Project 4

Due December 11, 2020 at 9:00 PM

Overview

This project specification is subject to change at any time for clarification. For this project you will be writing Datalog queries and will be developing a non-recursive Datalog interpreter. The EBNF rules that will be used for the dialect of Datalog are in the following section. For this project Datalog queries will be constructed of a set of rules that will each have a rule head. Each rule head specifies the rule name and a set of variables. Each rule that does not have a rule body is considered a fact (also known as external rule); these facts will be backed by CSV data files that have the tuples that specify when the rule is true. Any other tuples do not satisfy the fact rules. Rules that have a rule body are to be calculated and attempt to find the values that will satisfy all of the subgoals. If multiple rules exist with the same name, then rule invocations of that name are considered to be a union of the multiple rules. The goal of the Datalog interpreter is to find all tuples that satisfy all of the rules specified, with the base values coming from the fact rules.

Non-Recursive Datalog Programming Language

The following EBNF describes the non-recursive Datalog language. The Non-terminals on the right are identified by italics. Literal values are specified in bold. The operators not in bold or italics describe the options of the EBNF. These operators are {} for repetition, [] for optional, () for grouping, and | for or.

EBNF Rules:

```
Query := Rule { Rule }
Rule := [ RuleHead [ := RuleBody ] ] NewLine
RuleHead := RuleName ( HeadVariableList )
RuleBody := SubGoal { AND SubGoal }
RuleName := Identifier
HeadVariableList := Identifier { , Identifier }
SubGoal := RuleInvocation | NegatedRuleInvocation | EqualityRelation
RuleInvocation := RuleName ( BodyVariableList )
NegatedRuleInvocation := NOT RuleInvocation
EqualityRelation := InequalityRelation { EQOperator InequalityRelation }
BodyVariableList := InvocationVariable { , InvocationVariable }
InequalityRelation := Term { IEQOperator Term }
EQOperator := != | =
InvocationVariable := Identifier | EmptyIdentifier
Term := SimpleTerm { AddOperator SimpleTerm }
IEQOperator := < | > | <= | >=
SimpleTerm := UnaryExpression { MultOperator UnaryExpression }
AddOperator := + | -
UnaryExpression := ( UnaryOperator UnaryExpression ) | PrimaryExpression
MultOperator := * | / | %
UnaryOperator := ! | - | +
PrimaryExpression:= ( EqualityRelation ) | Constant | Identifier
Constant := IntConstant | FloatConstant | StringConstant
```

Token Rules:

```
Identifier := Alpha { ( Digit | Alpha ) }
EmptyIdentifier := _
IntConstant := Digit { Digit }
FloatConstant := Digit { Digit } [ . Digit { Digit } ]
StringConstant := " { ( CharacterLiteral | EscapedCharacter ) } "
Comment := # { ( WhiteSpace | Printable ) } NewLine
Digit := 0 – 9
Alpha := A – Z | a – z
WhiteSpace := Tab | CarriageReturn | \ NewLine | Space
Printable := Space – ~
CharacterLiteral := Space – ! | # – [ | ] – ~
EscapedCharacter := \b | \n | \r | \t | \\ | \' | \"
```

Semantics:

There are several requirements for the non-recursive Datalog queries that cannot be expressed in the EBNF. The following must be true of all valid queries:

- All subgoals must be defined prior to their use
- All repeated rule names must be defined contiguously
- All repeated rule names must have the same number of variables
- Any variable in the rule head or rule body must appear in a non-negated rule invocation
- A rule invocation may not appear in its own rule definition (i.e. recursive definition)
- Fact rules may only appear once
- Every rule invocation must provide the same number of variables as its definition

CSV Data Files

The facts can be stored in a CSV files. The name of the file without .csv extension will be the name of the facts loaded into the environment. Each column is a different attribute, with the header row containing the attribute name. Four data types are allowed in the CSV files: string, float, integer, and boolean. Rows two and on, each have one tuple in them.

The following is an example of fact R that would be in the file R.csv:

"a"	"b"	"c"	"d"
3	"Hello"	3.4	true
4	"World"	1.1	false
6	"Goodbye"	8.8	false
7	"None"	9.3	true

Datalog Examples

The following are a few Datalog query examples using the grammar described in the previous sections using the R fact described.

Simple fact query:

```
R(a,b,c,d)
```

This will result in the following output:

```
a b c d
3 Hello 3.4 true
4 World 1.1 false
6 Goodbye 8.8 false
7 None 9.3 true
```

Simple filtering for `a` being greater than 5:

```
R(a,b,c,d)
S(x) := R(x,_,_,_) AND x > 5
```

This will result in the following output:

```
x
6
7
```

Simple filtering for `c` being greater than 8.0 or less than 3.0:

```
R(a,b,c,d)
S(x) := R(_,_,x,_) AND x > 8.0
S(x) := R(_,_,x,_) AND x < 3.0
```

This will result in the following output:

```
x
1.1
8.8
9.3
```

Find all `b` associated with `c` not being the smallest in `R`:

```
R(a,b,c,d)
S(x) := R(_,x,c1,_) AND R(_,_,c2,_) AND c1 > c2
```

This will result in the following output:

```
x
Hello
Goodbye
None
```

Datalog Queries

Write Datalog queries for the questions posed below. Name your query files `query_X.nrdl` where `X` is the number of the question below. You may assume the following facts will be available:

Product(maker, model, year)
Car(model, city, highway, style, passengers, trunk, msrp)
Pickup(model, city, highway, passengers, cargo, towing, msrp)
EV(model, range, battery, passengers, msrp)

1) What Car models have a highway less than 35miles? The final rule head should be Answer(model).
2) Find all of the Pickup models that have a cargo capacity of at least 75cu ft. and a highway fuel economy less than 25MPG. The final rule head should be Answer(model).
3) Find all automakers that sell at least one vehicle that msrp less than $27,000 and at least one vehicle greater than $55,000. The final rule head should be Answer(maker).
4) Find the passenger capacities that exist for three or more vehicles. That is three or more different vehicles should have that passenger capacity. The final rule head should be Answer(passengers).
5) Find the automaker(s) of the highest combined fuel economy (55% city, 45% highway) of conventional vehicles (cars and pickups). The final rule head should be Answer(maker).
6) Find the vehicle model with the highest miles per gallon gasoline equivalent (MPGGE). For this problem assume combined fuel economy formula from above, and that a gallon of gasoline is equivalent to 33.1kWh. The final rule head should be Answer(model).
7) Find automaker(s) that sell a pickup with a highway fuel economy lower than all the cars it sells. The final rule head should be Answer(maker).

Datalog Interpreter

The Datalog interpreter will be constructed from several classes. A given NRDatalog class provides a class that can parse command line arguments and instantiate and call the appropriate classes. The following classes should be created with at least the specified interface.

```
// Class that parses the non-recursive Datalog language
public class NRDatalogParser{
  // Constructor for the parser
  public NRDatalogParser(PeekableCharacterStream stream);
  // Parses a query and returns true if valid query
  public boolean parseQuery();
  // Prints out the error if one has occurred
  public void printError(PrintStream ostream);
}
```

```
// Class that will construct the parse tree from
public class NRDatalogParseTree extends NRDatalogParser{
  // Constructor for the parse tree
  public NRDatalogParseTree(PeekableCharacterStream stream);
  // Parses a query and returns true if valid query
  public boolean parseQuery();
  // Prints out the error if one has occurred
  public void printError(PrintStream ostream);
  // Outputs a tree of the parsed query
  public void outputParseTree(PrintStream ostream);
}

// Class that will construct and execution tree from the parse
// tree. Executes the query and displays the results.
public class NRDatalogExecutionTree extends NRDatalogParseTree{
  // Constructor for the execution tree
  public NRDatalogExecutionTree(PeekableCharacterStream stream);
  // Parses a query and returns true if valid query
  public boolean parseQuery();
  // Prints out the error if one has occurred
  public void printError(PrintStream ostream);
  // Outputs a tree of how execution would occur for queries
  public void outputExecutionTree(PrintStream ostream);
  // Sets the verbosity setting for execution
  public void setVerbose(boolean verb);
  // Sets the number of threads during execution
  public void setThreadCount(int threadcount);
  // Sets the path to the data files
  public void setDataPath(String datapath);
  // Executes the parsed query
  public boolean executeQuery();
}
```

Suggested Approach

1. Work on writing the Datalog queries first. An example program is available to execute the queries. This will provide you the basic understanding of how the operations should work from the query writing perspective.
2. Update your Scanner to accommodate the new set of operators. Make sure to change the newline to an operator, if a whitespace newline is desired, it must be preceded with a backslash. There are only two keywords **AND** and **NOT**.
3. Write the NRDatalogParser class to recognize the non-recursive Datalog language. Similar to project 2 you will want one method per rule. Creating an enum class that represents each rule and creating an isFirst method may aid in the development of the rules.

4. Write the NRDatalogParseTree class to construct the parsed query as a tree. You will likely want to create some type of node class within the NRDatalogParseTree that holds the rule type, associated token (if any), parent, and children. You may find using a stack data member will be helpful in the recursive decent parsing. The top of the stack can hold the current parent node.
5. Write a class to hold the data sets. These should be able to hold a set of list of objects. Each list of objects can act as data tuple. It should also have a list of column names. This class should be able to be constructed from a PeekableCharacterStream (so to load using CSV), or from list of column names and the set of list of objects. The data set class should support:
    a. Appending tuples at least privately (this will help constructing new data sets)
    b. Filtering (or selecting) tuples given an object that has function returning if the tuple should be in the new data set or not (this will essentially remove rows from the data set)
    c. Projecting specified columns to a new data set (this is similar to cutting off columns that are not desired, may want to consider supporting renaming of the columns with this as well)
    d. Cartesian product with another data set to create a new data set
    e. Natural join with another data set to create new data set
    f. Filter (or selecting) tuples given tuples do not appear in another data set (the tuple should be kept if the combination of columns specified do not appear in the other data set.
    g. Union with another data set to create a new data set (it is assumed that both have the same column names)
    h. Reordering columns to create a new data set (may be helpful when projecting down temporary work into final rule)
6. Write the NRDatlogExecutionTree class to construct the tree to that it can easily be executed. You will likely want to create some type of node class within the NRDatlogExecutionTree that holds the rule type, associated token (if any), parent, children, and possibly a node index (may be helpful for sorting step).
    a. Translate the parse tree into an execution tree. The difference is that infix operators will be migrated up to be the parent of the operands instead of being children of the larger rule. You will likely want to write a function that recursively visits the nodes of the parse tree and returns a converted execution tree node equivalent. Tokens like open/close paren, assignment, and commas should be omitted. The body sub goals should also be sorted for simplicity of execution. Rule invocations should be first, followed by negated rule invocations and finally equality relations.
    b. Write a method to validate the semantic rules for the execution tree. This will make sure that execution should be able to proceed.
    c. Write a method to execute a rule without a rule body. This will essentially just load a data set and add it to the calculated rules.
    d. Write a method to execute a rule with a rule body. This will create a temporary calculated data set that will either be added to the calculated rules or unioned with the previously calculated rule of the same name.
        i. Create the ability to invoke rules by projecting them based upon the invocation. This projected invocation will then be either naturally joined or

cartesian product with the temporary calculated data set depending if there are any overlap in column names.

ii. Create the ability to do a negated invocation, this is similar to invoking rules; however, this will potentially be removing tuples from the temporary data set instead of adding tuples.

iii. Create the ability to filter the tuples based upon the equality relations. A filter object should be created using the subgoal and should be able to calculate if a tuple should be kept or omitted.

Examples of valid and invalid nrdl files are in `/home/cjnitta/ecs140a/proj4` on the CSIF. Script that runs the NRDatalog class can be found there too. Runnig the script without options, or with the `--help` option will print the help. Your code will be tested on the CSIF and is expected to compile and run on the CSIF.

You **must** submit the source file(s), a Makefile, and README.txt file, in a tgz archive. Do a `make clean` prior to zipping up your files so the size will be smaller. You will want to be in the parent directory of the project directory when creating the tgz archive. You can tar gzip a directory with the command:

```
tar -zcvf archive-name.tgz directory-name
```

You should avoid using existing source code as a primer that is currently available on the Internet. You **must** specify in your readme file any sources of code that you have viewed to help you complete this project. You must also provide the URL any code sources in comments of your source code. All class projects will be submitted to MOSS to determine if students have excessively collaborated. Excessive collaboration, or failure to list external code sources will result in the matter being referred to Student Judicial Affairs.

Helpful Hints

- The use of `peekNextToken()` will be helpful in some instances where it isn't clear what the next rule should be parsed. The language is designed so that only a single look ahead is necessary.
- Creating an enum class for the rule names will be helpful in multiple levels and is highly encouraged.
- Use the working example to output the parse tree and the execution tree. Being able to match them will help prior to executing the queries.
- Using Integer, Float, String, and Boolean types will allow the data sets to hold lists of Objects. The instanceof operator will be helpful in detecting the type during calculations of evaluations.

Extra Credit

After successfully creating a Datalog interpreter convert the NRDatalogExecutionTree into a multithreaded version. The executeQuery method needs to use multiple threads to execute the

query in parallel on systems with multiple cores. Larger data sets with a query that requires significant time will be provided later.