# **Syntax Analyzer**

**CS323 Programming Assignments**

**Names:**

Noah Sanderson, Section 07

Bethany Bui, Section 07

Ryann Stock, Section 07

**Assignment #2**

**Due Date:**

04/06/2025

**Submission Date:**

4/07/2025

**Executable File Name:**

NONE

**Test Case Files:**

1. Input [test\_syntax1.txt] Output [output\_syntax1.txt]
2. Input [test\_syntax2.txt] Output [output\_syntax2.txt]
3. Input [test\_syntax3.txt] Output [output\_syntax3.txt]

**Operating System:**

Windows

**Comments and Grade:**

# CS323 Documentation

## **Problem Statement**

Our objective for this assignment is to create a Syntax Analyzer (parser) for the Rat25S programming language. The parser will be able to read a source code file and break down the data into a structured format that will be easier to analyze and use by other software.

**Required Syntax Rules:**

* R1. <Rat25S> ::= $$ <Opt Function Definitions> $$ <Opt Declaration List> $$ <Statement List> $$
* R2. <Opt Function Definitions> ::= <Function Definitions> | <Empty>
* R3. <Function Definitions> ::= <Function> | <Function> <Function Definitions>
* R4. <Function> ::= function <Identifier> ( <Opt Parameter List> ) <Opt Declaration List> <Body>
* R5. <Opt Parameter List> ::= <Parameter List> | <Empty>
* R6. <Parameter List> ::= <Parameter> | <Parameter> , <Parameter List>
* R7. <Parameter> ::= <IDs > <Qualifier>
* R8. <Qualifier> ::= integer | boolean | real
* R9. <Body> ::= { < Statement List> }
* R10. <Opt Declaration List> ::= <Declaration List> | <Empty>
* R11. <Declaration List> := <Declaration> ; | <Declaration> ; <Declaration List>
* R12. <Declaration> ::= <Qualifier > <IDs>
* R13. <IDs> ::= <Identifier> | <Identifier>, <IDs>
* R14. <Statement List> ::= <Statement> | <Statement> <Statement List>
* R15. <Statement> ::= <Compound> | <Assign> | <If> | <Return> | <Print> | <Scan> | <While>
* R16. <Compound> ::= { <Statement List> }
* R17. <Assign> ::= <Identifier> = <Expression> ;
* R18. <If> ::= if ( <Condition> ) <Statement> endif | if ( <Condition> ) <Statement> else <Statement> endif
* R19. <Return> ::= return ; | return <Expression> ;
* R20. <Print> ::= print ( <Expression>);
* R21. <Scan> ::= scan ( <IDs> );
* R22. <While> ::= while ( <Condition> ) <Statement> endwhile
* R23. <Condition> ::= <Expression> <Relop> <Expression>
* R24. <Relop> ::= == | != | > | < | <= | =>
* R25. <Expression> ::= <Expression> + <Term> | <Expression> - <Term> | <Term>
* R26. <Term> ::= <Term> \* <Factor> | <Term> / <Factor> | <Factor>
* R27. <Factor> ::= - <Primary> | <Primary>
* R28. <Primary> ::= <Identifier> | <Integer> | <Identifier> ( <IDs> ) | ( <Expression> ) | <Real> | true | false
* R29. <Empty> ::= ε

**Removal of left recursion:**

* R3: <Function Definitions>
* R6: <Parameter List>
* R14: <Statement List>
* R18: <If>
* R25: <Expression>
* R26: <Term>

**Required detailed output:**

* Prints tokens, lexemes, and production rules used during parsing

**Required error handling:**

* Provides detailed syntax error messages with token details

## **How to use the program**

To use the program, you will install the executable package containing our syntax analyzer. Main.py will handle the test cases **(e.g. test\_syntax1.txt)**. After you obtain this document, please insert it in the same folder as the analyzer executable. Upon installation, you can run the program using **python main.py** and it should automatically run test cases in main.py and create input/output files. This file, as mentioned previously, should be located in the same directory as the executable. To begin using the analyzer, you will need to provide the following. Upon examination, syntax analyzer will breakdown the tokens received from the lexical analyzer and further separate them for easier readability .It will then produce tokens and identify the values as well as where everything is being processed. Upon completion of the analyzer, the program will write the output file in the same folder as the executable and will be ready for inspection.

## **Design of program**

For this portion of the program we used the lexical analyzer from the previous assignment as our base in order to help generate the tokens needed for the parser to function correctly. The token class will separate the values it is reading in the test cases and apply them to their according category. The lexer class performs its analysis through the use of FSM’s to help tokenize what is found in the input files. This class is also able to skip whitespaces as well as invalid characters to make the tokenization seamless. The parser class which is the newest addition to the project implements recursive parsing to help validate the grammar. One thing about the parser class is that even though it is a top down parser we have removed left recursion in order to remove the possibility of infinite loops during compilation. The parser primarily handles all the basic grammar rules which can be found within the code for the parser itself.

### **NFSM**

* **Identifiers**

RE: [a-zA-Z][a-zA-Z0-9]\*

(Begin) –[a-zA-Z] → (state 1) –[a-zA-Z0-9]\* → (Accepted)

* **Integers**

RE: [0-9]+

(Begin) –[0-9]+ → (accepted)

* **Real Numbers**

RE: [0-9]+ \. [0-9]+

(Begin) –[0-9]+ → (state 1) –0[.]. → (state 2) – [0-9]+ → (accepted)

## **Limitations**

**Limited error handling:** The parser stops at the first syntax error instead of attempting to recover and continue analysis.

**Semantic Check:** The analyzer verifies syntax structure but doesn’t perform semantic checks  **(e.g., type compatibility or undefined variables)**.

**Memory constraints:** The output file can be substantially large due to detailed production rule logging.

## **Shortcomings**

**Performance on Complex Expressions:** Nested expressions might require more processing time due to the recursive nature of the parser.

**Limited Context in Error Messages:** While error messages indicate the token where an error was detected, they may not always provide the broader context needed to understand the issue.