Printer-Friendly Version (PDF)

The Datalog Parser

CS236 Project Two

Introduction to Parsing

A lexical analyzer groups characters in an input stream (S) into tokens. Parsing determines if an input stream of tokens is a member of a language (L) as defined by a grammar (G). (i.e. is S elementof L(G)?)

Project Description

Using your LexicalAnalyzer from Project 1, write a parser for the Datalog grammar defined below. Follow the grammar very closely! Be sure to skip comment tokens.

If the parse is successful (i.e. if S elementof L(G)), output to the terminal "Success!" followed by all the schemes, facts, rules, queries, and the domain values (i.e. all the strings) that appear in the facts, rules, and queries. Include the number of items in each list. Note that the domain is a set (no duplicates) of strings. It MUST be sorted alphabetically. The list items are indented by 2 spaces.

If the parse is unsuccessful (i.e. if S !elementof L(G)), output "Failure!" followed by the offending token (indented by 2 spaces) and the line number on which the offending token is found. If there happens to be more than one offending token, you only need to find the first one.

Requirements (Get the Design Right)

Success in this project (indeed, for most programming projects) depends largely on the choice of data structures. The Datalog grammar suggests that these data structures could consist of a class for each of the many non-terminals. For instance, the <Datalog Program> non-terminal and the <Predicate> non-terminal COULD have the corresponding class shown on the right.

```
class DatalogProgram{
    private SchemeList schemes;
    private FactList facts;
    private RuleList rules;
    private QueryList queries;
//Constructors …
//Access Methods …
//Update Methods …
};
```

Through the appropriate use of inheritance you can make the implementation simpler in this project and especially in the subsequent projects.

```
class Predicate{
   private Token predicateName;
   private ParameterList
                    parameters;
//Constructors …
//Access Methods …
//Update Methods …
};
```

Creating these classes is part of this lab. The TAs will check that you have created classes for at least the following parts of the language: DatalogProgram, SchemeList, Scheme, FactList, Fact, RuleList, Rule, QueryList, Query, Domain, Predicate, and Parameter. You may also have others, but you must have these.

You must have a toString method on all of these objects so that you can easily print them out. You may derive the list-type objects from ArrayList. You are not required to make your own list object. Your parsing code should return a DatalogProgram object, and then you should traverse this structure and use toString as needed to print the required output.

The TAs will also check to see that you have code components that logically correspond either to a recursive descent parser or to a table-driven parser. Make the correspondence obvious, for this is the way to make complex programs simple to understand. To integrate this project with Project 4, be sure to have a

way to get a SchemeList, FactList, RuleList, and QueryList out of your DatalogProgram. How you choose and use your data structures for this project effects the difficulty of Project 4 (and Project 5). Inheritance and Polymorphism are always useful. Each data structure implementation has its benefits, drawbacks and obstacles to overcome.

The Datalog Grammar

In the grammar, we use the following meta-symbols: "<" and ">" enclose non-terminal names, "->" means "is defined as," "|" means "or," and "ε" represents the empty string. Follow this grammar very closely!

<Datalog Program>	-> Schemes : <Scheme List> Facts : <Fact List> Rules : <Rule List> Queries : <Query List> <EOF>
<Scheme List>	-> <Scheme><Scheme List Tail>
<Scheme List Tail>	-> <Scheme List> | ε
<Scheme>	-> <Scheme Name> (<Attribute List>)
<Scheme Name>	-> <Identifier>
<Attribute List>	-> <Attribute> <Attribute List Tail>
<Attribute List Tail>	-> , <Attribute List> | ε
<Attribute>	-> <Identifier>
<Fact List>	-> <Fact> <Fact List> | ε
<Fact>	-> <Fact Name> (<Constant List>) .
<Fact Name>	-> <Identifier>
<Constant List>	-> <String> <Constant List Tail>
<Constant List Tail>	-> , <Constant List> | ε
<Rule List>	-> <Rule> <Rule List> | ε
<Rule>	-> <Simple Predicate> :- <Predicate List> .
<Simple Predicate>	-> <Predicate Name> (<Argument List>)
<Predicate Name>	-> <Identifier>
<Argument List>	-> <Argument> <Argument List Tail>
<Argument List Tail>	-> , <Argument List> | ε
<Argument>	-> <String> | <Identifier>
<Predicate List>	-> <Predicate> <Predicate List Tail>
<Predicate List Tail>	-> , <Predicate List> | ε
<Predicate>	-> <Predicate Name> (<Parameter List>)
<Parameter List>	-> <Parameter> <Parameter List Tail>
<Parameter List Tail>	-> , <Parameter List> | ε
<Parameter>	-> <Argument> | <Comparator> | <Expression>
<Comparator>	-> <Comparator Name> (<Argument>,<Argument>)
<Comparator Name>	-> < | <= | = | != | >= | >
<Expression>	-> (<Parameter> <Operator> <Parameter>)
<Operator>	-> + | *
<Query List>	-> <Query> <Query List Tail>
<Query List Tail>	-> <Query List> | ε
<Query>	-> <Simple Predicate> ?

Examples

These are not sufficient to completely test your program. Your program must have output formatted exactly like the example outputs below:

Example 20 Input: ex20.txt

```
Schemes:
  employee(N,I,A,J)
  WhoToBlame(N,J)

Facts:
  employee('Dilbert','51','10 Main','EE').
  employee('Dilbert','51','10 Main','Marketing').
  employee('Dogbert','52','10 Main','EE').
  employee('PHB','53','Hades','Pain Management').

Rules:
  WhoToBlame(N,J) :- employee(N,'51',A,J),expr((A+A)).
  WhoToBlame(N,I) :- NotEquals(!=('Dilbert','PHB')).

Queries:
  WhoToBlame('Dilbert',J)?
  WhoToBlame(N,'EE')?
```

Example 20 Output: out20.txt

```
Success!
Schemes(2):
  employee(N,I,A,J)
  WhoToBlame(N,J)
Facts(4):
  employee('Dilbert','51','10 Main','EE').
  employee('Dilbert','51','10 Main','Marketing').
  employee('Dogbert','52','10 Main','EE').
  employee('PHB','53','Hades','Pain Management').
Rules(2):
  WhoToBlame(N,J) :- employee(N,'51',A,J),expr((A+A)).
  WhoToBlame(N,I) :- NotEquals(!=('Dilbert','PHB')).
Queries(2):
  WhoToBlame('Dilbert',J)?
  WhoToBlame(N,'EE')?
Domain(11):
  '10 Main'
  '51'
  '52'
  '53'
  'Dilbert'
  'Dogbert'
  'EE'
  'Hades'
  'Marketing'
  'PHB'
  'Pain Management'
```

Example 21 Input: ex21.txt

```
Schemes:
  snap(S,N,A,P)

#comment

  HasSameAddress(X,Y)

Facts: #comment
  snap('12345','C. Brown','12 Apple','555-1234').
  snap('33333','Snoopy','12 Apple','555-1234').

Rules:
  HasSameAddress(X,Y) :- snap(A,X,B,C),snap(D,Y,B,E).

#comment

Queries:
  HasSameAddress('Snoopy',Who)?
```

Example 21 Output: out21.txt

```
Success!
Schemes(2):
  snap(S,N,A,P)
  HasSameAddress(X,Y)
Facts(2):
  snap('12345','C. Brown','12 Apple','555-1234').
  snap('33333','Snoopy','12 Apple','555-1234').
Rules(1):
  HasSameAddress(X,Y) :- snap(A,X,B,C),snap(D,Y,B,E).
Queries(1):
  HasSameAddress('Snoopy',Who)?
Domain(6):
  '12 Apple'
  '12345'
  '33333'
  '555-1234'
  'C. Brown'
  'Snoopy'
```

Example 22 Input: ex22.txt

```
Schemes:
  snap(S,N,A,P)
  NameHasID(N,S)

Facts:
  snap('12345','C. Brown','12 Apple','555-1234').
  snap('67890','L. Van Pelt','34 Pear','555-5678').

Rules:
  NameHasID(N,S) :- snap(S,N,A,P)?

Queries:
  NameHasID('Snoopy',Id)?
```

Example 22 Output: out22.txt

```
Failure!
  (Q_MARK,"?",10)
```

Passoff Requirements

Build your project using the cs236Bundle. Your source code goes in the src/project2 directory. All classes must be package project2 and import project1. You must implement the body method in Project2 class. Its only task is to construct a DatalogProgram and return an output string. This class receives a filename from the command line arguments. Other requirements for all cs236 projects can be found on the policies page.