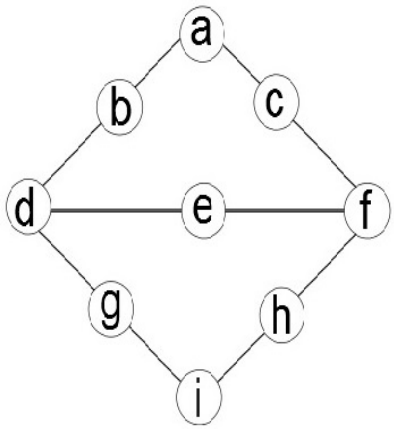
CSCI.4430/6430 Programming Languages Fall 2015
Programming Assignment #3

*This assignment is to be done either **individually** or **in pairs**. **Do not show your code to any other group** and **do not look at any other group's code**. Do not put your code in a public directory or otherwise make it public. However, you may get help from the TAs or the instructor. You are encouraged to use the LMS Discussions page to post problems so that other students can also answer/see the answers.*

Constraint Solving and Natural Language Processing

Part 1. Puzzle Solving.



In the image presented above, there are nine circles. Each circle must be assigned a number between 1-9 exactly once. Find the satisfying assignment of numbers such that the sum of the numbers across each side and the horizontal is 17. For example, the sum of the numbers filled in circles a, b, and d must add up to 17. Likewise, the sum of the numbers filled in circles d, e, and f must add up to 17.

Notes for Prolog Programmers:

Your final solution should have a one argument predicate called `puzzle`. The predicate's argument should represent the number assignments for each slot in the puzzle. The ordering of the list should be `[A,B,C,D,E,F,G,H,I]`, where the letters represent the slots in the above image.

By entering the following Prolog query, your program should return a valid solution to the puzzle:

```
?- puzzle(Solution).
Solution=[...]
```

Hitting semicolon should return alternate solutions.

Your code should be placed in a file called `puzzle.pl`.

Notes for Oz Programmers:

You should remove any other Oz installations and use Oz 1.4.0 downloadable from http://sourceforge.net/projects/mozart-oz/files/v1/ and Emacs downloadable from http://gnu.mirror.vexxhost.com/emacs/, to enable the Oz programming interface to find the proper emacs running command such as runemacs.exe, you need to create an environment variable OZEMACS and set its value to be something like F:\Programs \emacs-21.1\bin\runemacs.exe

Your final solution should have a one argument procedure called `Puzzle`. When executing the following Oz code, your program should return a list of all possible solutions, each of the form `[A B C D E F G H I]`, where the letters represent the slots in the above image.

```
{Browse {SearchAll Puzzle}}
```

See CTM Chapter 12.2 for constraint programming patterns.

Your code should be placed in a file called `puzzle.oz`.

Part 2. Sentence Parsing.

Here, you will create a grammar the accepts a set of sentences. You will then use this grammar to output a parse tree for a given input.

Below is the set of sentences that your grammar must be able to accept.

- the train flew.
- a bike flew.
- a flight left.
- a train arrived.
- every train left.
- the bike stayed.
- a train stayed.
- the flight flew.
- a flight stayed.
- the train arrived.
- every train flew.
- a person arrived.
- every bike stayed.
- a flight flew.
- a bike left.
- the person stayed.
- the person left.
- every bike left.
- a bike stayed.
- a train left.
- every bike flew.
- every person arrived.
- the flight arrived.
- a person left.
- the person flew.
- the bike left.
- every train arrived.
- a train flew.
- a flight arrived.
- every flight stayed.
- every person left.
- the bike arrived.
- every bike arrived.
- the person arrived.
- every flight arrived.
- the train left.
- the bike flew.
- the flight left.
- a person stayed.
- a person flew.
- every flight left.
- a bike arrived.
- every person flew.
- every train stayed.
- the flight stayed.
- every person stayed.
- the train stayed.
- every flight flew.

Notes for Prolog Programmers:

You will create a predicate called `loop` to allow the user to continuously input sentences and receive the resulting parse tree.

Example:

```
?- loop.
|: a train flew.
```

```
s(np(det(a),nom(noun(train))),vp(verb(flew)))
|: every train arrived.
s(np(det(every),nom(noun(train))),vp(verb(arrived)))
```

To assist you with this part, we have provided you with the `read_line` predicate, which can be found in read_line.pl.

To provide further assistance with Part 2 and Part 3, we have provided sample code, which can be found in parse_example.pl. This example program will show you how to load code from a separate file and how to use DCG syntax to accept a grammar and generate a simple parse tree.

The parse tree generated by your program must use the following predicates:

- s/2 - represents overall sentence; expects np and vp
- np/2 - represents noun phrase; expects det and nom
- vp/1 - represents verb phrase; expects verb
- det/1 - represents determiner; expects a word
- nom/1 - represents nominal; expects noun
- noun/1 - expects a word
- verb/1 - expects a word

Note: the number next to the predicate represents the number of arguments it expects.

Your code should be placed in a file called `parse.pl`.

Notes for Oz Programmers:

You should remove any other Oz installations and use Oz 1.4.0 downloadable from http://sourceforge.net/projects/mozart-oz/files/v1/ and Emacs downloadable from http://gnu.mirror.vexxhost.com/emacs/, to enable the Oz programming interface to find the proper emacs running command such as runemacs.exe, you need to create an environment variable OZEMACS and set its value to be something like F:\Programs \emacs-21.1\bin\runemacs.exe

You will write a function called `ParseSentence` whose only argument is a list representing a sentence. The function should return a parse tree represented with a tuple.

Example:

```
{Browse {ParseSentence [a train flew]}} % should display s(np(det(a) nom(noun(train))) vp(verb(flew)))
{Browse {ParseSentence [every train arrived]}} % should display s(np(det(every) nom(noun(train))) vp(verb(arrived)))
```

See CTM Chapter 9.4 for Oz program patterns for natural language parsing.

The parse tree generated by your program must use the following tuples:

- s/2 - represents overall sentence; has parts np and vp
- np/2 - represents noun phrase; has parts det and nom
- vp/1 - represents verb phrase; has a verb part
- det/1 - represents determiner; has an atom
- nom/1 - represents nominal; has a noun part
- noun/1 - has an atom
- verb/1 - has an atom

Note: the number next to the tuple represents the number of parts it has.

The definition for `Solve` are given in lib.oz. You may copy the declarations in this file to the beginning of your code.

Your code should be placed in a file called `parse.oz`.

Part 3. Simple Database.

For this part, you will be extending your grammar from Part 2 to simulate a simple database. Your program will receive commands to update the database and respond to queries with 'yes' or 'no'.

Commands to the database must take the following forms:

```
the <noun> _ <verb>.
the <noun> _ did not <pres_verb>.
```

<noun> is one of the nouns from Part 2.

<verb> is one of the verbs from Part 2.

<pres_verb> is either leave, fly, arrive, or stay.

_ is any word.

Example:

```
the person jane left.
the person jack did not leave.
```

The effect of entering a command should be the creation of a fact in your database.

Queries to the database must take the following forms:

```
did a <noun> <pres_verb>?
did every <noun> <pres_verb>?
```

Example:

```
did a train arrive?
did every bike stay?
```

A query must output either a 'yes' or a 'no' depending on whether the statement is true or false. Semantically, 'a' represents the existential quantifier and 'every' represents the universal quantifier. For queries using 'every', output 'yes' if there are no facts to query.

Example: After the following two commands:

```
the person john left.
the person jane left.
```

The query:

```
did a person leave?
```

should return 'yes', the query:

```
did every person leave?
```

should return 'yes', and after the command:

```
the person jake did not leave.
```

the query:

```
did a person leave?
```

should return 'yes', the query:

```
did every person leave?
```

should return 'no', the query:

```
did a train arrive?
```

should return 'no', and the query:

```
did every train arrive?
```

should return 'yes'.

Note: For simplicity, you can assume that duplicate commands or contradictory commands will not be given by the user.

Notes for Prolog Programmers:

You will write a `loop` predicate that will continuously prompt the user to enter a command or query. The program should output a 'yes' or 'no' in response to entering a query.

Your code should be placed in a file called `db.pl`.

Notes for Oz Programmers:

You should remove any other Oz installations and use Oz 1.4.0 downloadable from http://sourceforge.net/projects/mozart-oz/files/v1/ and Emacs downloadable from http://gnu.mirror.vexxhost.com/emacs/, to enable the Oz programming interface to find the proper emacs running command such as runemacs.exe, you need to create an environment variable OZEMACS and set its value to be something like F:\Programs \emacs-21.1\bin\runemacs.exe

You will write a `Handle` procedure whose first argument is a list representing either a command or a query, and second argument the database object.

Example:

```
DB = {New RelationClass init} % Init the database
{Handle [the person john left] DB}
{Handle [the person jane left] DB}
{Handle [did a person leave] DB} % should display 'yes'
{Handle [did every person leave] DB} % should display 'yes'
{Handle [the person jake did 'not' leave] DB}
{Handle [did a person leave] DB} % should display 'yes'
{Handle [did every person leave] DB} % should display 'no'
{Handle [did a train arrive] DB} % should display 'no'
{Handle [did every train arrive] DB} % should display 'yes'
```

The definition for `RelationClass` and `Solve` are given in lib.oz. You may copy the declarations in this file to the beginning of your code.

Your code should be placed in a file called `db.oz`.

Due Date: Thursday, 12/03, 7:00PM

Grading: The assignment will be graded mostly on correctness, but code clarity / readability will also be a factor (comment, comment, comment!).

Submission Requirements: Please submit a ZIP file with your code, including a README file. In the README file, place the names of each group member (up to two). Your README file should also have a list of specific features / bugs in your solution. Your ZIP file should be named with your LMS user name(s) as the filename, either userid1.zip or userid1_userid2.zip. Only submit one assignment per pair via LMS.