**CS323 Documentation**

About 2-3 pages

**Predictive Table-Driven Parser Edition**

# **1. Problem Statement**

The objective of the second project is to implement a syntax analyzer based on our lexical analyzer. The syntax parser should use one of the methods shown in class (RDP, predictive recursive descent parser, table driven predictive parser, top-down, bottom-up) and print out the parsed tokens and lexemes with their respective production rules. Ideally store and print out the parse tree used for analyzing the tokens. For this final iteration we used a predictive table based parser and attained the best results, though some shortcomings had to be resolved with hard-coded hotfixes.

**2. How to use your program**

1) Move any text files you want to test into the folder named Table.

1b) If running in a terminal, change directory first to Code and again to Table.

2) Run grammar.exe (if coming from step 1b, compile & run grammar.cpp)

3) When prompted, type in the name of the text file including the .txt extension. If successful, go to step 4. If unsuccessful, go back to step 1 if the text file is missing. Alternatively, the user can verify the correct spelling of the .txt file and repeat step 2.

4) View results in the terminal

5) The program will proceed in one of two ways:

5^) If all sentences (lines) in the file are syntactically correct, the program will print all the tokens/lexemes with their corresponding production rules, followed by a successful completion message and time benchmark at the end.

5\_) Otherwise if any syntax error is detected, the program will output all tokens, lexemes and production rules up until the token/lexeme with offending syntax is encountered. Here the program prints a message summarizing the grammar violation and terminates early with an exit code.

# **3. Design of your program**

< write major components of your program. Also, data structures you are utilizing, particular algorithms you have chosen etc. >

We use a modified lexer.cpp from the first project as our tokenizer, this class also defines the Token struct. Token holds three key pieces of information: the token’s number as defined by the state table rules; the name of the token’s lexeme type (e.g. IDENTIFIER), and the string contents of the lexeme itself.

The lexer() function defined in lexer\_v101.cpp is called in grammar\_v101.cpp. This processes a line of the text input until the end of the file is reached. A token of vectors accumulates the results of calling lexer() on each line. A significant modification from the previous iteration is the new version will check for and automatically insert a semicolon at the end of every line that does not contain one. Without this fix, the program cannot handle expression.txt at all because it lacks semicolons.

Once the file I/O is complete the token vector carries the entire list of tokens and lexemes. Sentence (line) separation has been artificially enforced by semicolons and we are ready for the syntactical analysis. At this point in the main function the strategy diverges from the previous iteration that was using the RDP. The ifstream is flushed as soon as lexer() has processed the last line of the file. A helper class Expr from file expr.cpp is brought in to handle the vast majority of the syntax analysis.

The new table-driven predictive parser marks a significant departure from prior iterations.

Data structure usage is rather wasteful and inefficient. The stack that interacts with the table-driven rules is purged each time it reaches the end of a line or an equals sign, by popping one element at a time. A deque named “fullstack” also interacts with this stack and its only purpose is to accumulate the stack contents in order to (attempt to) print out the parse tree in the correct order. Whenever the stack is emptied, this deque is also emptied soon after while printing its contents. In addition to these two structures there is an unordered set that holds the same elements as the stack, but is solely used to convert the raw production tokens into printable, expanded production rules.

/\*For each token, a series of checks are done. First, the program assesses the state of the boolean flags duringStatement and duringExpression. If duringStatement is set to true, the program checks for the presence of a semicolon and if so, resets both duringStatement and duringExpression to false and moves to the next token which is the start of a new line in the text file.

If duringExpression is true, then the program tests if the token belongs in T (term), or alternatively T’ (term prime) and E’ (expression prime). If it is not, it moves to the next boolean block and checks if duringExpression is false. If so, it checks if the current token is the start of a valid expression. Otherwise if duringStatement is false, the program checks if the current token could be the start of a statement.

Each of the check<X> functions assess whether the token (and potentially the tokens preceding or following it) satisfies further conditions of other check<X> functions. More details are in the Software Requirements Document (tbd).\*/

# **4. Any Limitation**

The code of expr.cpp is not elegant and uses much hard-coded logic, but since it seems to give the correct output we determined the hotfixes were enough. I decided not to rewrite these parts that still produce acceptable results at a cost to code flexibility.

The parser simulates processing the tokens line by line. The forced semicolons signal boundaries between the current sentence (line) and the next. The method for checking for line breaks is to look for a semicolon at the end of each line, and insert one if not present. A more fluid implementation would try to parse the token vector one input line at a time.

# **5. Any shortcomings for each iterations**

The Assignment and Declarative statements are hard-coded into an override clause during printing in order to not interfere with the arithmetic expressions. We were unsure how to implement Declaratives so they merely follow the Assignment statements.

As of the latest version the syntax error handler is quite robust at stopping files with incorrect syntax, but it needs to display more information such as the line number that each error occurred on. The error handler could miss certain unexpected syntax errors.

Semicolon production rules are hard-coded. They are artificially parsed and printed because we could not figure out how to fill its column in the table without having unwanted ripple effects. The results are one extra epsilon during printing and inaccuracies in the parse tree display, but these shortcomings are limited to the semicolon only.

# **Software Requirements Document Template**

### **(Use as a guide only – does not have to be exact)**

## **1. Introduction**

### **1.1 Purpose**

### **1.2 Document Conventions**

### **1.3 Intended Audience and Reading Suggestions**

### **1.4 Product Scope**

### **1.5 References**

## **2.Overall Description**

### **2.1 Product Perspective**

### **2.2 Product Functions**

### **2.3 User Classes and Characteristics**

### **2.4 Operating Environment**

### **2.5 Design and Implementation Constraints**

### **2.6 Assumptions and Dependencies**

## **3. External Interface Requirements**

### **3.1 User Interfaces**

### **3.2 Hardware Interfaces**

### **3.3 Software Interfaces**

### **3.4 Communications Interface**

## **4. System Features**

### **4.x System Feature X**

### **4.x.1 Description and Priority**

### **4.x.2 Stimulus/Response Sequences**

### **4.x.3 Functional Requirements**

## **5. Other Nonfunctional Requirements**

### **5.1 Performance Requirements**

### **5.2 Safety Requirements**

### **5.3 Security Requirements**

### **5.4 Software Quality Attributes**

### **5.5 Business Rules**

### **5.6 User Documentation**

## **6. Other Requirements**

## **Appendix A: Glossary**

## **Appendix B: Analysis Models**

### **Data Flow Diagrams (DFD)**

### **Class Diagrams**

### **State Transition Diagrams**

### **Entity Relationship Models**