Programming Assignment 3: Logic Programming Medley

(Due December 2nd, 2021 at 7pm)

CSCI 4430 - Programming Languages, Fall 2021, Rensselaer Polytechnic Institute

Part 1: A Family Tree *(40%)*

One classic use for logic programming is to explore a family tree and reason about the relationships. For part 1 of this assignment, will be provided with a rule file in Prolog or Oz that encodes a chunk of the family tree for the House of Windsor, the current royal family of the United Kingdom. The tree contains relationships from Queen Victoria (coronated 1838) through to Elizabeth II (coronated 1953), the current queen as of 2021. The tree is constructed with facts in the form of one of the following predicates:

Provided Facts

- **parentOf(Parent, Child)**
- **married(Wife, Husband)** (For this fact, the wife comes first. The provided **spouse** rule below will allow for the order to be swapped if need be)
- **birthYear(Person, Year)** (For the provided family tree and any tested trees, you can assume that every person in the tree has a unique birth year)
- **monarch(Person)** (this final fact is defined as if the Person was ever crowned King or Queen)

Two sample rules are provided for you:

Provided Rules

- **spouse(Person, Spouse)**: Person is married to Spouse or Spouse is married to Person.
- **childOf(Child, Parent)**: Parent is a parent of Child.

Your assignment for part 1 then is the creation of the following rules:

Rules To Be Written:

1. **siblings(Person, Sibling)**: Person and Sibling are children of the same parents, and Person and Sibling are distinct people *(i.e. not the same person; You can't be your own sibling for the purposes of this assignment)*.
2. **firstCousins(Person, Cousin)**: Person and Cousin are cousins, meaning one of Person's parents is siblings with one of Cousin's parents.
3. **hasAncestor(Person, Ancestor)**: Person is either a child, grandchild, great-grandchild, et cetera, of Ancestor.
4. **hasDescendant(Person, Descendant)**: Person is either a parent, grandparent, great-grandparent, et cetera, of Descendant *(hint: This can be defined in terms of the ancestor rule)*.
5. **listAncestors(Person, Ancestors)**: Produce a list of all ancestors of Person.
6. **listDescendants(Person, Descendants)**: Produce a list of all descendants of Person.
7. **hasHeir(Person, Heir)**: if Person was a monarch, their heir is their oldest child *(hint: This is why the birthYear fact is included)*.
8. **hasSuccessor(Person, Successor)**: if Person was a monarch, their successor is any of their children who also were/are a monarch.
9. **heirIsSuccessor(Person)**: returns True if Person's original heir ended up succeeding them.

Example results of each rule

The grade for part one will be determined by your rules ability to be correctly used in inference. A good starting point to verify that you have a correct implementation of these rules is by testing out the following queries. With the above rules working properly, your program should be able to show, with the provided knowledge base, that:

1. Edward VIII and George VI are siblings
2. Anne Princess-Royal, and Lady Sarah are first cousins
3. Victoria is an ancestor of Prince Charles
4. Mary of Teck is not a descendant of Edward VII
5. George V's ancestors are Victoria, Prince Albert, Edward VII, and Alexandra
6. George VI's descendants are Elizabeth II, Princess Margaret, David Earl-of-Snowdon, Lady Sarah, Prince Charles, Anne Princess-Royal, Prince Andrew, and Prince Edward
7. Edward VIII is the heir to George V.
8. Both Edward VIII and George VI are successors to George V.
9. Empress Victoria (of Germany) is not the successor to Victoria, even though she is technically Victoria's heir (but not by the standards of the 19th century).

How to Run in Prolog:

Write your rule definitions in a file called **`relations.pl`**, replacing the hard-coded answers in the provided version of the file. At any time, you can use **`part1_tester.pl`** to see if your code for the rules is working. With the two above files in the same folder, along with the provided **`windsor.pl`**, run the following from the command line to perform a test:

```
swipl -f part1_tester.pl -g main
```

This should run the `main` function in `part1_tester.pl` and produce some output automatically.

While this method is good in a pinch, for this part it is recommended to load knowledge-base file and the rules file into the SWI-Prolog interpreter manually and play around with your rules. This can be done, for example, like:

```
?- [windsor, relations].
true.

?- hasHeir(P, H).
```

After the last line above is entered, pressing [;] on your keyboard several times should show all monarchs and their heirs unified with `P` and `H`. You can (and should) run other tests as well, to make sure your rules function like their descriptions dictate.

How to Run in Oz:

The family tree encoding and given rules are provided in **`windsor.oz`**. Write your rule definitions in the file **`relations.oz`**. The sample queries are provided for you in **`part1Tester.oz`**. This file can also be extended with your own custom tests. To test your relations simply verify by compiling **`relations.oz`** and **`part1Tester.oz`** and running **`part1Tester.oz`**

```
ozc -c windsor.oz #Only need to run this once
ozc -c relations.oz
ozc -c part1Tester.oz
ozengine part1Tester.ozf
```

Part 2: Microprocessor Constraint Language *(50%)*

Despite the recent silicon shortage in its industry, microprocessor manufacturing is one of the most key sectors of business in the modern age. For this problem, you are to implement a natural language processing program that will read in **definitions** of processors and **constraints** on the processor attributes, which are core count, area, and cost. Once several definitions are specified and all three attributes are constrained, your program should run a constraint computation to calculate how many of each processor should be placed on a common chip, such that the constraints are satisfied, and return these counts to be printed.

You can assume for the purposes of this assignment that the maximum count of any processor type arranged on a common chip is 16. Hence, results for any processor will range in values from 0 to 16 in integer steps. You can also assume that any group of sentences input into your program will contain at least one **definition** and at least one **constraint** for each of the three constrained values. Hence, do not worry about default values or error handling if you do not get enough information.

Processor **definitions** take on the following form:

Definition Sentence Blueprint

Processor type [ID] has [C] cores, uses [A] square centimeters, and costs [D] dollars.

- **ID**: a, b, c, . . . , x, y, z
- **C**: 1, 2, 3, . . .
- **A**: 1, 2, 3, . . .
- **D**: 1, 2, 3, . . .

Constraint Sentence Blueprint

Attribute **constraints** take on the following form:

[Attribute] [Imperative] be [[Comparison] [Value] | [Interval] [Range]].

- **Attribute**: (the |) (core count | area | cost)
  - Note that the empty space after the bar in the first parentheses is the empty string, meaning the word "the" is optional.
- **Imperative**: must | is to
- **Comparison**: equal to | less than | greater than
- **Value**: 1, 2, 3, . . .
- **Interval**: in the (interval | range) of
- **Range**: [**Value**] to [**Value**]
  - Note that intervals are assumed to be closed, hence "12 to 14" includes the values 12, 13, and 14, for example.

What Is To Be Done

This program should run from a **main file**, either `part2.pl` or `part2.oz` depending on your chosen language. As provided, both main files only read in sentences from stdin that are either **definitions** or **constraints**, and print out a list tokens for each sentence (which are produced by **read_line**). You should edit the **main file** to do the following:

1. Use a Natural Language Processing grammar, such as a **Definitie Clause Grammar**, to be able to take the lists of tokens produced from **read_line** and bind the key information (ex. ID, C (core count), Attribute, Constraint, etc.) to variables
2. store the variables with key information (noted above) so they can be used in a constraint calculation.
3. perform a constraint calculation based on the statements read in once reading is complete
4. output satisfactory results of the constraint computation to stdout (see the example below for sample printout)

Example Definitions, Constraints, and Results

Notice: In both languages, all atoms are made lowercase in the tokenization step (In Prolog: read_line, In Oz: Helper.tokenize) before they reach any of processing you do with your NLP grammer, so do not worry about dealing with upper case letters that are present in the spec files.

`procSpec1.txt`

```
Processor type a has 16 cores, uses 4 square centimeters, and costs 1000 dollars.
Processor type b has 8 cores, uses 3 square centimeters, and costs 100 dollars.
Processor type c has 4 cores, uses 2 square centimeters, and costs 10 dollars.
Processor type d has 2 cores, uses 1 square centimeters, and costs 1 dollars.
The core count must be greater than 49.
Area must be less than 17.
The cost is to be less than 2400.
```

(Note the extra newline at the end of the file shown above; This is good practice to make it easier to read the last line.)

Results:

```
a = 2, b = 1, c = 0, d = 5;
a = 2, b = 1, c = 1, d = 3;
a = 2, b = 1, c = 2, d = 1;
a = 2, b = 2, c = 0, d = 1;
a = 2, b = 2, c = 0, d = 2;
a = 2, b = 2, c = 1, d = 0;
```

`procSpec2.txt`

```
Core count is to be in the interval of 48 to 52.
The cost must be greater than 2149.
Processor type d has 2 cores, uses 1 square centimeters, and costs 1 dollars.
Processor type c has 4 cores, uses 2 square centimeters, and costs 10 dollars.
Processor type b has 8 cores, uses 3 square centimeters, and costs 100 dollars.
Processor type a has 16 cores, uses 4 square centimeters, and costs 1000 dollars.
The area must be in the range of 14 to 18.
```

(Note that the definition sentences occur in between the constraint sentences, but this does not affect the output. Also note that the order of the definition sentences being reversed causes the output below to be reversed, printing `d` to `a` instead of `a` to `d` on each line.)

Results

```
d = 0, c = 0, b = 2, a = 2;
d = 0, c = 1, b = 0, a = 3;
d = 0, c = 1, b = 2, a = 2;
d = 1, c = 0, b = 2, a = 2;
d = 2, c = 0, b = 0, a = 3;
d = 2, c = 0, b = 2, a = 2;
```

How to Run in Prolog:

A file, **`part2.pl`**, is provided that will read in a file with **definitions** and **constraints** until encountering the end of the file (EOF). Note that **`part2.pl`** imports the file **`read_line.pl`**, which is also provided in the PA3 web directory. As it stands, all it does is print out the parsed data from the statements. See the **What Is To Be Done** section above on what you need to add to **`part2.pl`**.

To test this file, run the following:

```
swipl -f part2.pl -g main < procSpec1.txt
```

Notes for Oz Programmers:

Compiling and Running

All coding will be done in **`part2.oz`**. **`part2Helper.oz`** contains a helper (`Helper.tokenize`) that will read and tokenize the spec file for you.

```
ozc -c part2Helper.oz #Only need to do this once
ozc -c part2.oz
ozengine part2.ozf procSpec1.txt #Run one or both of these to test
ozengine part2.ozf procSpec2.txt
```

A Warning About Attempting to Circumvent The Natural Language Processing Grammar in Oz

In Oz, it is possible to do much more advanced list manipulation out of the box than in prolog. Hence, many people may be inclined to make an attempt at extracting the relevant information from the token list by looking at the positions of tokens in the line or looking for keywords that surround a token and extracting from there. While it is certainly possible to do this, we have explicitly designed the grammar so this approach will be very difficult. Thus, you are strongly encouraged to make use of a Natural Language Processing Grammar and to avoid naive extraction approaches.

What To Submit & Grading

For **Part 1**, please submit either **`relations.pl`** or **`relations.oz`**, depending on the language you choose. These files will be tested with a main file similar to **`part1_template.pl`** or **`part1Tester.oz`**, but with queries not revealed publicly. This should be no issue, as properly implemented rules should be rather flexible.

For **Part 2**, please submit either your completed version of **`part2.pl`** or **`part2.oz`**, depending on the language you choose. The **`read_line`** (**`Helper.tokenize`**) files in both languages are provided and do not need to be edited.

You should also submit a **`readme.txt`** file with your name (or names if you have a partner). If you have any comments; reports of bugs, self assessments, anything, put it in your **`readme`**.

For all submitted files, please zip them up into a **`.zip`** file, with its filename matching the pattern:

`[your_rcs_id]_[language_identifier].zip`

For example, **`jsmith_pl.zip`** and **`jdoe_oz.zip`** are acceptable. For partners, please specify both of your RCS ID's, like so: **`jdoe_jsmith_pl.zip`**.

The **grade components** are as follows:

Part	Percentage
1: A Family Tree	40%
2: Microprocessor Constraint Language	(see sub-parts below)
2a: NLP Parser	25%
2b: Constraint Calculation	25%
Code Clarity & Comments	10%

Warning About Posting Solutions Publicly & Inter-Team Code Sharing

Do not post any code you submit for this assignment onto any public website or network share. Also, your code from your team (which can be at most two people) should be wholly yours; Do not look at or use code made by other students and/or teams as your own. If violation of either of these rules is detected, your grade will be heavily affected.

Links & Documentation for (SWI) Prolog

- Quickstart documentation:
  - **`https://www.swi-prolog.org/pldoc/man?section=quickstart`**
- Installers for version 8.4 (Windows/MAC):
  - **`https://www.swi-prolog.org/download/stable`** (for Linux, check your package manager for `swi-prolog`)
- Definate Clause Grammar documentation:
  - **`https://www.swi-prolog.org/pldoc/man?section=DCG`**

- Constraint Logic Programming over Finite Domains documentation:
  - `https://www.swi-prolog.org/pldoc/man?section=clpfd`