else throw l;

}

catch (Exception e) {

throw e;

}

//latestResults.SetSuccess(isCorrect(translateSolution(p, solutionMatrix)));

//Translate matrix solution to verbal solution

return translateSolution(p, solutionMatrix);

}

void brains\_Update(Deducer sender, SolutionUpdateArgs e) {

if (updater != null) {

int category = e.item[0] - 'A';

int catItem = Convert.ToInt32(e.item.Substring(1));

updater(this, new SolutionBoxEventArgs(p.getNameAt(category, catItem), e.line

, category));

}

}

private static List<List<string>> translateSolution(Puzzle p, int[,] solutionMatrix) {

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

for (int x = 0; x < p.height; x++) {

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

for (int y = 0; y < p.width; y++) {

string item = p.getNameAt(y, solutionMatrix[x, y]);

row.Add(item);

}

solutionStrings.Add(row);

}

return solutionStrings;

}

void brains\_solveComplete(Deducer sender, SolveCompleteArgs e) {

this.latestResults = new Results(e.isSuccessful, e.turns, e.timeTaken);

}

private bool isCorrect(List<List<string>> solutionLines) {

foreach (List<string> row in solutionLines) {

if (rowIsInProvidedSolution(row)) {

continue;

}

else {

return false;

}

}

return true;

}

private bool rowIsInProvidedSolution(List<string> row) {

bool lineFound = false;

foreach (List<string> line in p.ProvidedSolution) {

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

if (line[i].ToLower().Contains(row[i].ToLower())) {

lineFound = true;

}

else {

lineFound = false;

break;

}

}

if (lineFound) return lineFound;

}

return false;

}

public Results getLastResults() {

return latestResults;

}

}

}

5b. Logix – Category.cs

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;

}

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]) {

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;

}

}

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 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;

}

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 string[] getAllListedItems(Rows row) {

string list = "";

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

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

}

return splitItems(list);

}

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();

}

/// <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;

}

}

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);

}

}

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 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;

}

}

}

5c. Logix – Deducer.cs

using Representation;

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

namespace Logix {

public delegate void SolveCompleteHandler(Deducer sender, SolveCompleteArgs e);

public delegate void DeducerUpdateHandler(Deducer sender, SolutionUpdateArgs e);

public class Deducer

{

public event SolveCompleteHandler Concluded;

public event DeducerUpdateHandler Update;

private RelationFactory relationBuilder;

private List<Category> cats;

private string[] keywords;

private List<string> clues;

private List<string> usedClues;

private int puzzleBreadth;

private int puzzleDepth;

private Solution solution;

private const int MAXTURNS = 200;

private int Absurdio\_Spacing;

/// <summary>

/// Constructor for testing purposes

/// </summary>

/// <param name="x"></param>

/// <param name="y"></param>

/// <param name="keys"></param>

public Deducer(int x, int y, string[] keys = null) {

this.puzzleBreadth = x;

this.puzzleDepth = y;

this.relationBuilder = RelationFactory.getInstance();

this.cats = new List<Category>();

this.keywords = keys ?? new string[4];

CategoryBuilder catBuilder = new CategoryBuilder();

char c = 'A';

for (int i = 0; i < puzzleBreadth; i++, c++) {

catBuilder.newCat();

catBuilder.setIdentifier(c);

catBuilder.setSize(puzzleDepth);

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

catBuilder.setKeyword(keywords[i]);

}

this.cats.Add(catBuilder.build());

cats[i].Matched += Deducer\_Matched;

}

solution = new Solution(x,y);

solution.goSlow = false;

}

public Deducer(Puzzle p, bool slow = false) {

this.puzzleBreadth = p.width;

this.puzzleDepth = p.height;

this.relationBuilder = RelationFactory.getInstance();

this.cats = new List<Category>();

this.keywords = p.getKeywords();

CategoryBuilder catBuilder = new CategoryBuilder();

char c = 'A';

for (int i = 0; i < puzzleBreadth; i++, c++) {

catBuilder.newCat();

catBuilder.setIdentifier(c);

catBuilder.setSize(puzzleDepth);

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

catBuilder.setKeyword(keywords[i]);

catBuilder.setValues(p.getItems().GetRange(i \* puzzleDepth

, puzzleDepth).ToArray());

}

this.cats.Add(catBuilder.build());

cats[i].Matched += Deducer\_Matched;

}

solution = new Solution(p.width, p.height);

solution.goSlow = slow;

solution.Updater += solution\_Updater;

}

void solution\_Updater(Solution sender, SolutionUpdateArgs e) {

if (Update != null) {

Update(this, e);

}

}

internal void enterCategoryValues(char ident, object[] vals) {

getCategoryFromIdentifier(ident).enterValues(vals);

}

public Category getCategoryFromIdentifier(char ident) {

foreach (Category l in this.cats) {

if (l.identifier == ident) { return l; }

}

throw new ArgumentException("No Category found for identifier: " + ident);

}

void Deducer\_Matched(Category sender, MatchEventArgs e) {

List<Relation> newRules = solution.considerRelationInSolution(

relationBuilder.createRelation(e.item1, e.item2, true));

combineRelationRanges(ref newRules, solution.checkAllButOnes());

if (newRules.Count > 0) {

foreach (Relation r in newRules) {

addRelationToClues(r);

}

}

}

private void OnSolutionComplete(SolveCompleteArgs solveCompleteArgs) {

if (Concluded != null) {

Concluded(this, solveCompleteArgs);

}

}

public List<Category> getCategoryCollection() {

return cats;

}

internal void setClues(List<string> clues) {

this.clues = clues;

}

internal object getRemainingClueCount() {

return this.clues.Count;

}

/// <summary>

/// The main high-level algorithm for solving puzzles

/// </summary>

/// <returns></returns>

internal int[,] Go() {

Stopwatch ticker = new Stopwatch();

int turn = 1;

usedClues = new List<string>();

Absurdio\_Spacing = clues.Count;

ticker.Start();

while (clues.Count > 0 && turn < MAXTURNS && !solution.isComplete()) {

string clue = clues[0];

List<Relation> relations = considerRelationToCategories(

relationBuilder.createRelation(clue));

usedClues.Add(clues[0]);

clues.RemoveAt(0);

if (turn % Absurdio\_Spacing == 0 || clues.Count == 1) {

relations.AddRange(Absurdio());

}

if (relations != null && relations.Count() > 0) {

foreach (Relation r in relations) {

addRelationToClues(r);

}

}

turn++;

}

ticker.Stop();

TimeSpan t = ticker.Elapsed;

OnSolutionComplete(new SolveCompleteArgs(solution.isComplete(), turn, t));

return solution.getFinalMatrix();

}

/// <summary>

/// Checks a Relation against each Category

/// </summary>

/// <param name="relation"></param>

/// <returns></returns>

private List<Relation> considerRelationToCategories(Relation relation) {

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

foreach (Category cat in cats) {

combineRelationRanges(ref result, cat.considerRelationToCategory(relation

, usedClues.Contains(relation.getRule())));

}

return result;

}

/// <summary>

/// Updates one set of Relations with all distinct Relations from another

/// </summary>

/// <param name="relations1"></param>

/// <param name="relations2"></param>

private void combineRelationRanges(ref List<Relation> relations1

, List<Relation> relations2) {

if (relations1 == null && relations2 == null) {

return;

}

if (relations1 == null) {

relations1 = relations2.Distinct<Relation>().ToList();

}

if (relations2 == null) {

return;

}

//combine unique

foreach (Relation r in relations2) {

if (relations1.Any(a => a.CompareTo(r)==0)) { continue; }

relations1.Add(r);

}

relations1 = relations1.Distinct<Relation>().ToList();

}

/// <summary>

/// Performs a check for any other relations that can be deduced by considering

/// related items' pairings.

/// (Reductio ad absurdum)

/// </summary>

/// <returns>A list of relations discovered.</returns>

private List<Relation> Absurdio() {

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

bool fullyChecked = false;

while (!fullyChecked) {

foreach (Category cat in cats) {

List<Relation> deductedRelations = cat.checkDeductibles();

if (deductedRelations.Count() > 0) {

results.AddRange(deductedRelations);

}

for (int itemIndex = 1; itemIndex <= cat.size; itemIndex++) {

string[] matchedItems = Category.getMatchedItems(cat, itemIndex);

if (matchedItems != null) {

string[] unmatchedItems = Category.getUnmatchedItems(cat, itemIndex);

foreach (string match in matchedItems) {

//see if matched item has negative connections to items first item doesn't

string[] relatedUnmatches = Category.getUnmatchedItems(

getCategoryFromIdentifier(match[0])

, Convert.ToInt32(match[1].ToString()));

if (relatedUnmatches == null) continue;

string[] newNegatives = relatedUnmatches.Except(unmatchedItems

?? new string[] {""}).ToArray();

if (newNegatives.Count() > 0) {

string comparedItem = cat.identifier.ToString() + itemIndex;

combineRelationRanges(ref results, createNegativeRelations

(comparedItem, newNegatives));

}

}

}

List<Relation> allButOnes = checkNegativesForAllButOne(cat, itemIndex);

if (allButOnes != null) {

combineRelationRanges(ref results, allButOnes);

}

}

}

fullyChecked = true;

}

return results;

}

/// <summary>

/// Creates negative relations for the item against a list of related items

/// </summary>

/// <param name="item"></param>

/// <param name="notRelateds"></param>

/// <returns></returns>

private List<Relation> createNegativeRelations(string item, string[] notRelateds) {

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

foreach (string s in notRelateds) {

if (item[0] == s[0]) {

continue;

}

relations.Add(relationBuilder.createRelation(item, s, false));

}

return relations;

}

/// <summary>

/// Returns a relation per category if only one valid option remains.

/// </summary>

/// <param name="cat"></param>

/// <param name="itemIndex"></param>

/// <returns></returns>

private List<Relation> checkNegativesForAllButOne(Category cat, int itemIndex) {

string[] Negatives = Category.getUnmatchedItems(cat, itemIndex);

if (Negatives == null || Negatives.Count() < cat.size-1) {

return null;

}

char ident = 'A';

for (int catIndex = 0; catIndex < cat.size; catIndex++, ident++) {

if (Negatives.Where(a => a.StartsWith(ident.ToString())).Count() == cat.size - 1) {

//All but one negated. Find positive

string positiveMatchItem = "";

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

if (!Negatives.Contains(ident.ToString() + i)) {

positiveMatchItem = ident.ToString() + i;

}

}

addRelationToClues(relationBuilder.createRelation(cat.identifier.ToString()

+ itemIndex, positiveMatchItem, true));

addInverseRelation(cat.identifier.ToString() + itemIndex, positiveMatchItem

, Category.Rows.Positives);

}

}

return null;

}

/// <summary>

/// Adds Relation to clue bank if not seen already, or if it is Relative

/// </summary>

/// <param name="r"></param>

internal void addRelationToClues(Relation r) {

if (clues.Contains(r.getRule()) || clues.Contains(r.getFlippedRule())) return;

if (r.isDirect() && (usedClues.Contains(r.getRule())

|| usedClues.Contains(r.getFlippedRule()))) return;

this.clues.Add(r.getRule());

}

private void addInverseRelation(string p1, string p2, Category.Rows row) {

getCategoryFromIdentifier(p1[0]).addRelation(p1, p2, row);

}

}

}

6a. CZParser – Parser.cs

using System.Collections.Generic;

using Representation;

namespace CZParser {

public class Parser {

private Puzzle puzzle;

internal Tagger tagger;

internal Translator translator;

public Parser(Puzzle p) {

puzzle = p;

tagger = new Tagger(puzzle.getCategories(), puzzle.getItems()

, puzzle.getKeywords());

translator = new Translator(p.height, puzzle.getKeywords());

}

public List<string> Read() {

List<string> tagLines = tagger.tagClues(puzzle.getClues());

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

foreach (string line in tagLines) {

results.AddRange(translator.makeRelations(line));

}

return results;

}

}

}

6b. CZParser – Tagger.cs

using System;

using System.Collections.Generic;

using System.Linq;

namespace CZParser

{

class Tagger {

internal CategoryDictionary catWords;

internal TermsDictionary terms;

internal List<string> keywords;

private ParsingBuffer buffer;

public Tagger(List<string> categories, List<string> items, string[] keys) {

catWords = new CategoryDictionary(categories, items);

if (keys.Contains("currency")) {

char curr = ' ';

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

if (keys[i].ToLower() == "currency") {

curr = items[i\*(items.Count/categories.Count)][0];

}

}

terms = new TermsDictionary(keys, curr);

}

else {

terms = new TermsDictionary(keys);

}

keywords = new List<string>();

keywords.AddRange(keys);

}

internal List<string> tagClues(List<string> clues) {

List<string> results = new List<string>() { };

foreach (string clue in clues) {

results.Add(formTagString(clue));

}

return results;

}

internal string formTagString(string clue) {

List<string> words = separateWordsAndPunctuation(clue);

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

string auxCat1 = "";

string auxCat2 = "";

foreach (string word in words) {

if(word.Length==1 && char.IsPunctuation(word[0])) {

if (word == "." || word == ";" || word == ",") {

firstTagPattern.Add(word);

}

continue;

}

auxCat1 = catWords.findItemMatches(word);

auxCat2 = terms.defineItem(word);

firstTagPattern.Add(string.IsNullOrEmpty(auxCat1) ? auxCat2 : string.IsNullOrEmpty(auxCat2) ? auxCat1 : auxCat1 + "," + auxCat2);

}

return condenseToString(firstTagPattern);

}

internal static List<string> separateWordsAndPunctuation(string clue) {

if (clue.Contains('(')) {

clue = removeParentheticals(clue);

}

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

string[] words = clue.Split(new char[] {' ', '-'}, StringSplitOptions.RemoveEmptyEntries);

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

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

char punctuation = '0';

words[i].Replace("\"", "");

if (words[i].EndsWith(",") || words[i].EndsWith(";") || words[i].EndsWith(".")) {

punctuation = words[i][words[i].Length - 1];

words[i] = words[i].Remove(words[i].Length - 1);

}

result.Add(words[i]);

if (char.IsPunctuation(punctuation))

result.Add(punctuation.ToString());

}

return result;

}

private static string removeParentheticals(string clue) {

while (clue.Contains('(')) {

clue = clue.Remove(clue.IndexOf(" ("), (clue.IndexOf(')') - clue.IndexOf(" (")) + 1);

}

return clue;

}

private string condenseToString(List<string> tagLine) {

string result = "";

this.buffer = new ParsingBuffer(3);

bool whitespaceGap = false;

string heldTag = ""; //held items require recordable items afterwards to themselves be recorded.

string lastNonBlank = "";

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

if (string.IsNullOrEmpty(tagLine[i])) {

whitespaceGap = true;

if (!buffer.isEmpty() && !buffer.stretchesOverWhitespace()) {

string catMention = buffer.pullCategoryTitle();

result += string.IsNullOrEmpty(catMention) ? "" : catMention + " ";

buffer.Clear();

}

continue;

}

if (i > 0 && tagLine[i] == lastNonBlank) {

continue;

}

if (buffer.isEmpty()) {

string newItem = evaluateTag(lastNonBlank, tagLine[i], ref heldTag);

result += addTag(ref heldTag, newItem);

if (newItem == null) {

continue;

}

else {

lastNonBlank = newItem;

}

}

else {

if (buffer.hasCombinedCats()) {

string aux = evaluateCatTags(lastNonBlank, tagLine[i], ref heldTag);

result += addTag(ref heldTag, aux);

if (aux != null) {

lastNonBlank = aux;

}

}

else if (buffer.hasMixedTags()) {

if (isCatTag(tagLine[i])) {

if (buffer.Contains(tagLine[i])) {

result += addTag(ref heldTag, tagLine[i]);

buffer.Clear();

buffer.Add(tagLine[i]);

}

else {

throw new ParserException("Wasn't expecting this combination: " + buffer.ToString() + " followed by " + tagLine[i]);

}

}

else {

if (!whitespaceGap) {

buffer.dropNonTermTags();

string aux = evaluateTermTags(lastNonBlank, tagLine[i], ref heldTag);

result += addTag(ref heldTag, aux);

if (aux != null) {

lastNonBlank = aux;

}

}

else {

throw new ParserException("Wasn't expecting this combination: " + buffer.ToString() + " followed by " + tagLine[i]);

}

}

}

else {

string aux = evaluateTermTags(lastNonBlank, tagLine[i], ref heldTag);

result += addTag(ref heldTag, aux);

if (aux != null) {

lastNonBlank = aux;

}

}

}

whitespaceGap = false;

}

return finaliseResult(result);

}

private string addTag(ref string heldTag, string p) {

if (string.IsNullOrEmpty(p)) {

return null;

}

string pretag = "";

if (!string.IsNullOrEmpty(heldTag)) {

pretag = heldTag;

heldTag = "";

}

return pretag + p + " ";

}

private string evaluateTag(string previous, string tag, ref string heldTag) {

if (endOfHeldTagIsBut(heldTag)) {

if (isDisassociative(tag) || tag == "Tt") {

heldTag = tag + " ";

return null;

}

}

if (tagMustBeHeld(tag)) {

if (tag == ",") {

heldTag = tag + " ";

}

else {

heldTag += tag + " ";

}

return null;

}

if (TermsDictionary.isAnd(tag)) {

buffer.Add(tag);

return null;

}

if (isCatTag(tag)) {

if (tag.Length > 1) {

return tag;

}

if (!string.IsNullOrEmpty(keywords[tag[0] - 'A'])) {

return tag;

}

return null;

}

else if (isCombinedCatTag(tag)) {

//check previous item

if (previous != "," && tag.Contains(previous)) {

//already covered by previous recorded tag

return null;

}

buffer.Add(tag);

}

else if (isTermTag(tag)) {

if (TermsDictionary.isSingleTermItem(tag)) {

if (continuesPhrase(tag, heldTag)) {

heldTag = "";

return null;

}

if (!TermsDictionary.isSingleTermItem(heldTag.Trim())) {

return tag;

}

return null;

}

if (TermsDictionary.isOf(tag)) {

if (pairsWithOf(previous)) {

return previous + " " + tag;

}

else return null;

}

if (TermsDictionary.isWith(tag)) {

if (pairsWithWith(previous)) {

return previous + " " + tag;

}

else return null;

}

buffer.Add(tag);

}

else if (hasNumber(tag) && hasCatTag(tag)) {

return evaluateTag(previous, getCatTagFromCombo(tag), ref heldTag);

}

else if (isCombinedTermCompCatTag(tag)) {

return evaluateTag(previous, getTermTagFromCombo(tag), ref heldTag);

}

else if (hasCatTag(tag) && hasTermTag(tag)) {

string catTag = getCatTagFromCombo(tag);

if (catTag != null && catTag.Length > 1) {

return evaluateTag(previous, getCatTagFromCombo(tag), ref heldTag);

}

return null;

}

return null;

}

private bool continuesPhrase(string tag, string heldTag) {

//wittering phrase, not required

if (endOfHeldTagIsBut(heldTag.Trim()) && tag == "Tt") {

return true;

}

return false;

}

private bool endOfHeldTagIsBut(string heldTag) {

if (!heldTag.Trim().Contains(" ")) {

return TermsDictionary.isBut(heldTag.Trim());

}

string lastPart = heldTag.Substring(heldTag.Trim().LastIndexOf(" "));

return TermsDictionary.isBut(lastPart.Trim());

}

private string evaluateCatTags(string previous, string tag, ref string heldTag) {

//buffer contains cat items (from a multi-option tag)

if(tag.Length == 1 && char.IsPunctuation(tag[0])) {

buffer.Clear();

if (!string.IsNullOrEmpty(heldTag)) {

heldTag += ',';

}

heldTag += tag;

return null;

}

if (buffer.Contains(tag)) {

//could be a two-option tag after a three-option tag.

if (isCatTag(tag)) {

buffer.Clear();

return tag;

}

else {

foreach (string bit in tag.Split(new char[] { ',' }, StringSplitOptions.RemoveEmptyEntries)) {

if (!isCatTag(bit)) {

tag = removeTagFromGroupTag(tag, bit);

}

}

if (isCatTag(tag)) {

buffer.Clear();

return tag;

}

}

buffer.Add(tag);

return null;

}

if (isCatTag(tag)) {

//previous combo was shared item part. Ignore in preference to this

buffer.Clear();

return tag;

}

else {

if (isTermTag(tag)) {

//we have a term/combo cat-term tag

string category = buffer.pullCategoryTitle();

//drop previous buffer, start considering term with cat mention, if necessary

buffer.Clear();

return string.IsNullOrEmpty(category) ? evaluateTag(previous, tag, ref heldTag) : category + " " + evaluateTag(previous, tag, ref heldTag);

}

else {

try {

if (buffer.Contains(getCatTagFromCombo(tag))) {

buffer.Clear();

return getCatTagFromCombo(tag);

}

}

catch (ParserException p) {

throw p;

}

throw new ParserException("Wasn't expecting this buffer: " + buffer.ToString() + " to be followed by " + tag);

}

}

}

private string evaluateTermTags(string previous, string tag, ref string heldTag) {

string newTerm = "";

if (isMixedTag(tag)) {

newTerm = getTermTagFromCombo(tag);

}

else {

newTerm = tag;

}

if (PatternBank.completesTagPattern(buffer.ToString(), newTerm)) {

string aux = buffer.ToString();

if (newTerm != "Th") {

aux += " " + newTerm;

}

buffer.Clear();

return aux.Replace(",", " ");

}

else if (PatternBank.continuesTagPattern(buffer.ToString(), newTerm)) {

buffer.Add(newTerm);

return null;

}

if (isMixedTag(tag)) {

buffer.Add(tag);

return null;

}

else {

buffer.Clear();

return evaluateTag(previous, tag, ref heldTag);

}

}

/// <summary>

/// Removes excess whitespace and category titles then checks subclausing

/// </summary>

/// <param name="result"></param>

/// <returns></returns>

private string finaliseResult(string result) {

result = result.Trim();

result = result.Replace("Tt Td", "Td");

result = result.Replace("Td Tb ", "");

if (result.Substring(result.LastIndexOf(" ") + 1) == "Te") {

result = result.Substring(0, result.LastIndexOf(" "));

}

if (result.StartsWith("Te") && !result.Substring(3).Contains("Te")) {

result = result.Substring(3);

}

//no need for category titles

foreach (string bit in result.Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries)) {

if (isCatTag(bit) && bit.Length == 1) {

result = result.Replace(bit + " ", "");

if (result.EndsWith(bit)) {

result = result.Substring(0, result.Length - 1);

}

}

}

result = checkSubclausing(result);

return result.Replace(" ", " ").Trim();

}

private string checkSubclausing(string tagline) {

if (!tagline.Contains(',')) {

return tagline;

}

string result = "";

string[] sentences = tagline.Split(new char[] { '.', ';' }, StringSplitOptions.RemoveEmptyEntries);

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

if (i > 0) {

result += " . ";

}

if (!sentences[i].Contains(',')) {

result += sentences[i];

continue;

}

string[] parts = sentences[i].Split(new string[] { " , ", ", ", "," }, StringSplitOptions.RemoveEmptyEntries);

if (parts.Count() < 3) {

result += recombineParts(parts);

continue;

}

else {

string reworkedSentence = parts[0];

if (hasManyParts(parts[1].Trim())) {

reworkedSentence += " , " + parts[1] + " ,";

}

else {

reworkedSentence += " " + parts[1];

}

for (int j = 2; j < parts.Count(); j++) {

reworkedSentence += " " + parts[j];

}

result += reworkedSentence;

}

}

return result;

}

private string recombineParts(string[] parts) {

string result = "";

foreach (string s in parts) {

result += s + " ";

}

return result.Trim();

}

private bool hasManyParts(string p) {

return p.Contains(" ");

}

private bool tagMustBeHeld(string tag) {

if (tag.Length == 1 && char.IsPunctuation(tag[0])) {

return true;

}

if (isDisassociative(tag)) {

return true;

}

if (TermsDictionary.isBut(tag)) {

return true;

}

return false;

}

private bool isCombinedTermCompCatTag(string tag) {

bool hasCompCat = false;

bool hasTerm = false;

string[] bits = tag.Split(new char[] { ',', '{', '}' }, StringSplitOptions.RemoveEmptyEntries);

foreach (string t in bits) {

if (t.Length == 1 && isCatTag(t)) {

hasCompCat = true;

}

else if (isTermTag(t)) {

hasTerm = true;

}

else {

return false;

}

}

return hasCompCat && hasTerm;

}

private bool hasCatTag(string tag) {

string[] bits = tag.Split(new char[] { ',', '{', '}' }, StringSplitOptions.RemoveEmptyEntries);

foreach (string t in bits) {

if (isCatTag(t)) {

return true;

}

}

return false;

}

private bool hasTermTag(string tag) {

string[] bits = tag.Split(new char[] { ',', '{', '}' }, StringSplitOptions.RemoveEmptyEntries);

foreach (string t in bits) {

if (isTermTag(t)) {

return true;

}

}

return false;

}

private bool hasNumber(string tag) {

string[] bits = tag.Split(new char[] { ',', '{', '}' }, StringSplitOptions.RemoveEmptyEntries);

foreach (string t in bits) {

if (char.IsDigit(t[0])) {

return true;

}

}

return false;

}

private bool isMixedTag(string tag) {

bool hasTerm = false;

bool hasCat = false;

string[] bits = tag.Split(new char[] { ',', '{', '}' }, StringSplitOptions.RemoveEmptyEntries);

foreach (string t in bits) {

if (!hasCat && isCatTag(t)) {

hasCat = true;

}

else if (!hasTerm && isTermTag(t)) {

hasTerm = true;

}

}

return hasCat && hasTerm;

}

internal static bool isCatTag(string tag) {

if (tag.Length == 1 && !char.IsPunctuation(tag[0])) {

if (char.IsDigit(tag[0])) {

return false;

}

return true;

}

int index;

if (tag.Contains(",") == false && Int32.TryParse(tag.Substring(1), out index)) {

return Char.IsLetter(tag[0]);

}

return false;

}

private bool isDisassociative(string tag) {

return tag == "Td";

}

private bool isTermTag(string tag) {

return tag[0] == 'T';

}

private bool isConsiderable(string tag) {

return false; //No extras currently being considered. May be where semantic information is dropped in?

}

private bool isCombinedCatTag(string tag) {

string[] bits = tag.Split(new char[] { ',', '{', '}' }, StringSplitOptions.RemoveEmptyEntries);

foreach (string t in bits) {

if (!isCatTag(t)) {

return false;

}

}

return true;

}

private string removeTagFromGroupTag(string tag, string badTag) {

string newTag = "";

foreach (string bit in tag.Split(new string[] { ", " }, StringSplitOptions.RemoveEmptyEntries)) {

if (bit.Trim() == badTag.Trim()) {

continue;

}

newTag += string.IsNullOrEmpty(newTag) ? bit : "," + bit;

}

return newTag;

}

private string getCatTagFromCombo(string tag) {

//can only safely return if ONE complete cat tag is found

string possible = "";

foreach (string bit in tag.Split(new char[] { ',', '{', '}' }, StringSplitOptions.RemoveEmptyEntries)) {

if (isTermTag(bit)) {

tag.Replace(bit, "");

}

else if (isCatTag(bit) && bit.Length > 1) {

if (string.IsNullOrEmpty(possible)) {

possible = bit;

}

else {

return null;

}

}

}

return possible;

}

private string getTermTagFromCombo(string tag) {

foreach (string bit in tag.Split(new char[] { ',', '{', '}' }, StringSplitOptions.RemoveEmptyEntries)) {

if (isTermTag(bit)) {

return bit;

}

}

throw new ParserException("Wasn't expecting to not find a term tag in here: " + tag);

}

private bool pairsWithOf(string tag) {

if (tag == "Tf" || tag == "Tl") {

return true;

}

return false;

}

private bool pairsWithWith(string tag) {

if (tag == "Tp") {

return true;

}

return false;

}

}

}

6c. CZParser – Translator.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using Representation;

namespace CZParser

{

class Translator

{

private static ParsingBuffer buffer = new ParsingBuffer(6);

private List<char> keyCategories;

private static string MIN = "1";

private string MAX;

public Translator(int depth, string[] keys) {

keyCategories = new List<char>();

MAX = depth.ToString();

char cat = 'A';

for (int i = 0; i < keys.Length; i++, cat++) {

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

keyCategories.Add(cat);

}

}

}

internal List<string> makeRelations(string line) {

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

if (containsMultipleStatements(line)) {

relations.AddRange(getRelationsFromMultiLine(line));

}

else {

relations.AddRange(getRelationsFromLine(line));

}

return relations;

}

private static bool containsMultipleStatements(string line) {

if (line.Contains(".") || line.Contains(";") || line.Contains("Tb ") || line.Contains("Te ") || line.Contains("Ta "))

return true;

return false;

}

private List<string> getRelationsFromMultiLine(string line) {

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

string[] lines = line.Split(new string[] { ". ", "; " }, StringSplitOptions.RemoveEmptyEntries);

if (line.Contains("; Tt Tx") || line.Contains(". Tt Tx") || line.Contains("; Tt Tp") || line.Contains(". Tt Tp")) {

//two separate relations and also a comparative between one item from each to be created. Left then right.

string termBlock = getTermBlock(lines[1]);

string postBlock = getPostTermBlockPattern(lines[1]);

List<string> leftRelation = getRelationsFromLine(lines[0]);

relations.AddRange(leftRelation);

string leftItem = getFirstItem(leftRelation[0]);

List<string> rightRelation = getRelationsFromLine(postBlock, false);

relations.AddRange(rightRelation);

string rightItem = getFirstItem(postBlock);

relations.Add(formRelation(leftItem, rightItem, termBlock));

if (lines.Count() > 2) {

relations.AddRange(getRelationsFromMultiLine((line.Replace(lines[0], "")).Replace(lines[1], "")));

}

}

else if (line.Contains("; Tx") || line.Contains(". Tx") || line.Contains("; Tp") || line.Contains(". Tp")) {

//two separate relations and also a comparative between one item from each to be created. Right then left.

string termBlock = getTermBlock(lines[1]);

string postBlock = getPostTermBlockPattern(lines[1]);

List<string> leftRelation = getRelationsFromLine(lines[0]);

relations.AddRange(leftRelation);

string leftItem = getFirstItem(leftRelation[0]);

List<string> rightRelation = getRelationsFromLine(postBlock, false);

relations.AddRange(rightRelation);

string rightItem = getFirstItem(postBlock);

relations.Add(formRelation(rightItem, leftItem, termBlock));

if (lines.Count() > 2) {

relations.AddRange(getRelationsFromMultiLine((line.Replace(lines[0], "")).Replace(lines[1].Trim(), "")));

}

}

else if (line.Contains("; Td") || line.Contains(". Td")) {

//an additional relation must be added to the first-mentioned item.

List<string> leftRelation = getRelationsFromLine(lines[0]);

relations.AddRange(leftRelation);

string leftItem = getFirstItem(leftRelation[0]);

relations.AddRange(getRelationsFromMultiLine(leftItem + line.Substring(lines[0].Length + ";".Length)));

}

else if (line.Contains("Ta Td")) {

//form two negatives relating to first item, not left-preceding item.

string leftRelation = line.Substring(0, line.IndexOf("Ta "));

string leftItem = getFirstItem(leftRelation);

relations.AddRange(getRelationsFromLine(leftRelation));

relations.AddRange(getRelationsFromLine(leftItem + line.Substring(line.IndexOf("Ta ") + 2)));

}

else if (line.Contains("Tf")) {

//a negative/comparative relation and a second relation to the former item to be created

int i = 0;

}

else if (line.Contains("Tl")) {

//a negative/comparative relation and a second relation to the latter item to be created

int i = 0;

}

else if (line.Contains("Tt")) {

//a second relation must be formed with an item from the first relation

int i = 0;

}

else if (!line.Contains(";") && !line.Contains(".") && line.Contains("Td") && line.Contains("Te")) {

//a second relation must be formed with the first item of the first relation

relations.AddRange(makeEitherOrRelations(line));

}

else if (line.Contains("Tb")) {

if (isSeparatingBut(line)) {

lines = line.Split(new string[] { ". ", "; ", "Tb " }, StringSplitOptions.RemoveEmptyEntries);

//two relations to the leftmost item

List<string> foundRels = getRelationsFromLine(lines[0]);

relations.AddRange(foundRels);

string leftItem = getFirstItem(foundRels[0]);

foundRels = getRelationsFromLine(leftItem + " " + lines[1], false);

relations.AddRange(foundRels);

if (lines.Count() > 2) {

relations.AddRange(getRelationsFromMultiLine(line.Substring(lines[0].Length + lines[1].Length + "Tb ".Length)));

}

}

else {

line = line.Remove(line.IndexOf("Tb"), 3);

relations.AddRange(getRelationsFromLine(line));

}

}

else {

foreach (string l in lines) {

relations.AddRange(getRelationsFromLine(l, false));

}

}

return relations;

}

private List<string> makeEitherOrRelations(string line) {

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

bool eitherOr = line.Contains("Td Te"); // C (C) Td Te C Te C (Te C)\* || C (C) Td C Te C (Te C)\*

string[] lines = line.Split(new string[] { "Te " }, StringSplitOptions.RemoveEmptyEntries);

string firstRelationLine = eitherOr ? lines[0] + lines[1] : lines[0];

string secondRelationLine = eitherOr ? lines[2] : lines[1];

List<string> leftRelation = getRelationsFromLine(firstRelationLine);

relations.AddRange(leftRelation);

string leftItem = getFirstItem(leftRelation[0]);

relations.Add(leftItem + Representation.Relations.Negative + getFirstCat(secondRelationLine));

int linesDone = eitherOr ? 3 : 2;

if (lines.Count() > linesDone) {

for (int i = linesDone; i < lines.Count(); i++) {

relations.Add(leftItem + Representation.Relations.Negative + getFirstCat(lines[i]));

}

}

return relations;

}

private bool isSeparatingBut(string line) {

if (line.IndexOf("Tb") > 3) {

return true;

}

return false;

}

private static string getTermBlock(string p) {

string terms = "";

while (p[0] == 'T') {

terms += p.Substring(0, p.IndexOf(' ') + 1);

p = p.Remove(0, p.IndexOf(' ') + 1);

}

return terms;

}

private static string getPostTermBlockPattern(string p) {

while (p[0] == 'T') {

p = p.Remove(0, p.IndexOf(' ') + 1);

}

return p.Trim();

}

/// <summary>

/// gets first Cat from string

/// </summary>

/// <param name="p"></param>

/// <returns></returns>

private static string getFirstItem(string p) {

string result = p[0].ToString();

int i = 1;

int aux = 0;

while (i < p.Length && Int32.TryParse(p[i].ToString(), out aux)) {

result += p[i];

i++;

}

return result;

}

/// <summary>

/// Gets first Cat from string beginning with a Term

/// </summary>

/// <param name="p"></param>

/// <returns></returns>

private static string getFirstCat(string p) {

while (p[0] == 'T') {

p = p.Substring(p.IndexOf(' ') + 1);

}

if (p.IndexOf(' ') == -1) {

return p;

}

return p.Substring(0, p.IndexOf(' '));

}

private List<string> getRelationsFromLine(string line, bool firstLine = true) {

line = line.Trim();

if (firstLine && line.StartsWith("Tt")) {

line = line.Substring(3);

}

if (isCatPair(line)) {

return new List<string> { line.Substring(0, line.IndexOf(" ")) + Representation.Relations.Positive + line.Substring(line.IndexOf(" ") + 1) };

}

else if (isNegCatPair(line)) {

return getNegativeFromPair(line);

}

else if (hasSubclause(line)) {

string leftpart = line.Substring(0, line.IndexOf(",")).Trim();

string subclause = line.Substring(line.IndexOf(",") + 1);

subclause = subclause.Substring(0, subclause.IndexOf(",")).Trim();

string rightpart = line.Substring(line.LastIndexOf(",")+2);

List<string> results = getRelationsFromLine(leftpart + " " + subclause);

results.AddRange(getRelationsFromLine(leftpart + " " + rightpart));

return results;

}

else if (line.StartsWith("Td") && firstLine) {

//Neither/Nor line - all items should be disassociated to each other, tho' some may be in same category

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

List<string> items = getAllCatItems(line);

int j = 0;

for (int i = items.Count - 1; i > 0; i--) {

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

if (items[j][0] == items[j + x][0]) {

continue;

}

result.Add(items[j] + Relations.Negative + items[j + x]);

}

j++;

}

return result;

}

else if (line.StartsWith("Td") && !firstLine) {

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

try {

string firstTag = line.Substring(3, line.IndexOf(' ', 3) - 3);

string secondTag = line.Substring(line.IndexOf(firstTag) + firstTag.Length + 1);

secondTag = secondTag.Substring(0, secondTag.IndexOf(' '));

if (Tagger.isCatTag(secondTag)) {

if (line == "Td " + firstTag + " " + secondTag) {

result.Add(firstTag + Relations.Negative + secondTag);

}

else {

//still more to come

result.Add(firstTag + Relations.Negative + secondTag);

result.AddRange(getRelationsFromLine(line.Substring(line.IndexOf(secondTag) + secondTag.Length).Trim(), firstLine));

}

}

else {

result = getRelationsFromLine(line.Substring(line.IndexOf(' ') + 1), firstLine);

}

return result;

}

catch (IndexOutOfRangeException ex) {

return result;

}

}

else if ((line.StartsWith("Te") && line.Substring(3).Contains("Te")) || (! line.StartsWith("Te") && line.Contains("Te"))) {

return makeEitherOrRelations(line);

}

else if (!line.StartsWith("T") && !line.Contains("Tp")) {

return formRelationsUsingBuffer(line);

}

else if (line.Contains("Tp")) {

//first remove any category initials that are not comparative

line = removeNonCompCats(line);

return formRelationsUsingBuffer(line);

}

else {

throw new ParserException("Unable to handle tag pattern: " + line);

}

}

private bool hasSubclause(string line) {

return line.Contains(",");

}

private List<string> getAllCatItems(string line) {

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

string[] bits = line.Split(new char[] { ' ' });

foreach (string s in bits) {

if (s[0] != 'T') {

items.Add(s);

}

}

return items;

}

private string removeNonCompCats(string line) {

foreach (string bit in line.Split(new char[] { ' ' })) {

if (bit.Length == 1) {

if (!keyCategories.Contains(bit[0])) {

line = line.Remove(line.IndexOf(bit + " "), 2);

}

}

}

return line;

}

private List<string> formRelationsUsingBuffer(string line) {

try {

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

foreach (string bit in line.Split(new char[] { ' ' })) {

if (!buffer.isEmpty()) {

if (buffer.ToString().Length == 2 && buffer.ToString()[0] == bit[0]) {

//two items of the same category next to each other should not be related

buffer.Clear();

buffer.Add(bit);

continue;

}

}

buffer.Add(bit);

if (PatternBank.holdsTagPattern(buffer)) {

result.Add(makeRelation(buffer.ToString()));

buffer.Clear();

buffer.Add(bit);

}

}

buffer.Clear();

return result;

}

catch (Exception e) {

throw e;

}

finally {

buffer.Clear();

}

}

private string makeRelation(string p) {

try {

string[] bits = p.Split(new char[] { ',', ' ' }, StringSplitOptions.RemoveEmptyEntries);

switch (PatternBank.getPatternNumber(p)) {

case 0: return bits[0] + Relations.Positive + bits[1];

case 1: return bits[0] + Relations.Negative + bits[2];

case 2:

case 3: return formRelation(bits[0], bits[4], bits[1] + " " + bits[2] + " " + bits[3]);

case 4: return twoTermRelative(bits[0], bits[1], bits[2], bits[3], bits[4]);

case 5: return twoTermRelative(bits[0], bits[3], bits[1], bits[2], bits[4]);

case 6: return oneTermRelative(bits[0], bits[1], bits[2], bits[3]);

case 7: return oneTermRelative(bits[0], bits[1], bits[2], bits[3]);

case 8: return oneTermRelative(bits[0], bits[2], bits[1], bits[3]);

case 9: return oneTermRelative(bits[0], bits[2][3].ToString(), bits[1], bits[3]);

case 10: return oneTermRelative(bits[0], keyCategories[0].ToString(), bits[1], bits[2]);

case 11: return twoTermRelative(bits[0], keyCategories[0].ToString(), bits[1], bits[2], bits[3]);

case 12: return bits[0] + Relations.Positive + keyCategories[0].ToString() + (bits[1][3] == '-' ? MIN : MAX);

case 13: return bits[0] + Relations.Negative + keyCategories[0].ToString() + (bits[2][3] == '-' ? MIN : MAX);

case 14: return bits[0] + Relations.makeRelatedCat(bits[3]) + Relations.Positive + Relations.makeSemanticField(bits[1] + "-" + bits[2]);

case 15: return twoTermRelative(bits[3], bits[1][3].ToString(), bits[0], bits[2], bits[4]);

case 16: return twoTermRelative(bits[0], keyCategories[0].ToString(), "1", bits[2], bits[3]);

default:

throw new ParserException("No logic to handle pattern number " + PatternBank.getPatternNumber(p));

}

}

catch (Exception e) {

throw e;

}

}

private static string oneTermRelative(string left, string owned, string term, string right) {

string direction = term.Contains("+") ? Relations.GreaterThan : Relations.LessThan;

string relatedCat = Relations.makeRelatedCat(owned);

if (Tagger.isCatTag(right)) {

return left + relatedCat + direction + right + relatedCat;

}

return left + relatedCat + direction + right.Substring(3, right.Length - 4);

}

private static string twoTermRelative(string left, string owned, string amount, string comparator, string right) {

string relatedCat = Relations.makeRelatedCat(owned);

string difference = amount.Contains('(') ? amount.Substring(3, amount.Length - 4) : amount;

if (comparator.Contains("+")) {

return left + relatedCat + Relations.Subtract + right + relatedCat + Relations.Positive + difference;

}

else {

return right + relatedCat + Relations.Subtract + left + relatedCat + Relations.Positive + difference;

}

}

private string formRelation(string leftItem, string rightItem, string termBlock) {

string[] bits = termBlock.Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);

string direction = "";

string amount = "";

string category = keyCategories[0].ToString();

foreach (string bit in bits) {

if (bit[1] == 'x') {

amount = bit.Substring(3, bit.Length - 4);

}

if (bit[1] == 'q') {

category = bit.Substring(3, 1);

}

if (bit[1] == 'p') {

direction = bit.Substring(3, 1);

}

}

if (string.IsNullOrEmpty(amount)) {

//unquantified relation

direction = direction == "-" ? Relations.LessThan : Relations.GreaterThan;

return leftItem + "(" + category + ")" + direction + rightItem + "(" + category + ")";

}

else {

if (direction == "-") {

return rightItem + "(" + category + ")-" + leftItem + "(" + category + ")=" + amount;

}

else {

return leftItem + "(" + category + ")-" + rightItem + "(" + category + ")=" + amount;

}

}

}

private static bool isCatPair(string line) {

string[] words = line.Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);

if (words.Count() == 2) {

return true;

}

return false;

}

private static bool isNegCatPair(string line) {

string[] words = line.Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);

if (words.Count() == 3 && TermsDictionary.isNegative(words[0]) && Tagger.isCatTag(words[1]) && Tagger.isCatTag(words[2])) {

return true;

}

if (words.Count() == 3 && TermsDictionary.isNegative(words[1]) && Tagger.isCatTag(words[0]) && Tagger.isCatTag(words[2])) {

return true;

}

if (words.Count() == 3 && TermsDictionary.isNegative(words[2]) && Tagger.isCatTag(words[0]) && Tagger.isCatTag(words[1])) {

return true;

}

return false;

}

private static List<string> getNegativeFromPair(string line) {

string[] words = line.Split(new char[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);

string left = "";

string right = "";

foreach (string word in words) {

if (TermsDictionary.isNegative(word)) {

continue;

}

if (string.IsNullOrEmpty(left)) {

left = word;

}

right = word;

}

return new List<string> { left + Relations.Negative + right };

}

}

}

6c. CZParser – ParsingBuffer.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace CZParser

{

internal class ParsingBuffer

{

private string[] items;

private int size;

internal ParsingBuffer(int n) {

size = n;

items = new string[size];

}

public override string ToString() {

string result = "";

foreach (string item in items) {

if (string.IsNullOrEmpty(item)) {

continue;

}

if (!string.IsNullOrEmpty(result)) {

result += ",";

}

result += item;

}

return result;

}

internal void Clear() {

this.items = new string[size];

}

internal void Add(string item) {

int index = firstNonEmpty();

if (index == -1) {

throw new ArgumentException("Parsing buffer is full; cannot add " + item + " to " + this.ToString());

}

items[index] = item;

}

private int firstNonEmpty() {

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

if (string.IsNullOrEmpty(items[i])) {

return i;

}

}

return -1;

}

internal bool isEmpty() {

return items == null || string.IsNullOrEmpty(items[0]);

}

internal bool Contains(string p) {

foreach (string item in items) {

if (item == null) {

return false;

}

if (item == p)

return true;

else {

foreach (string bit in item.Split(new char[] { '{', '}', ',' }, StringSplitOptions.RemoveEmptyEntries)) {

if (bit == p)

return true;

}

}

}

return false;

}

internal void dropNonTermTags() {

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

string newItem = "";

if (items[i] != null) {

string[] bits = items[i].Split(new char[] { ',', '{', '}' }, StringSplitOptions.RemoveEmptyEntries);

foreach (string b in bits) {

if (b[0] == 'T') {

//never more than one TermTags in a tag.

newItem = b;

}

}

items[i] = newItem;

}

}

}

internal bool hasCombinedCats() {

if (isEmpty()) {

return false;

}

return !this.ToString().Contains("T");

}

internal bool hasMixedTags() {

if (isEmpty()) {

return false;

}

if (!this.ToString().Contains("T")) {

return false;

}

string[] bits = this.ToString().Split(new char[] { ',', '{', '}' }, StringSplitOptions.RemoveEmptyEntries);

foreach (string t in bits) {

if (Tagger.isCatTag(t)) {

return true;

}

}

return false;

}

internal bool stretchesOverWhitespace() {

//Buffers of Combined cat tags ("B2,B") do not stretch

//Buffers of Term tags (other than To) do stretch

return !this.hasCombinedCats();

}

internal string pullCategoryTitle() {

if (string.IsNullOrEmpty(items[1]) && !string.IsNullOrEmpty(items[0])) {

//if we've used more than one buffer space, cannot assume mention of category title

foreach (string bit in items[0].Split(new char[] { ',', ' ' }, StringSplitOptions.RemoveEmptyEntries)) {

if (Tagger.isCatTag(bit) && bit.Length == 1) {

return bit;

}

}

}

return null;

}

}

}

Appendix B7a LogixTests – CategoryTest.cs

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Logix;

using System.Collections.Generic;

namespace LogixTests

{

[TestClass]

public class CategoryTest

{

private CategoryBuilder catBuilder;

private RelationFactory relationBuilder;

[TestMethod]

public void Create\_And\_Test\_Line() {

catBuilder.newCat();

catBuilder.setIdentifier('B');

catBuilder.setSize(5);

catBuilder.setKeyword("Numeric");

var cat1 = catBuilder.build();

Assert.AreEqual('B', cat1.identifier);

object[] list = { 5, 10, 15, 20, 25 };

cat1.enterValues(list);

object result = cat1.retrieveValue("B3");

Assert.AreEqual("15", result.ToString());

}

[TestInitialize]

public void Initialize() {

this.relationBuilder = RelationFactory.getInstance();

this.catBuilder = new CategoryBuilder();

}

[TestMethod]

public void Test\_AllButOneFound() {

catBuilder.quickSet('A', 5);

Category cat = catBuilder.build();

cat.addRelation("A1", "B1");

cat.addRelation("A2", "B2");

cat.addRelation("A3", "B3");

cat.addRelation("A4", "B4");

string AforB5 = cat.checkForMatch("B5");

Assert.AreEqual("A5", AforB5);

}

[TestMethod]

public void Test\_InconclusiveResult() {

catBuilder.quickSet('A', 5);

Category cat = catBuilder.build();

cat.addRelation("A1", "B1");

string AforB = cat.checkForMatch("B5");

Assert.IsNull(AforB);

}

[TestMethod]

public void Consider\_Positive\_Rule() {

catBuilder.quickSet('A', 5);

Category cat = catBuilder.build();

Relation r = relationBuilder.createRelation("A1=B3");

List<Relation> rules = cat.considerRelationToCategory(r, false);

Assert.AreEqual("A1", cat.checkForMatch("B3", Category.Rows.Positives));

}

[TestMethod]

public void Consider\_Relative\_Rule() {

catBuilder.quickSet('B', 5);

Category cat = catBuilder.build();

cat.setKeyword("Numeric");

cat.enterValues(new object[] { 5, 10, 15, 20, 25 });

cat.addRelation("B1", "A3");

Relation r = relationBuilder.createRelation("A1(B)-A3(B)=5");

List<Relation> rules = cat.considerRelationToCategory(r, false);

Assert.AreEqual("B2", cat.checkForMatch("A1", Category.Rows.Positives));

}

}

}

Appendix B7b LogixTests – DeducerTest.cs

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Logix;

using System.Collections.Generic;

using Representation;

namespace LogixTests

{

[TestClass]

public class DeducerTest

{

private RelationFactory relationBuilder;

private Representation.Results results;

[TestInitialize]

public void Initialise() {

relationBuilder = RelationFactory.getInstance();

}

[TestMethod]

public void Create\_Deducer() {

Deducer brains = new Deducer(4,5);

List<Category> cats = brains.getCategoryCollection();

Assert.AreEqual(4, cats.Count);

}

[TestMethod]

public void Retrieve\_Line\_From\_Identifier() {

Deducer brains = new Deducer(4, 5);

Category l = brains.getCategoryFromIdentifier('C');

Assert.IsNotNull(l);

l.considerRelationToCategory(relationBuilder.createRelation("C1=B3")

, false);

string consideredItem = l.checkForMatch("B3");

Assert.AreEqual("C1", consideredItem);

}

[TestMethod]

public void Set\_Clues() {

List<string> clues = new List<string>{ "A3=D1", "A2(C)-A3(C)=2", "A2=B3"

, "B2=D3", "D2!=C2" };

Deducer brains = new Deducer(4, 3);

brains.setClues(clues);

Assert.AreNotEqual(0, brains.getRemainingClueCount());

}

[TestMethod]

public void First\_Problem\_Test() {

List<string> clues = new List<string> { "A3=D1", "A2(C)-A3(C)=2", "A2=B3"

, "B2=D3", "D2!=C2" };

Deducer brains = new Deducer(4, 3, new string[] {"", "", "days", ""});

brains.Concluded += brains\_Concluded;

brains.setClues(clues);

brains.enterCategoryValues('C', new object[] {"Monday", "Wednesday"

, "Friday"});

int[,] matrix = brains.Go();

Assert.IsNotNull(matrix);

int[,] providedMatrix = new int[3,4]{{1,2,1,3},{2,3,3,2},{3,1,2,1}};

Assert.IsTrue(solutionsMatch(providedMatrix, matrix));

}

[TestMethod]

public void Second\_Problem\_Test() {

List<string> clues = new List<string> { "C2=D1", "A1=B1", "A3=C3"

, "B3!=D3", "B2=D2"};

Deducer brains = new Deducer(4, 3, new string[] { "", "", "", "" });

brains.Concluded += brains\_Concluded;

brains.setClues(clues);

int[,] matrix = brains.Go();

Assert.IsNotNull(matrix);

int[,] providedMatrix = new int[3, 4] { { 1, 1, 1, 3 }, { 2, 3, 2, 1 }

, { 3, 2, 3, 2 } };

Assert.IsTrue(solutionsMatch(providedMatrix, matrix));

}

[TestMethod]

public void Third\_Problem\_Test() {

List<string> clues = new List<string> { "C1=D3", "C1!=B2", "B2!=C2"

, "C2!=A4", "B1=D1", "B4=A3", "B4!=D4", "C4=A2", "C4!=D2"

, "B3(A)-C3(A)=1" };

Deducer brains = new Deducer(4, 4, new string[] { "Numeric", "", "", "" });

brains.enterCategoryValues('A', new object[] { 1, 2, 3, 4 });

brains.Concluded += brains\_Concluded;

brains.setClues(clues);

int[,] matrix = brains.Go();

Assert.IsNotNull(matrix);

int[,] providedMatrix = new int[4, 4] { { 1, 1, 2, 1 }, { 2, 2, 4, 4 }

, { 3, 4, 3, 2 }, { 4, 3, 1, 3 } };

Assert.IsTrue(solutionsMatch(providedMatrix, matrix));

}

[TestMethod]

public void Fourth\_Problem\_Test() {

List<string> clues = new List<string> { "A1!=B4", "A2!=B1","A2!=B2"

,"A3!=B5","A5!=B3","C3=D3","D3!=B5","D4=B4","B4!=A2"

,"A2!=D1","B1!=A1" };

clues.AddRange(new List<string> {"A5=C1","C2=B2","B2!=D5","A4=B5"

,"A3=D2","A3!=C5"});

Deducer brains = new Deducer(4, 5, new string[] { "", "", "", ""});

brains.Concluded += brains\_Concluded;

brains.setClues(clues);

int[,] matrix = brains.Go();

Assert.IsNotNull(matrix);

int[,] providedMatrix = new int[5, 4] { { 1, 2, 2, 1 }, { 2, 3, 3, 3 }

, { 3, 1, 4, 2}, { 4, 5, 5, 5 }, { 5, 4, 1, 4 } };

Assert.IsTrue(solutionsMatch(providedMatrix, matrix));

}

[TestMethod]

public void Fifth\_Problem\_Test() {

List<string> clues = new List<string> {"B1(D)-C4(D)=2", "B1!=C4"

, "B1=C2","B1!=A2","A2!=B5","A1!=D1","A1=B2","A5=C5"

,"A5(D)-B3(D)=1", "A5!=B3","B4(D)<A3(D)", "B4!=A3"};

clues.AddRange(new List<string> { "B4(D)>C1(D)", "C1(D)<A3(D)", "A4!=C2"

, "A4!=C4", "A4!=C5", "?A4=C1?A4(D)>C3(D):A4(D)>C1(D)"

, "A4!=D1", "B4!=C1" });

Deducer brains = new Deducer(4, 5, new string[] { "", "", "", "Numeric" });

brains.Concluded += brains\_Concluded;

brains.setClues(clues);

brains.enterCategoryValues('D', new object[] { 1, 2, 3, 5, 6 });

int[,] matrix = brains.Go();

Assert.IsNotNull(matrix);

int[,] providedMatrix = new int[5, 4] { { 1, 2, 4, 3 }, { 2, 3, 1, 1 }

, { 3, 1, 2, 4 }, { 4, 5, 3, 5 }, { 5, 4, 5, 2 } };

Assert.IsTrue(solutionsMatch(providedMatrix, matrix));

}

private bool solutionsMatch(int[,] providedMatrix, int[,] matrix) {

List<string> strung1 = new List<string>();

List<string> strung2 = new List<string>();

for (int x = 0; x < matrix.GetLength(0); x++) {

string item1 = "";

string item2 = "";

for (int y = 0; y < matrix.GetLength(1); y++) {

item1 += providedMatrix[x, y];

item2 += matrix[x, y];

}

strung1.Add(item1);

strung2.Add(item2);

}

foreach (string s in strung1) {

if (!strung2.Contains(s)) {

return false;

}

}

foreach (string s in strung2) {

if (!strung1.Contains(s)) {

return false;

}

}

return true;

}

void brains\_Concluded(Deducer sender, SolveCompleteArgs e) {

results = new Results(e.isSuccessful, e.turns, e.timeTaken);

}

}

}

Appendix B7c LogixTests – RelationTest.cs

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Logix;

namespace LogixTests

{

[TestClass]

public class RelationTest

{

private RelationFactory relationBuilder;

[TestInitialize]

public void Initialize() {

this.relationBuilder = RelationFactory.getInstance();

}

[TestMethod]

public void Create\_Correct\_Relation\_Type() {

Relation r = relationBuilder.createRelation("A1(B)-A3(B)=5");

Assert.AreEqual(r.GetType().Name, "RelativeRelation");

Assert.IsTrue(r.isRelative());

r = relationBuilder.createRelation("A1=B3");

Assert.AreEqual(r.GetType().Name, "DirectRelation");

Assert.IsTrue(r.isDirect());

r = relationBuilder.createRelation("?A1=B2?A1(C)>B1(C):A1(C)>B2(C)");

Assert.AreEqual(r.GetType().Name, "ConditionalRelation");

Assert.IsTrue(r.isConditional());

}

[TestMethod]

public void Detect\_Positive\_Or\_Negative() {

Relation r = relationBuilder.createRelation("A1=B3");

Assert.IsTrue(r.isPositive());

r = relationBuilder.createRelation("A1!=B3");

Assert.IsFalse(r.isPositive());

}

[TestMethod]

public void Item\_Retrieval() {

Relation r = relationBuilder.createRelation("A1=B3");

string bItem = r.getBaseItem('B');

Assert.AreEqual("B3", bItem);

string aItem = r.getRelatedItem('B');

Assert.AreEqual("A1", aItem);

}

[TestMethod]

public void Check\_Representation() {

Relation r = relationBuilder.createRelation("A1(C)-B1(C)=5");

Assert.IsTrue(Representation.Relations.isQuantified(r.getRule()));

r = relationBuilder.createRelation("A1(C)>B1(C)");

Assert.IsFalse(Representation.Relations.isQuantified(r.getRule()));

}

}

}

Appendix B8 – ParserTest.cs

using System;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Representation;

using System.Collections.Generic;

using System.Xml;

using CZParser;

namespace ParserTests

{

[TestClass]

public class ParserTest

{

private List<Puzzle> puzzles;

[TestInitialize]

public void loadPuzzles() {

puzzles = new List<Puzzle>();

XmlDocument sourceDoc = new XmlDocument();

sourceDoc.LoadXml(ParserTests.Properties.Resources.puzzles);

sourceDoc.Normalize();

XmlNodeList xmlPuzzles = sourceDoc.GetElementsByTagName("puzzle");

foreach (XmlNode n in xmlPuzzles) {

if (string.IsNullOrEmpty(n["title"].InnerText)) {

continue;

}

puzzles.Add(new Puzzle(n));

}

}

[TestMethod]

public void Initialise\_Parser() {

Puzzle p = puzzles[1];

Parser parser = new Parser(p);

Assert.IsNotNull(parser);

}

[TestMethod]

public void Check\_Dictionary\_Creation() {

Puzzle p = puzzles[0];

Parser parser = new Parser(p);

List<string> puzzle1Items = new List<string> { "brendan", "briese", "gareth"

, "gale", "zachary", "zeffer", "baseball", "cap", "bowler", "hat"

, "flat", "cap", "monday", "wednesday", "friday", "dual", "carriageway"

, "river", "tree" };

for (int i = 0; i < puzzle1Items.Count; i++) {

Assert.AreEqual(puzzle1Items[i], parser.tagger.catWords.getItems()[i]);

}

List<string> puz1Quants = new List<string> { "day", "days", "night", "nights" };

for (int i = 0; i < puz1Quants.Count; i++) {

Assert.AreEqual(puz1Quants[i], parser.tagger.terms.getQuantifiers()[i]);

}

}

[TestMethod]

public void Check\_Dictionary\_Tagging() {

Puzzle p = puzzles[14];

Parser parser = new Parser(p);

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

string auxCat1 = "";

string auxCat2 = "";

var words = Tagger.separateWordsAndPunctuation("Oliver Newton does not play the

part of either a corner shop owner or a graphic designer working from a

studio in their attic.");

foreach (string word in words) {

if (word.Length == 1 && char.IsPunctuation(word[0])) {

firstTagPattern.Add(word);

continue;

}

auxCat1 = parser.tagger.catWords.findItemMatches(word);

auxCat2 = parser.tagger.terms.defineItem(word);

firstTagPattern.Add(string.IsNullOrEmpty(auxCat1) ? auxCat2

: string.IsNullOrEmpty(auxCat2) ? auxCat1 : auxCat1 + "," + auxCat2);

}

List<string> testTags = new List<string> {"A5", "B4", "Td", "To", "Te", "D4"

, "D4", "Te", "D2", "." };

List<string> producedTags = justTags(firstTagPattern);

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

Assert.AreEqual(testTags[i], producedTags[i]);

}

}

private List<string> justTags(List<string> firstTagPattern) {

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

foreach (string item in firstTagPattern) {

if (!string.IsNullOrEmpty(item)) {

result.Add(item);

}

}

return result;

}

[TestMethod]

public void Check\_First\_Tagging() {

Puzzle p = puzzles[1];

Parser parser = new Parser(p);

List<string> taggedClues = parser.tagger.tagClues(p.getClues());

Assert.IsNotNull(taggedClues);

List<string> correctTags = new List<string>() {"C2 D1", "A1 B1 ; A3 C3"

, "B3 Td D3", "B2 D2" };

Assert.AreEqual(correctTags.Count, taggedClues.Count);

foreach (string clue in taggedClues) {

Assert.IsTrue(correctTags.Contains(clue));

}

}

[TestMethod]

public void Check\_First\_Translating() {

Puzzle p = puzzles[1];

Parser parser = new Parser(p);

List<string> relations = parser.Read();

List<string> manualRelations = new List<string> {"C2=D1", "A1=B1", "A3=C3"

, "B3!=D3", "B2=D2" };

foreach (string rule in manualRelations) {

Assert.IsTrue(relations.Contains(rule));

}

foreach (string rule in relations) {

Assert.IsTrue(manualRelations.Contains(rule));

}

}

[TestMethod]

public void Check\_Second\_Tagging() {

Puzzle p = puzzles[0];

Parser parser = new Parser(p);

List<string> taggedClues = parser.tagger.tagClues(p.getClues());

Assert.IsNotNull(taggedClues);

List<string> correctTags = new List<string>() { "A3 D1 ; Tt Tx(2) Tq(C) Tp(-) A2

B3", "B2 D3", "Td C2 D2" };

Assert.AreEqual(correctTags.Count, taggedClues.Count);

foreach (string clue in taggedClues) {

Assert.IsTrue(correctTags.Contains(clue));

}

}

[TestMethod]

public void Check\_Second\_Translating() {

Puzzle p = puzzles[0];

Parser parser = new Parser(p);

List<string> relations = parser.Read();

List<string> manualRelations = new List<string> { "A3=D1", "A2(C)-A3(C)=2"

, "A2=B3", "B2=D3", "C2!=D2" };

foreach (string rule in manualRelations) {

Assert.IsTrue(relations.Contains(rule));

}

foreach (string rule in relations) {

Assert.IsTrue(manualRelations.Contains(rule));

}

}

[TestMethod]

public void Check\_Third\_Tagging() {

Puzzle p = puzzles[3];

Parser parser = new Parser(p);

List<string> taggedClues = parser.tagger.tagClues(p.getClues());

Assert.IsNotNull(taggedClues);

List<string> correctTags = new List<string>() { "C1 D3 . Td B2 Td C2 Td A4"

, "B1 D1", "B4 A3 Td D4", "C4 A2 Td D2", "B3 Tx(1) Tp(+) C3" };

Assert.AreEqual(correctTags.Count, taggedClues.Count);

foreach (string clue in taggedClues) {

Assert.IsTrue(correctTags.Contains(clue));

}

}

[TestMethod]

public void Check\_Third\_Translating() {

Puzzle p = puzzles[3];

Parser parser = new Parser(p);

List<string> relations = parser.Read();

List<string> manualRelations = new List<string> {"C1=D3", "C1!=B2", "B2!=C2"

, "C2!=A4", "B1=D1", "B4=A3", "A3!=D4", "C4=A2", "A2!=D2"

, "B3(A)-C3(A)=1" };

foreach (string rule in manualRelations) {

Assert.IsTrue(relations.Contains(rule));

}

foreach (string rule in relations) {

Assert.IsTrue(manualRelations.Contains(rule));

}

}

[TestMethod]

public void Check\_Fourth\_Tagging() {

Puzzle p = puzzles[2];

Parser parser = new Parser(p);

List<string> taggedclues = parser.tagger.tagClues(p.getClues());

Assert.IsNotNull(taggedclues);

List<string> correctTags = new List<string>() { "A3 Tp(+) Tq(D) C3 Tb Tp(-)

Tq(D) B1", "B2 Tp(+) Tq(D) C4 Tb Tp(-) A2", "A1 Tp(+) Tq(D) B3 Tb Tp(-)

C1", "B4 Tx(£250) Tp(+) A4", "C2" };

Assert.AreEqual(correctTags.Count, taggedclues.Count);

foreach (string clue in taggedclues) {

Assert.IsTrue(correctTags.Contains(clue));

}

}

[TestMethod]

public void Check\_Fourth\_Translating() {

Puzzle p = puzzles[2];

Parser parser = new Parser(p);

List<string> relations = parser.Read();

List<string> manualRelations = new List<string> { "A3(D)>C3(D)", "A3(D)<B1(D)"

, "B2(D)>C4(D)", "B2(D)<A2(D)", "A1(D)>B3(D)", "A1(D)<C1(D)"

, "B4(D)-A4(D)=£250" };

foreach (string rule in manualRelations) {

Assert.IsTrue(relations.Contains(rule));

}

foreach (string rule in relations) {

Assert.IsTrue(manualRelations.Contains(rule));

}

}

[TestMethod]

public void Check\_Fifth\_Tagging() {

Puzzle p = puzzles[5];

Parser parser = new Parser(p);

List<string> taggedclues = parser.tagger.tagClues(p.getClues());

Assert.IsNotNull(taggedclues);

List<string> correctTags = new List<string>() { "A1 C4 ; Td B1", "A3 B4 ; B3

C3", "A2 Tg(woman) B2 Tg(male)", "D3 C2", "Td A4 Td C1 D4" };

Assert.AreEqual(correctTags.Count, taggedclues.Count);

foreach (string clue in taggedclues) {

Assert.IsTrue(correctTags.Contains(clue));

//Will fail until we can understand gender.

}

}

[TestMethod]

public void Check\_Fifth\_Translating() {

Puzzle p = puzzles[5];

Parser parser = new Parser(p);

List<string> relations = parser.Read();

//"A2=B2" is not the real rule we should have; it is the one we will get

until the test above passes

List<string> manualRelations = new List<string> { "A1=C4", "A1!=B1", "A3=B4"

, "B4=A3", "B3=C3", "A2=B2", "D3=C2", "A4!=C1", "A4!=D4", "C1!=D4" };

foreach (string rule in manualRelations) {

Assert.IsTrue(relations.Contains(rule));

}

foreach (string rule in relations) {

Assert.IsTrue(manualRelations.Contains(rule));

}

}

[TestMethod]

public void Check\_Sixth\_Tagging() {

Puzzle p = puzzles[7];

Parser parser = new Parser(p);

List<string> taggedclues = parser.tagger.tagClues(p.getClues());

Assert.IsNotNull(taggedclues);

List<string> correctTags = new List<string>() { "C1 D3","A2 B5 Tp(+) B1"

,"B2 D1 ; Tt Tp(+) A3","A1 D4 Tp(-) D5","B4 C5 ; A5 C2","D2 , Tp(+) B3 ,

Td A4 . B3 Td Ts(-)" };

Assert.AreEqual(correctTags.Count, taggedclues.Count);

foreach (string clue in taggedclues) {

Assert.IsTrue(correctTags.Contains(clue));

}

}

[TestMethod]

public void Check\_Sixth\_Translating() {

Puzzle p = puzzles[7];

Parser parser = new Parser(p);

List<string> relations = parser.Read();

List<string> manualRelations = new List<string> { "C1=D3","A2=B5","B5(C)>B1(C)"

,"B2=D1","B2(C)>A3(C)","A1=D4","D4(C)<D5(C)","B4=C5","A5=C2"

,"D2(C)>B3(C)","D2!=A4","B3!=C1" };

foreach (string rule in manualRelations) {

Assert.IsTrue(relations.Contains(rule));

}

foreach (string rule in relations) {

Assert.IsTrue(manualRelations.Contains(rule));

}

}

[TestMethod]

public void Check\_Seventh\_Tagging() {

Puzzle p = puzzles.Find(z => z.getId() == 14);

Parser parser = new Parser(p);

List<string> taggedclues = parser.tagger.tagClues(p.getClues());

Assert.IsNotNull(taggedclues);

List<string> correctTags = new List<string>() { "A5 B4 Td Te D4 Te D2","B3 C2 ;

B2 Td C5 Te D2","A2 Td B1 . A3 Td C1","A4 C4 Td D5 Te D1"

,"D4 C5 Td A3 ; B5 D5 Td C3"};

Assert.AreEqual(correctTags.Count, taggedclues.Count);

foreach (string clue in taggedclues) {

Assert.IsTrue(correctTags.Contains(clue));

}

}

[TestMethod]

public void Check\_Seventh\_Transating() {

Puzzle p = puzzles.Find(z => z.getId() == 14);

Parser parser = new Parser(p);

List<string> relations = parser.Read();

List<string> manualRelations = new List<string> { "A5=B4", "B4!=D4", "A5!=D2"

, "B3=C2", "B2!=C5", "B2!=D2", "A2!=B1", "A3!=C1", "A4=C4", "C4!=D5"

, "A4!=D1", "D4=C5", "C5!=A3", "B5=D5", "D5!=C3" };

foreach (string rule in manualRelations) {

Assert.IsTrue(relations.Contains(rule));

}

foreach (string rule in relations) {

Assert.IsTrue(manualRelations.Contains(rule));

}

}

[TestMethod]

public void Check\_Eighth\_Tagging() {

Puzzle p = puzzles.Find(z => z.getId() == 7);

Parser parser = new Parser(p);

List<string> taggedclues = parser.tagger.tagClues(p.getClues());

Assert.IsNotNull(taggedclues);

List<string> correctTags = new List<string>() { "C1 D3", "A2 B5 Tp(+) B1"

, "B2 D1 ; Tt Tp(+) A3", "A1 D4 Tp(-) D5", "B4 C5 ; A5 C2", "D2

, Tp(+) B3 , Td A4 . B3 Td Ts(-)" };

Assert.AreEqual(correctTags.Count, taggedclues.Count);

foreach (string clue in taggedclues) {

Assert.IsTrue(correctTags.Contains(clue));

}

}

[TestMethod]

public void Check\_Eighth\_Translating() {

Puzzle p = puzzles.Find(z => z.getId() == 7);

Parser parser = new Parser(p);

List<string> relations = parser.Read();

List<string> manualRelations = new List<string> { "C1=D3", "A2=B5",

"B5(C)>B1(C)", "B2=D1","B2(C)>A3(C)", "A1=D4","D4(C)<D5(C)"

, "B4=C5", "A5=C2","D2(C)>B3(C)", "D2!=A4","B3!=C1" };

foreach (string rule in manualRelations) {

Assert.IsTrue(relations.Contains(rule));

}

foreach (string rule in relations) {

Assert.IsTrue(manualRelations.Contains(rule));

}

}

[Ignore,TestMethod]

public void Transform\_XML\_Solution() {

XmlDocument sourceDoc = new XmlDocument();

sourceDoc.LoadXml(ParserTests.Properties.Resources.puzzles\_sample);

sourceDoc.Normalize();

XmlDocument newSource = new XmlDocument();

newSource.AppendChild(newSource.CreateElement("folio"));

var newRoot = newSource.GetElementsByTagName("folio").Item(0);

var folio = sourceDoc.ChildNodes[1];

for (int i = 0; i < 45 && i < folio.ChildNodes.Count; i++) {

newRoot.AppendChild(newSource.ImportNode(folio.ChildNodes[i], true));

var thispuzzle = newRoot.ChildNodes[i];

XmlNode solutionNode = sourceDoc.GetElementsByTagName("solution").Item(i);

String solution = solutionNode.InnerText;

var newSolNode = newSource.CreateElement("solution");

var oldSolNode = newSource.GetElementsByTagName("solution").Item(i);

foreach (string s in solution.Split(new string[] { "{{", "},{", "}}" }

, StringSplitOptions.RemoveEmptyEntries)) {

var row = newSource.CreateElement("row");

row.InnerText = s;

newSolNode.AppendChild(row);

}

thispuzzle["box"].ReplaceChild(newSolNode, oldSolNode);

}

newSource.Save("C:\\Users\\Abbie\\Documents\\GitHub\\CZ2012\\ParserTests

\\Resources\\puzzles-sample2.xml");

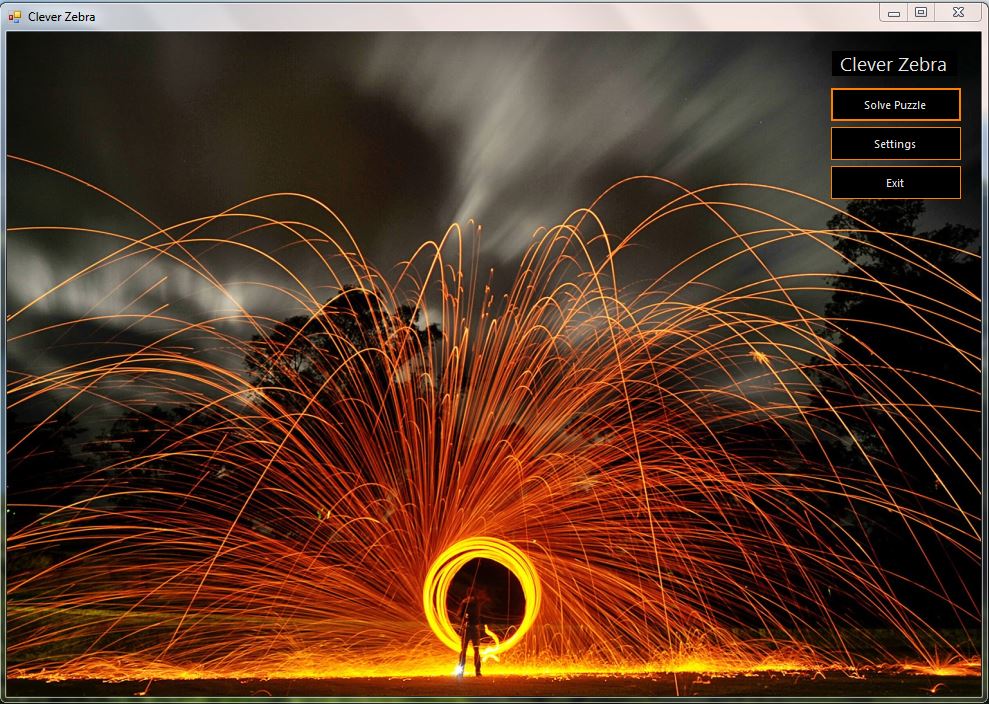
}

}

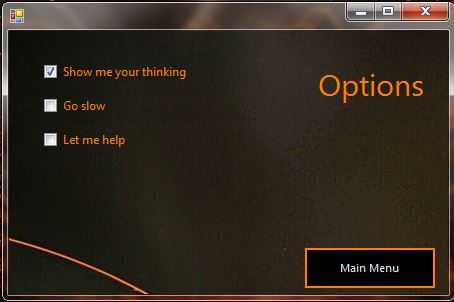
}

Appendix C – User Interface Screenshots

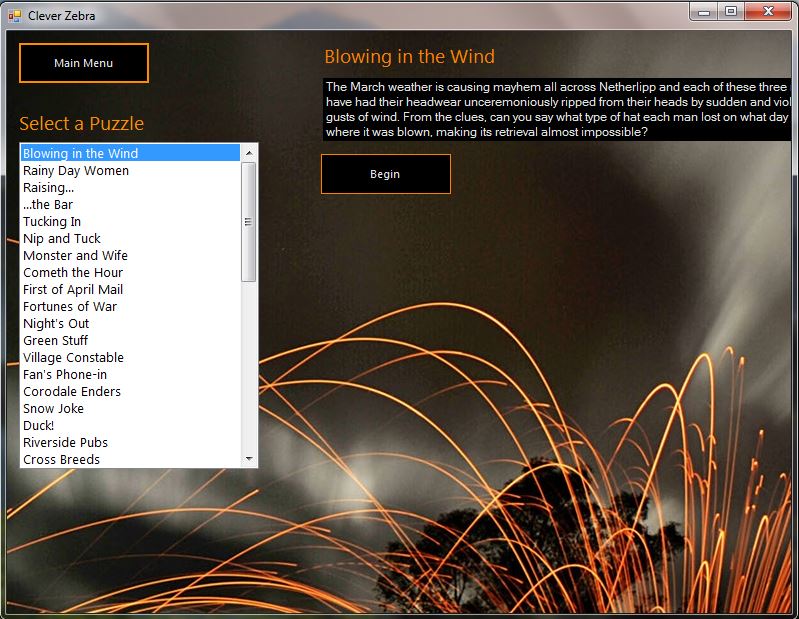
Main Menu



Options Menu



Puzzle Selection Page



Solving Window - Setup



Solving Window - Puzzle Complete



Appendix D – Test Results

