COVER PAGE

CS323 Programming Assignments

1. **Name:**	**Section:**
Conner Robbins	8
Darren Chen	8
Mariah Salgado	8
Affan Siddiqui	7

2.	Assignment Number	1
3.	Due Date	20250406
4.	Submission Date	20250406
5.	Executable File name	main.exe

(A file that can be executed without compilation by the instructor, such as .exe, .jar, etc - **NOT** a source file such as .cpp)

6. Names of testcase files	input test file	output test file
sample_rat25s	`sample_rat25s.txt`	`sample_rat25s_lexer_output.txt, sample_rat25s_parser_output.txt`
newtestcase1	`newtestcase1.txt`	`newtestcase1_lexer_output.txt, newtestcase1_parser_output.txt`
newtestcase2	`newtestcase2.txt`	`newtestcase2_lexer_output.txt, newtestcase2_parser_output.txt`
newtestcase3	`newtestcase3.txt`	`newtestcase3_lexer_output.txt, newtestcase3_parser_output.txt`

7.	Operation System	Windows 11, macOS 15 Sequoia

To be filled out by the Instructor:

Comments and Grade:

CS323 Documentation

1. Problem Statement

This assignment requires the development of a syntax analyzer for the Rat25S programming language. The syntax analyzer must validate the structure of the source code by checking whether the sequence of tokens conforms to the grammar rules of the Rat25S programming language.

The main requirements:

- Look at Rat25S grammar and check to see if there are any left-recoursences or backtracking in any of the statements. If so, rework statements to prevent such.
- Examine the current group code and prepare the code so that it can be a stable foundation for the syntax analyzer.
- For the syntax analyzer portion of the compiler project, create a recursive descent parser to parse through tokens created from lexar() and should print to an output file the tokens, lexemes and the production rules used.
- If a syntax error occurs, your parser should generate a meaningful error message, such as the token, lexeme, line number, error type, etc. Then, your program may exit, or you may continue for further analysis.
- Ensure the program may compile as well as parse through the Rat25S source code.

2. How to use your program

1. **Compilation**:

No compilation is required; we are using Python library pyinstaller to transform .py into .exe.

2. **Execution**.

Ensure the test case is being used in the same folder as file_read.py (essentially where the other .py as well as .txt files are.)

To run the program to evaluate the team's syntax analyzer navagite as so.

```
CPSC_323_Compiler_Project / dist / main / main.exe
```

Drop a testcase textfile in CPSC_323_Compiler_Project2

Output Format:

```
<Rat25S> -> $$ <Opt Function Definitions> $$ <Opt Declaration List> $$ <Statement List> $$
Matched: Token: SEPARATOR, Lexeme: $$
<Opt Function Definitions> -> <Function Definitions> | ε
<Function Definitions> -> <Function> <Function Definitions Prime>
<Function> -> function <Identifier> ( <Opt Parameter List> ) <Opt Declaration List> <Body>
Matched: Token: KEYWORD, Lexeme: function
Matched: Token: Identifier, Lexeme: convertx
Matched: Token: SEPARATOR, Lexeme: (
<Opt Parameter List> -> <Parameter List> | ε
<Parameter List> -> <Parameter> <Parameter List Prime>
<Parameter> -> <Identifier> <Qualifier>
Matched: Token: Identifier, Lexeme: fahr
<Qualifier> -> integer | boolean | real
Matched: Token: KEYWORD, Lexeme: integer
<Parameter List Prime> -> ε
Matched: Token: SEPARATOR, Lexeme: )
<Opt Declaration List> -> <Declaration List> | ε
<Opt Declaration List> -> ε
<Body> -> { <Statement List> }
Matched: Token: SEPARATOR, Lexeme: {
<Statement List> -> <Statement> <Statement List Prime>
<Statement> -> <Assign> | <If> | <While> | <Return> | <Scan> | <Print> | <Block>
<Return> -> return <Expression> ;
Matched: Token: KEYWORD, Lexeme: return
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Real> | ( <Expression> )
Matched: Token: Identifier, Lexeme: 5
<Term Prime> -> * <Factor> <Term Prime> | / <Factor> <Term Prime> | % <Factor> <Term Prime>
Matched: Token: OPERATOR, Lexeme: *
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Real> | ( <Expression> )
Matched: Token: SEPARATOR, Lexeme: (
<Expression> -> <Term> <Expression Prime>
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Real> | ( <Expression> )
Matched: Token: Identifier, Lexeme: fahr
<Term Prime> -> ε
<Expression Prime> -> + <Term> <Expression Prime> | - <Term> <Expression Prime>
Matched: Token: OPERATOR, Lexeme: -
<Term> -> <Factor> <Term Prime>
<Factor> -> - <Primary> | <Primary>
<Primary> -> <Identifier> | <Integer> | <Real> | ( <Expression> )
Matched: Token: Identifier, Lexeme: 32
<Term Prime> -> ε
<Expression Prime> -> ε
Matched: Token: SEPARATOR, Lexeme: )
```

3. Design of your program

Major Components

1. **Main Program**: main.py
    a. This module will serve as the main hub for the compilation process.
2. **Modules**:
    a. **Lexar Function**:lexar_component.py
        i. Tokenizing the input through the queue_hub function
        ii. Refining the tokens with the lex_hub function and FSMs
        iii. Write the output to <file_intake_name>_output.txt
    b. **Syntax Analyzer**:syntax.py
        i. Receives queue of tokens and parses the tokens via Rat25S grammar which consists of.
        ii. Verifies if the current token matches what's expected based on the grammar rules.
        iii. It reports errors when the code may not match the expected language rules.

Data Structures

1. **Token Record**: A tuple containing (lexeme, token_type) where:
    a. lexeme (string): The actual text from the source code
    b. token_type (string): The classification of the token (e.g., "KEYWORD", "IDENTIFIER", "INTEGER", "REAL", "OPERATOR", "SEPARATOR")
2. **Recursive Descent Parser**: Consists of 39 different functions that intake tokens via the lexer function created to parse through source code to see if proper grammar was utilized.
3. **Rule Record**: A collection of Rat25s grammar rules that are corralled into an array and are called when its corresponding function is being called.
4. **debug**: A switch that can be set to either display where the program is in the parsing processes or not.

Rat25S Grammar

```
R1. <Rat25S>  ::=   $$ <Opt Function Definitions>   $$ <Opt Declaration List>  $$ <Statement List>  $$
    -    This is how you are able to construct Rat25
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>   ::= 🎁
```

Step 1: List all Non Terminal Symbols (In the Sequence in which they occur)

1. <Rat25S>
2. <Opt Function Definitions>
3. <Function Definitions>
4. <Function>
5. <Opt Parameter List>
6. <Parameter List>
7. <Parameter>
8. <Qualifier>
9. <Body>
10. <Opt Declaration List>
11. <Declaration List>
12. <Declaration>
13. <IDs>
14. <Statement List>
15. <Statement>
16. <Compound>
17. <Assign>
18. <If>
19. <Return>
20. <Print>
21. <Scan>
22. <While>
23. <Condition>
24. <Relop>
25. <Expression>
26. <Term>
27. <Factor>
28. <Primary>
29. <Empty>

Step 2: {For each NT (N) do

If RHS begins with a NT (A) earlier in the list then

- Substitute A

- Remove any direct recursion}

1. <Rat25S>
   a. <Rat25S> ::= $$ <Opt Function Definitions> $$ <Opt Declaration List> $$ <Statement List> $$
2. <Opt Function Definitions>
   a. <Opt Function Definitions> ::= <Function Definitions> | <Empty>
3. <Function Definitions>
   a. <Function Definitions> ::= <Function>
   b. <Function Definitions> ::= <Function> <Function Definitions>
4. <Function>
   a. <Function> ::= function <Identifier> (<Opt Parameter List>) <Opt Declaration List> <Body>
5. <Opt Parameter List>
   a. <Opt Parameter List> ::= <Parameter List> | <Empty>
6. <Parameter List>
   a. R6. <Parameter List> ::= <Parameter> | <Parameter> , <Parameter List>
7. <Parameter>
   a. <Parameter> ::= <IDs > <Qualifier>
8. <Qualifier>
   a. <Qualifier> ::= integer | boolean | real
9. <Body>
   a. <Body> ::= { < Statement List> }
10. <Opt Declaration List>
    a. <Opt Declaration List> ::= <Declaration List> | <Empty>
11. <Declaration List>

   a. <Declaration List> := <Declaration> ; | <Declaration> ; <Declaration List>

12. <Declaration>
    a. ~~<Declaration> ::= <Qualifier > <IDs>~~
    b. <Declaration> ::= integer <IDs> | boolean <IDs> | real <IDs>
13. <IDs>
    a. <IDs> ::= <Identifier> | <Identifier>, <IDs>
14. <Statement List>
    a. <Statement List> ::= <Statement> | <Statement> <Statement List>
15. <Statement>
    a. <Statement> ::= <Compound> | <Assign> | <If> | <Return> | <Print> | <Scan> | <While>
16. <Compound>
    a. <Compound> ::= { <Statement List> }
17. <Assign>
    a. <Assign> ::= <Identifier> = <Expression> ;
18. <If>
    a. <If> ::= if (<Condition>) <Statement> endif |
19. <Return>
    a. <Return> ::= return ; | return <Expression> ;
20. <Print>
    a. <Print> ::= print (<Expression>);
21. <Scan>
    a. <Scan> ::= scan (<IDs>);
22. <While>
    a. <While> ::= while (<Condition>) <Statement> endwhile
23. <Condition>
    a. <Condition> ::= <Expression> <Relop> <Expression>
24. <Relop>
    a. <Relop> ::= == | != | > | < | <= | =>
25. <Expression>

~~a. <Expression> ::= <Expression> + <Term>~~

~~b. <Expression> ::= <Expression> - <Term>~~

~~c. <Expression> ::= <Term>~~

~~Introduce new Non-Terminal symbol <Expression Prime>~~

~~(a+c). <Expression> ::= <Term> <Expression Prime>~~

~~<Expression Prime> ::= + <Term> <Expression Prime> | ε~~

~~(b+c). <Expression> ::= <Term> <Expression Prime>~~

~~<Expression Prime> ::= - <Term> <Expression Prime> | ε~~

~~So,~~

~~d. <Expression> ::= <Term> <Expression Prime>~~

~~e. <Expression Prime> ::= + <Term> <Expression Prime> | ε~~

~~f. <Expression> ::= <Term> <Expression Prime>~~

~~g. <Expression Prime> ::= - <Term> <Expression Prime> | ε~~

~~Combining:~~

h. <Expression> ::= <Term> <Expression Prime>

i. <Expression Prime> ::= + <Term> <Expression Prime>| - <Term> <Expression Prime> | ε

26. <Term>

~~a. <Term> ::= <Term> * <Factor>~~

~~b. <Term> ::= <Term> / <Factor>~~

~~c. <Term> ::= <Factor>~~

~~Introduce new Non-Terminal symbol <Term Prime>~~

~~(a+c). <Term> ::= <Factor> <Term Prime>~~

~~<Term Prime> ::= * <Factor> <Term Prime> | ε~~

~~(b+c). <Term> ::= <Factor> <Term Prime>~~

~~<Term Prime> ::= / <Factor> <Term Prime> | ε~~

~~So,~~

~~d. <Term> ::= <Factor> <Term Prime>~~

~~e. <Term Prime> ::= * <Factor> <Term Prime> | ε~~

~~f. <Term> ::= <Factor> <Term Prime>~~

g. ~~<Term Prime> ::= / <Factor> <Term Prime> | ε~~

~~Combining:~~

h. <Term> ::= <Factor> <Term Prime>

i. <Term Prime> ::= * <Factor> <Term Prime> | / <Factor> <Term Prime> | ε

27. <Factor>
    a. <Factor> ::= - <Primary> | <Primary>
28. <Primary>
    a. <Primary> ::= <Identifier> | <Integer> | <Identifier> (<IDs>) | (<Expression>) | <Real> | true | false
29. <Empty>
    a. <Empty> ::= ε

Removing Backtracking using Left Factorization:

R3. <Function Definitions> ::= <Function> <Function Definitions Prime>

R4. <Function Definitions Prime> ::= ε | <Function Definitions>

R6. <Parameter List> ::= <Parameter> <Parameter List Prime>

R7. <Parameter List Prime> ::= ε | , <Parameter List>

R11. <Declaration List> := <Declaration> <Declaration List Prime>

R12. <Declaration List Prime> ::= ε | <Declaration List>

R13. <IDs> ::= <Identifier> <IDs Prime>

R14. <IDs Prime> ::= ε | , <IDs>

R15. <Statement List> ::= <Statement> <Statement List Prime>

R16. <Statement List Prime> ::= ε | <Statement List>

R18. <If> ::= if (<Condition>) <Statement> <If Prime>

R19. <If Prime> ::= endif | else <Statement> endif

R19. <Return> ::= return <Return Prime>

R19. <Return Prime> ::= ; | <Expression> ;

R30. <Primary> ::= <Identifier> <Primary Prime> | <Integer> | (<Expression>) | <Real> | true | false

R30. <Primary Prime> ::= ε | (<IDs>)

Rules without left recursion and backtracking are:

R1. <Rat25S> ::= $$ <Opt Function Definitions> $$ <Opt Declaration List> $$ <Statement List> $$

R2. <Opt Function Definitions> ::= <Function Definitions> | <Empty>

R3. <Function Definitions> ::= <Function> <Function Definitions Prime>

R4. <Function Definitions Prime> ::= ε | <Function Definitions>

R5. <Function> ::= function <Identifier> (<Opt Parameter List>) <Opt Declaration List> <Body>

R6. <Opt Parameter List> ::= <Parameter List> | <Empty>

R7. <Parameter List> ::= <Parameter> <Parameter List Prime>

R8. <Parameter List Prime> ::= ε | , <Parameter List>

R9. <Parameter> ::= <IDs> <Qualifier>

R10. <Qualifier> ::= integer | boolean | real

R11. <Body> ::= { <Statement List> }

R12. <Opt Declaration List> ::= <Declaration List> | <Empty>

R13. <Declaration List> ::= <Declaration> <Declaration List Prime>

R14. <Declaration List Prime> ::= ε | <Declaration List>

R15. <Declaration> ::= integer <IDs> | boolean <IDs> | real <IDs>

R16. <IDs> ::= <Identifier> <IDs Prime>

R17. <IDs Prime> ::= ε | , <IDs>

R18. <Statement List> ::= <Statement> <Statement List Prime>

R19. <Statement List Prime> ::= ε | <Statement List>

R20. <Statement> ::= <Compound> | <Assign> | <If> | <Return> | <Print> | <Scan> |
<While>

R21. <Compound> ::= { <Statement List> }

R22. <Assign> ::= <Identifier> = <Expression> ;

R23. <If> ::= if (<Condition>) <Statement> <If Prime>

R24. <If Prime> ::= endif | else <Statement> endif

R25. <Return> ::= return <Return Prime>

R26. <Return Prime> ::= ; | <Expression> ;

R27. <Print> ::= print (<Expression>);

R28. <Scan> ::= scan (<IDs>);

R29. <While> ::= while (<Condition>) <Statement> endwhile

R30. <Condition> ::= <Expression> <Relop> <Expression>

R31. <Relop> ::= == | != | > | < | <= | =>

R32. <Expression> ::= <Term> <Expression Prime>

R33. <Expression Prime> ::= + <Term> <Expression Prime> | - <Term> <Expression Prime> | ε

R34. <Term> ::= <Factor> <Term Prime>

R35. <Term Prime> ::= * <Factor> <Term Prime> | / <Factor> <Term Prime> | ε

R36. <Factor> ::= - <Primary> | <Primary>

R37. <Primary> ::= <Identifier> <Primary Prime> | <Integer> | (<Expression>) | <Real> | true | false

R38. <Primary Prime> ::= ε | (<IDs>)

R39. <Empty> ::= ε

Algorithm

The recursive descent parser reads tokens from a lexer and checks whether the sequence of tokens follows the language’s grammar rules. This works by representing each grammar rule as a function, which verifies the expected pattern of tokens.

Each function handles a specific part of the language grammar. These functions either advance through the token stream when the patterns match or report syntax errors when the patterns do not. The parser builds a parse tree through these function calls, validating that the source code follows the correct syntax structure before any further processing.

This structure is called "recursive" because functions can call other functions to check nested language elements. The debugging mode prints each grammar rule as it is being processed, showing the exact path the parser follows through the code’s structure.

4. Any Limitation

None.

5. Any shortcomings

None