CMSC 124
Machine Problem - Documentation

Introduction

The program scans, parses, and evaluate a string from an input perl script. The lexemes and tokens are scanned in the scanner class and stores it in an arraylist. The contents of the arraylist are then parsed in the parser class and evaluate the logical expression. The parsing algorithm makes use of a recursive-descent parser. It follows the following grammar:

<Start> := <Sentence> ;
<Sentence> := <AtomicSentence> | <ComplexSentence>
<AtomicSentence> := “TRUE” | “FALSE” | “P” | “Q” | “R”
<ComplexSentence> := “(“ <Sentence> “)” | <Sentence> <Connective> <Sentence> | “NOT” <Sentence>
<Connective> := “AND” | “OR” | “IMPLIES” | “EQUIVALENT”

Below is a sample output of the program in the command prompt. To run the program, navigate through the directory of the java project and run the following commands:

- javac project.java
- java project <name of pl file>

```
====================

Next Lexeme: FALSE token: CONSTANT
false

====================

Next Lexeme: P token: IDENT
Next Lexeme: AND token: OP
Next Lexeme: Q token: IDENT

  P     Q    [P, AND, Q]
true  true    true
true  false    false
false  true    false
false  false    false

====================
```

```
================================

Next Lexeme: NOT token: NOT
Next Lexeme: P token: IDENT

  P  [NOT, P]
true   false
false   true

================================

Next Lexeme: P token: IDENT
Next Lexeme: AND token: OP
Next Lexeme: NOT token: NOT
Next Lexeme: ( token: L_PAREN
Next Lexeme: P token: IDENT
Next Lexeme: OR token: OP
Next Lexeme: Q token: IDENT
Next Lexeme: ) token: R_PAREN
Next Lexeme: OR token: OP
ERROR IN PARSING. <sentence> <connective> <sentence>
```

Pseudocode

This program will scan, parse and evaluate an input perl script.

1. Scan a character from the file. This method is in scanner.java
2. Call lex() function to evaluate the lexeme and scan another character until end of file. Store the lexemes and tokens in an array. The list of tokens are the following:
    - CONSTANT: “True” | “False”
    - OP: “AND” |“OR” | “IMPLIES” | “EQUIVALENT”
    - IDENT: “P” | “Q” | “R”
    - L_PAREN: “(“
    - R_PAREN: “)”
    - NOT: “NOT”
    - SEMICOLON: “;”
    - ERROR: error token for invalid inputs
3. Pass the array of lexemes and tokens to the parser class that would parse and evaluate the string.
4. Next() would return the current lexeme and token and peek() would return the next token after the current token. The functions sentence(), atomicSentence(), and complexSentence() is based on the non-terminals of the grammar rule and would return a Boolean value.

5. Parsing algorithm:

```
Start():
        result = Sentence() //evaluate current line
        while (ctr<2^identCount){
                //manipulate Boolean values of identifiers
                //ex: P = T T F F and Q = T F T F
                Start() //evaluate current line again with different values of identifiers
        }
        //proceed to next line
        Start()


Sentence():
        If peek() is CONSTANT or IDENT:
                value = atomicSentence()
        if peek() is L_PAREN or NOT:
                value = complexSentence()
        while peek() is an OP:
                next()
                nextValue = complexSentence()
                value = evaluate(value, OP, nextValue)
        if peek() is ERROR:
                print error and terminate the program.


        return value


atomicSentence():
        next();
        return bool of the atomic sentence


complexSentence():
        if peek() is CONSTANT OR IDENT:
                value = sentence()
        if peek() is L_PAREN:
                next()
                value = sentence()
                if no R_PAREN:
                        ERROR
                Else: next()
        If peek() is NOT:
                next()
                value = !sentence()
        return value()
```

Variables and labels

- **lexeme** – string unit based on the grammar rule like “TRUE”, “P”, “AND”, “IMPLIES”, etc.
- **token** – categorizes the lexemes into CONSTANT, IDENT, OP, NOT, SEMICOLON, PARENTHESIS
- **lexemes** – arraylist of lexemes and tokens
- **identCount** – count of identifiers.
- **P, Q, R** – boolean values for identifiers
- **sentenceValue** – boolean value returned by calling sentence()
- **nextVal** – boolean value to be evaluated with sentenceValue
- **value** – boolean value returned by other functions

Error Handling

- Syntax errors were already checked while scanning the file for lexemes and tokens. If error is found, a message would appear and terminate the program.

- Conditions were specified to check if the string follows the grammar rule. For example, a left parenthesis must have a matching right parenthesis, a NOT must follow either a constant, identifier or a left parenthesis, and an operator must have a left and right hand side.

Limitations

The program is limited to three variable identifiers P, Q, and R. R would not be evaluated if P and Q does not exist and Q would not be evaluated if P is not initiated.