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. In other words, you need to determine if $S \in L(G)$. The test is accomplished by creating an LL(1) parser for a simple variant of the Datalog language.

Project Description

Using your LexicalAnalyzer from Project 1, write a parser for the Datalog grammar defined below. You will want to add a few methods to your LexicalAnalyzer to make it easy for you to access the tokens and use the tokens in order to determine if the given input is an instance of a valid Datalog program.

If the parse is successful, $S \in L(G)$, return the string: 'Success!' followed by all the schemes, facts, rules, queries, and the domain values (i.e., all the strings that appear in the facts). Include the number of items in each list. Note that the domain is a sorted set (no duplicates) of strings.

If the parse is unsuccessful, $S \notin L(G)$, output 'Failure!' followed by the offending token, (i.e., its triple including its token-ID name, string value, and line number). Note that the parser stops after encountering the first offending token.

Requirements (Get the Design Right)

The following are required for successful pass-off (and will help you succeed in this project):

- You must implement an LL(1) parser (that is to say a deterministic top-down parser that chooses the rule to expand based on the current token). The parser implementation must also be **recursive** meaning it either uses the runtime stack to keep track of rules in the parse or it manages a stack or rules directly similar to the parse table. Either solution works though using the runtime stack is more direct and simple (i.e., create a method for every rule in the grammar).
- A class for each type of object. You must create classes for at least the following parts of the language: *datalogProgram*, *rule*, *predicate*, *parameter*, and possibly *expressions* or other aspects of the grammar. Additionally, you will need list objects to collect lists of schemes, facts, rules, or queries. Each class that contains objects to be output must have a *toString* method, and the output of the program must be formed by these *toString* methods (not by the parse methods).
- You need to collect a list of *domain* values. A domain value is any string constant that appears in a *fact*. The set of all unique string constants is the **domain**. Store the domain as a sorted list. As a reminder, only unique values should appear in the list (no duplicates).

Datalog Grammar

Production start with lower-case letters (nonterminals). Any production that starts with an upper-case letter, such as those at the end of

the grammar, represent tokens (terminals). These tokens are input from your scanner and are defined similarly in the Lexical Analyzer. Note that comments do not appear anywhere in the grammar because comments should be ignored. In terms of parsing, this means that you should be able to skip any number of comments showing up at any place in the program. The easiest way to do that is to simply have the lexical analysis ignore comments.

```
grammar Datalog;

/*****************************************************
 * Productions
 *****************************************************/
datalogProgram  ->      SCHEMES COLON scheme schemeList
                        FACTS COLON factList
                        RULES COLON ruleList
                        QUERIES COLON query queryList


scheme          ->      ID LEFT_PAREN ID idList RIGHT_PAREN


schemeList      ->      scheme schemeList | lambda


idList          ->      COMMA ID idList | lambda


fact            ->      ID LEFT_PAREN STRING stringList RIGHT_PAREN PERIOD


factList        ->      fact factList | lambda


rule            ->      headPredicate COLON_DASH predicate predicateList PERIOD


ruleList        ->      rule ruleList | lambda


headPredicate   ->      ID LEFT_PAREN ID idList RIGHT_PAREN


predicate       ->      ID LEFT_PAREN parameter parameterList RIGHT_PAREN


predicateList   ->      COMMA predicate predicateList | lambda


parameter       ->      STRING | ID | expression


parameterList   ->      COMMA parameter parameterList | lambda


expression      ->      LEFT_PAREN parameter operator parameter RIGHT_PAREN


operator        ->      ADD | MULTIPLY


query           ->      predicate Q_MARK
```

```
queryList       ->      query queryList | lambda


stringList      ->      COMMA STRING stringList | lambda


/*****************************************************
 * Token definitions from the lexer project
 * IGNORE: INCLUDED FOR COMPLETENESS FOR ANTLR
 *****************************************************/

COMMA   :       ','
        ;

PERIOD  :       '.'
        ;

Q_MARK  :       '?'
        ;

LEFT_PAREN      : '('
                ;

RIGHT_PAREN     : ')'
                ;

COLON           : ':'
                ;

COLON_DASH      : ':-'
                ;

MULTIPLY        : '*'
                ;

ADD             : '+'
                ;

SCHEMES         : 'Schemes'
                ;
FACTS           : 'Facts'
                ;
RULES           : 'Rules'
                ;
QUERIES         : 'Queries'
                ;

ID  :   ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
    ;

STRING
    :  '\'' ( ESC_SEQ | ~('\\'|'\'') )* '\''
    ;

COMMENT
    :   '#' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
    |   '#|' ( options {greedy=false;} : . )* '|#' {$channel=HIDDEN;}
    ;

WS  :   ( ' '
        | '\t'
        | '\r'
```

```
        | '\n'
        ) {$channel=HIDDEN;}
    ;

fragment
HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;

fragment
ESC_SEQ
    :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
    |   UNICODE_ESC
    |   OCTAL_ESC
    ;

fragment
OCTAL_ESC
    :   '\\' ('0'..'3') ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7')
    ;

fragment
UNICODE_ESC
    :   '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
    ;
```

Output Specifications

To format the output for pass-off, you must make it formatted exactly like the examples, including capitalization, punctuation, and whitespace. The output for the parser is show below in the examples.

- Indentation is two spaces
- The ':-' is separated by one space on either side
- Schemes, rules, facts, and queries should be listed in program order (i.e., the order in which they appear in the file)
- The domain should be sorted (e.g., standard *std::string* order)
- The domain includes only strings that appear in facts

Examples

You program must **exactly match** the output format in the examples. Pass-off involves comparing your output to expected output from a proper solution. Any mismatch in your output means the pass-off failed. Further, these examples are not comprehensive in testing your parser. They only are sufficient to define the output format.

Example 1

Input

```
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)?
```

Output

```
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 2

Input

```
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)?
```

Output

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

FAQ

What do I have to change in my Lexical Analyzer?

The lexical analyzer needs to give the parser tokens. The parser also needs to be able to determine the type of the token too (i.e., STRING, COMMA, PERIOD, etc.). The analyzer may need to change to include the ability to give the parser tokens for analysis.

Why do we need the Parameter, Predicate, Rule, and Datalog Program classes, and what should be in these classes?

Each lab projects builds on the previous project. These classes are intended to support the next project. To determine what needs to be in each of these classes, use the grammar as a guide. For example, a parameter is defined as *STRING* | *ID* | *expression*. A such, the parameter class should be able to look like a *STRING*, *ID*, or *expression*, and a method should exist to determine the type of parameter. As another example, A predicate is defined as *ID LEFT_PAREN parameter parameterList RIGHT_PAREN*. A class to represent a predicate needs to store the ID, and a list of parameters. Other classes are defined similarly: guided by the grammar.

As a further point of clarification, the datalog program has a list for schemes, facts, rules, and queries. It is tempting to build new classes for a scheme or query; however, the predicate class should suffice to represent these objects.

What strings need to be included in the domain?

The domain consists of all strings found in the facts. It does *not* include strings in the rules or queries.

How should I represent an expression?

Use polymorphism! An expression is a parameter. That parameter has an operator with a left parameter and a right parameter. It is possible to define different types of parameters using polymorphism: an ID parameter, a STRING parameter, and an expression parameter that has an operator with a left parameter and a right parameter. Such a structure makes expressions, even arbitrarily nested expression trivial.

How should I print expressions?

Print expression exactly how they look on the input: left parameter operator right parameter. Remember the left or the right parameter can be either or both expressions. No problem! It is possible to have a to_string method for each type of parameter using polymorphism. That way, printing is easy: just call to_string() on the parameter, and the to_string method knows how to do the rest (i.e., if it is an expression, it first calls left.to_string(), add the operator, and then calls right.to_string().

What is the easiest way to handle a parse error?

The most direct way to handle a parse error is with an exception [http://www.cplusplus.com/doc/tutorial/exceptions/]. A parse error is able to **throw** the offending token. And the caller of the parser is able to catch the same token and provide the required output.

Submission

Review the project standards for creating a zip archive.

Navigate to Learning Suite [http://learningsuite.byu.edu], select 'CS 236', and click on 'Assignments' on the course home page. Click on the link to the relevant project and at the bottom of the description click on 'View/Submit'. Use the resulting upload dialog to upload your zip archive.

Pass-off

Pass-off your project directly to a TA during normal TA hours. TAs help students on a first-come/first-serve basis and are under no obligation to stay later than designated hours so plan accordingly. Please review the syllabus for the pass-off requirements.

cs-236/datalog-parser.txt · Last modified: 2015/03/06 17:52 by egm