CMPS 109 Winter 2017
Computer Science Department
Quarter Project

Simple **R**ule-Base **I**nference Engine (SRI)

Dr. Karim Sobh

Assigned: Thursday, February 6th, 2017 in Class

Goals

This is a programming project that is designed to let you utilize all the concepts and apply all the libraries, tools, and advanced programming techniques explained in class to build a middleware for the **S**imple **R**ule-Base **I**nference Engine (SRI), which is in the form of a virtual machine capable of inferencing simple logic. The project is composed of four phases, where each phase depends on the preceding ones. It is required also to conduct a UML Object Oriented Design and maintain it by performing constructive updates to it through out the development and the buildup of the middleware.

Problem Description

SRI is a virtual machine that is designed mainly to apply variable substitutions within simple logical rules, from preloaded or deducted facts. SRI operates mainly on two main components; a knowledge base (KB) and a rule base (RB). The knowledge base is represented with simple facts, while the rule base is a set of rules that are used by SRI to deduce new rules. The facts and the rules in this sense forms a program that is given to SRI to inference and execute queries against.

A SRI session is an interactive program executing to receive commands from the user. Each command should be represented in one line, and commands can be one of the following:

1. Fact.
2. Rule.
3. Transactional command.

An SRI session can be started with a preload commands stored in a file, or even loaded incrementally through transactional commands. Moreover, a running session can store all its facts and rules into a file to be loaded into another session; think of that as an export mechanism.

A Rule-Base engine such as prolog (AI programming language), is a very sophisticated piece of software that is based on backward chaining, and has a lot of powerful options that can process rules with a lot of logical operators and perform many advanced functionalities. SRI is kind of a minimized version of prolog, and we would like to keep it as simple as possible by providing some restrictions on rules to be processed by the rule-base engine. Consequently, you are not required to implement backward chaining for example, and you can achieve the same effect by simple implementation means utilizing the STL containers and STD functions. Nevertheless, you will be utilizing multi-threading to parallelize the execution of rules as much as possible, as well as deploying the POSIX sockets library to provide distributed processing for federated knowledge bases.

Details

As we have just explained, SRI manages two sets of data, simply facts and rules, which are represented in a horn clause like format. For example the following are a set of facts that SRI can understand:

```
Father(Roger,John)
Mother(Marry,John)
Father(Roger,Albert)
Mother(Marry,Albert)
Father(Allen,Margret)
Mother(Margret,Robert)
Mother(Margret,Bob)
```

Where Father and Mother are relationships that applies to their parameters enclosed within the fact brackets. This kind of statement entails truth about the fact presented, and thus is kind of knowledge that SRI needs to store and maintains to be able to use it when needed, and hence these facts are stored in the KB. Notice that within the facts brackets, all the parameters needs to be factual terms, basically they are not variables.

Now, we need to be able to inquire the KB through inferencing rules, but first how a rule would look like. Here is an example of a SRI rule:

```
Parent($X,$Y):- OR Father($X,$Y) Mother($X,$Y)
GrandFather($X,$Y):- AND Father($X,$Z) Parent($Z,$Y)
GrandMother($X,$Y):- AND Mother($X,$Z) Mother($Z,$Y)
GrandMother($X,$Y):- AND Mother($X,$Z) Father($Z,$Y)
```

A rule is identified by two parts separated by the token ":-". The first part of the rule is used to match a query. The second part, which is the rule target, is basically all the unifications that need to be performed and converted to all possible facts. The rule target may include references to other rules, which has to be inferenced recursively. The right hand side of the rule should start with a logical operator that will apply on all the predicates of the rule target.

As we can see, to be able to differentiate between facts parameters and rule parameters, all rule parameters are required to be uppercase and start with the '$' character. The fact and the rule relationship names should be composed of alphabet characters: A-Z and a-z, without any white spaces.

A query is an attempt to inference a rule of a fact. For instance, using the above KB and RB, here are some query examples and their corresponding substitutions:

Father($X,$Y)
X: Roger, Y:John
X:Roger, Y:Albert
X:Allen, Y:Margret

This will traverse all KB and RB looking for facts or rules for the Father relationship. It should fetch anything that match from the KB and look for all rules whose left hand side has the Father relation and inference all the right hand side predicates of the rule the same way. In the above example we can only find facts in the KB about the Father relationship, but here is a more elaborative example that utilizes the RB as well.

Parent($A,$B)
A:Roger, B:John
A:Roger, B:Albert
A:Allen, B:Margret
A:Marry,B:John
A:Marry,B:Albert
A:Margret,Robert
A:Margret,Bob

This query should find the rule Parent($X,$Y) in the RB, it will parse its right hand side into individual predicates and identify the logical operator, which in our case is the OR operator. For the OR operator, each predicate will be executed separately and the results will be concatenated, and all duplicates will be removed if any.

Now, as an example of the AND pipelining operator, the GrandFather rule emphasize a relation between the second parameter of the first predicate and the first parameter of the second predicate of the rule target; $Z. So a GrandFather query would execute as follows:

GrandFather($A,$B)
First part:
Father ($A,$Z)
A:Roger, Z:John
A:Roger, Z:Albert
A:Allen, Z:Margret

Second Part: we then substitute with all the Z in the Parent rule.
Parent($Z,$B)
Parent(John,$B) → Empty
Parent(Albert,$B) → Empty
Parent(Margret,$B) → B:Robert
B: Bob

So the final result will be:
A:Margret, B: Robert
A:Margret, B: Bob

Also you can do filtering on queries, for example:
GrandFather($A,Robert)
A:Margret

Finally if the query matches a number of facts and rules, all of them need to be executed and the duplicate matchings need to be removed from the results set.

To be able to manage the KB and the RB in a more flexible way we introduce a set of transactional commands

Name	**Description**	**Attributes**
LOAD	Load facts and rules, into the KB and the RB respectively from a file.	A full path of a .sri file
DUMP	Save the facts and rules saved in the runtime KB and RB into a file in a format that can be loaded later.	A full path of a .sri file
FACT	Define a fact	An SRI fact
RULE	Define a rule	An SRI rule
INFERENCE	Issue a query	An SRI QUERY *A name for a fact relation to store the results under (optional)*
DROP	Drop facts or a rule	The name of the fact or the rule to be dropped from the KB or the RB.

Note: the # symbol is used to represent comments

And example of an SRI file (family_relation.sri)

FACT Father(Roger,John)
FACT Mother(Marry,John)
FACT Father(Roger,Albert)
FACT Mother(Marry,Albert)
FACT Father(Allen,Margret)
FACT Mother(Margret,Robert)
FACT Mother(Margret,Bob)

RULE Parent($X,$Y):- OR Father($X,$Y) Mother($X,$Y)
RULE GrandFather($X,$Y):- AND Father($X,$Z) Parent($Z,$Y)
RULE GrandMother($X,$Y):- AND Mother($X,$Z) Mother($Z,$Y)
RULE GrandMother($X,$Y):- AND Mother($X,$Z) Father($Z,$Y)

INFERENCE GrandFather($X,$Y) # prints the results on the terminal
INFERENCE GrandFather($X,$Y) GF # will declare the results under a fact with the name GF
DROP GF # delete all the facts created by the previous inference statement
DROP Parent # delete the Parent rule
DUMP updated_family_relation.sri # generate a sri file of the current KB and RB

SRI Command Line Terminal:
Part of SRI is a command line session component that enables the user to enter commands, load files, and dump scripts. Basically on starting SRI a terminal command line tool is started that can accept commands line by line and execute it if valid or generate an error otherwise. Also the SRI command line can accept upon invoking it a file to load upon start.

Project Phases:

Phase 1:

Conduct a UML design for the SRI middleware. Your UML design needs to utilize at least use case scenarios, class diagrams, and sequence diagrams. The design should take into consideration all the requirements stated in the project description and project details sections, as well as the requirements stated in the description of the subsequent phases. We do not expect that you provide a full complete design in this phase as the software design is a repetitive evolving process with a feedback mechanism from the incremental development cycles. We expect that an initial design document is submitted on the deadline of this phase, and an updated version of the design with each subsequent phase should be delivered. Nevertheless, it is recommended to study all the phases requirements and try to incorporate their requirements as much as possible.

Deliverables: a PDF design document with all the diagrams: use cases, class diagrams, and sequence diagrams, and full documentation of all assumptions made.

Phase 2:

The implementation of a SRI single-threaded virtual machine that satisfies all the requirements defined in the project description and the project details.

Deliverables: all the source code, and an updated version of the design document; you need to submit a design document even if not updated.

Phase 3:

In this phase you will modify the SRI single-threaded virtual machine, submitted in phase 2, to support multi-threading. Basically, you will providing parallelization of the OR logical operator, and pipelining of the AND operator.

Deliverables: the updated version of all the source code, and an updated version of the design document; you need to submit a design document even if not updated.

Phase 4:

In this phase you need to decouple your SRI virtual machine to realize a client/server virtual machine. The session command line tool will be the client component and the rest of the SRI virtual machine will be on the server side. Using POSIX sockets, the client reads commands from the user, sends it to the server, receives the results and display it on the terminal. The main thing here is that the server component can concurrently serve more that one client, and the client can change its server by providing its IP address. Based on the requirements of this phase the server component must be multi-threaded.

Deliverables: the updated version of all the source code, and an updated version of the design document; you need to submit a design document even if not updated.

What to submit

1. UML Design document in PDF format including all UML diagrams and design details.
2. All your code.
3. Your code should be split among header files (.h) and source files (.cpp), and all your implementation need to be in the source files. You need to have a very good reason for including implementation in header files, and an explanation of such motives need to be included in your report.
4. Very detailed in-code documentation.
5. Make files.
6. A report describing how to build the software using g++, including all design decisions taken and their alternatives, and explaining anything unusual to the teaching assistant. The report should be in PDF format.
7. Do not submit any object, or executable files. Any file uploaded that can be reproduced will result in grade deductions.
8. **VERY IMPORTANT: YOUR CODE NEED TO RUN ON UNIX, PREFERABLY LINUX.**

Important Assumptions:

Some specifications that are built by default in the compiler's builds should not be assumed for granted. Please read the documentation of QNX as an example of a real-time operating system that withdraws some of the features granted by some builds of the compiler for better performance. .

http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.ide.userguide%2Ftopic%2Fmemory_Illegal_dealloc_.html

This is just an example, and my point is to urge you to try to be as conservative as possible with practices you usually do and are guarded when the compiler default build options are being enabled. You are developing code that you do not know who is going to be using it, and where is it going to be deployed.

How to submit:

After submitting your first 2 assignments, your git repo should be configured and ready to use. Include everything for phase1 under hw3, phase 2 under hw4, phase 3 under hw5, phase 4 under hw6. Keep on updating your repository as many times as needed. When you are confident of your submission, submit the commit Id of the desired version to the ecommons.

This is a group project and you should elect a group captain among your group members, and all the submissions will be performed through his/her git account. IMPORTANT: EVERY GROUP SHOULD SUBMIT THROUGH THE GROUP CAPTAIN ONLY, IF THERE ARE STUDENTS WHO ARE WORKING INDIVIDUALLY WE THEN CONSIDER THEM THE GROUP CAPTIN.

Deadline:

1. Phase 1: Wednesday 15th February, 2017.
2. Phase 2: Monday 27th February, 2017.
3. Phase 3: Monday 6th March, 2017.
4. Phase 4: Monday 13th March, 2017.

Grading

This assignment is worth 30% of the overall course grade. The assignment will be graded on a 100% grade scale, and then will be scaled down to the 30% its worth. The grading of the assignment will be broken down according to the following criteria:

1. Phase 1:	**10 %**
2. Phase 2:	**50 %**
a. Test cases	20 %
b. Code Quality	20%
c. Inline Documentation	5%
d. Updated Design Document and Final report	5%
3. Phase 3:	**20 %**
a. Test cases	8 %
b. Code Quality	8%
c. Inline Documentation	2%
d. Updated Design Document and Final report	2%
4. Phase 4:	**20 %**
a. Test cases	8 %
b. Code Quality	8%
c. Inline Documentation	2%
d. Updated Design Document and Final report	2%

The project will be evaluated based on the correct execution of the SRI virtual machine, the code correctness and neatness, design decisions aspects, and how much you have utilized the concepts of abstraction, encapsulation, modularity, and hierarchy. It is very important to highlight that although documentation is worth a small % in each phase, yet missing or unclear documentation might result in more grade deduction if the grader cannot verify the correctness of your code and logic, and the provided documentation is not sufficient to clarify such matters.

Delays

You have up to 2 calendar days of delay in each phase, after which the corresponding phase will not be accepted and your grade in that case will be ZERO. For every day (of the 2 allowed days per phase), a penalty of 5% will be deducted from the total grade (The 100% scale).