



XYZ programming language

Design and Implementation

Team Members:

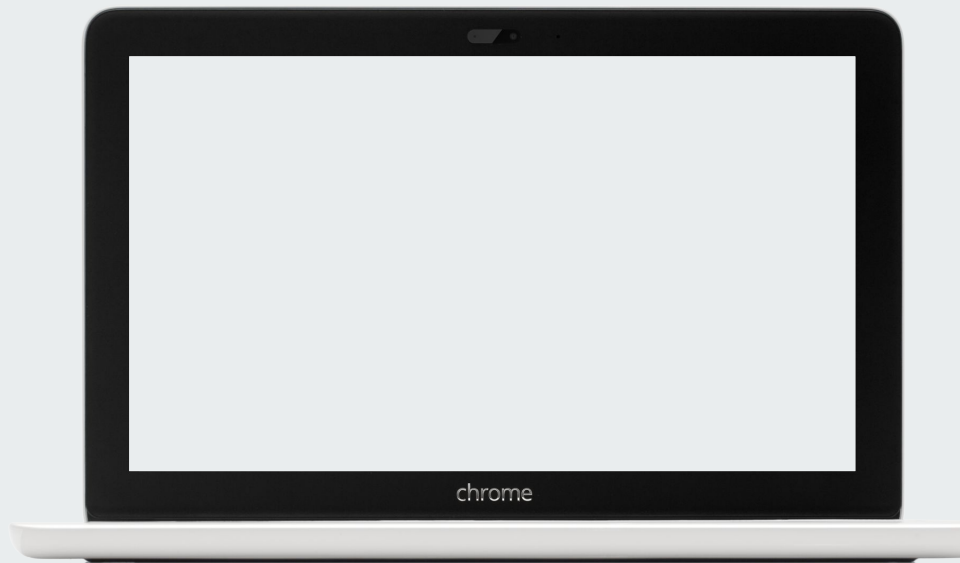
Paromita Roy

Subham Kumar

Amarjeet Singh

Siri Rachappa Jarmale

Vinay Kantilal Chavhan



Outline

Introduction

Workflow followed

Design of Grammar of different elements

Lexer

Parser

Evaluator

Sample Programs

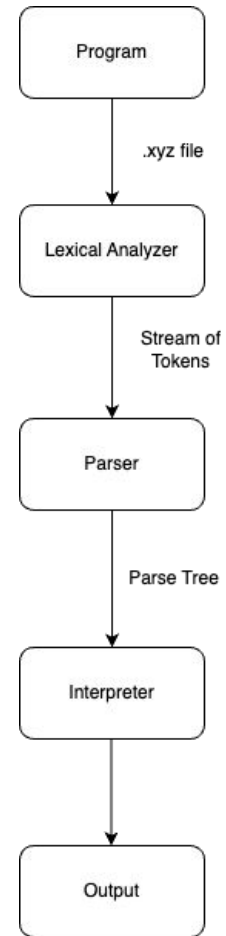
Introduction

"xyz" is a programming language that supports basic programming constructs, such as variables, control structures, and expressions. It allows developers to create programs by defining blocks of code enclosed in curly braces {}. The language supports integers, strings, and boolean values as data types, and variables can be assigned values using the assignment operator =.

The language provides a variety of control statements, including if-else statements, while loops, for loops, and for-range loops, which are used for flow control in a program. Arithmetic expressions can include addition, subtraction, multiplication, and division, and boolean expressions can include logical AND, OR, and NOT operations.

Additionally, the language supports ternary expressions, which are used to conditionally evaluate expressions based on a boolean condition. Finally, the print statement is used to output the value of an expression to the console. "xyz" is a simple and powerful programming language that can be used to create a wide range of programs.

Workflow



Design of grammar

grammar XYZ;

program : '{' block '}';

block : declaration_command block|
declaration_command : datatype identifier '.' | datatype assignment_operator '
| assignment_operator '.' | control_statements | print_statement;

assignment_operator : identifier '=' expression ;

expression : digits | identifier | string | bool_expression | arithmetic_expression
| ternary_expression ;

bool_expression : 'not' identifier | 'not' bool_values | identifier 'and' identifier |
identifier 'or' identifier | bool_values 'and' bool_values | bool_values 'or'
bool_values | bool_values ;

arithmetic_expression : arithmetic_expression1 '+' arithmetic_expression |
arithmetic_expression1 '-' arithmetic_expression | arithmetic_expression1;

arithmetic_expression1 : ident_number '*' arithmetic_expression1 |
ident_number '/' arithmetic_expression1 | ident_number;

ident_number : identifier | digits ;

control_statements : if_control_statement | while_control_statement |
for_control_statement | for_range_control_statement ;

if_control_statement : 'if' '(' conditional_statement ')' 'then' '{' block '}' 'else' '{' block '}' ;

while_control_statement : 'while' '(' conditional_statement ')' '{' block '}';

for_control_statement : 'for' '(' for_update ';' conditional_statement ';' for_update ')' '{' block
'}';

for_update : identifier '=' expression ;

for_range_control_statement: 'for' identifier 'in' 'range' '(' digits ',' digits ')' '{' block '}'

ternary_expression : '(' identifier conditional_operators identifier ')' '?' expression ':' expression ;

conditional_statement : expression conditional_operators expression | bool_values ;

identifier : lowerChar (upperChar |lowerChar | digits)* ;

conditional_operators : '=='| '<| '>| '<='| '>='| '!=';

bool_values : 'true'| 'false' ;

print_statement : 'print' '(' expression ')' '!';

datatype : 'int' | 'string' | 'bool' ;

digits : (digit)+;

string: "" (lowerChar | upperChar)+ "" ;

digit : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;

lowerChar : 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' |
'y' | 'z' ;

upperChar : 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V'
| 'W' | 'X' | 'Y' | 'Z' ;

Comment_statement : '/' '#' (.)*? '#' -> skip;

WS: ['\t\r\n']+ -> skip ;

The lexer.py file

The first few lines of the file import the necessary libraries, including the `sys` module for accessing command-line arguments and the `ply.lex` module for creating the lexer.

Next, the `tokens` tuple is defined, which lists all of the valid tokens in the "xyz" language. Each token is represented by a string that describes its name, such as 'RES' for reserved keywords or 'INT' for integers.

After defining the tokens, the file defines a number of functions that use regular expressions to match each token type. For example, the `t_RES` function uses a regular expression to match any of the reserved keywords in the language (`int`, `string`, `bool`, and `print`). Similarly, the `t_INT` function matches any sequence of digits, while the `t_IDENTIFIER` function matches any valid identifier (a string consisting of letters, digits, and underscores that doesn't start with a digit).

Some of the other functions define more complex regular expressions to match specific patterns. For example, the `t_OPERATOR` function matches any of the arithmetic or bitwise operators (`+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `~`, `<`, `>`, and `==`), as well as the assignment operator `=`. It checks if the matched operator is the equality operator `==` and if so, sets the token type to 'OPERATOR'.

The `t_CONTROL_STATEMENT` and `t_BOOLEAN` functions match the control statements (if, else, while, for, in, and range) and the boolean operators (and, or, and not), respectively.

The `t_LEFT_BRACE`, `t_RIGHT_BRACE`, `t_LEFT_PARAN`, and `t_RIGHT_PARAN` functions match the various types of parentheses and braces used in the language.

The `t_SPECIAL_CHAR` function matches any other single character or semicolon, which could be used as a separator.

The `t_ignore` variable specifies any characters that should be ignored by the lexer, such as spaces, tabs, and newline characters.

The `t_error` function is called when the lexer encounters an unknown text, which raises a `TypeError` with a message indicating the unrecognized text.

Finally, the main function reads in the input file line by line, removes any leading or trailing whitespace, and passes each line to the lexer. The lexer breaks each line into individual tokens using the regular expressions defined earlier and appends them to a list. Once all the lines have been processed, the resulting list of tokens is written to a file called `tokens.txt`. If the input file does not have the `.xyz` extension, the program prints an error message and exits.

The parser.pl file

The parser.pl file contains the syntax analysis of the programming language, which takes the output from the lexical analysis done by the lexer. The parser uses context-free grammar to identify the syntactic structure of the program. The parser uses the DCG (Definite Clause Grammar) format to define the grammar rules. The starting point of the parser is the program rule, which represents the entire program. The program rule consists of the block rule, which contains a declaration and a command. The declaration rule can be a single declaration or multiple declarations, separated by a period. The single declaration rule contains a data type, an identifier, and an expression. The command rule can be a single command or multiple commands, separated by a period. The expression rule consists of the term rule or an expression added or subtracted from a term. The term rule consists of the term1 rule or a term multiplied or divided by a term1. The term1 rule is either an identifier or a number. The command rule consists of a single command, which can be an assignment, a while loop, a for loop, an if-else statement, or a ternary operator. The single command assignment rule consists of an identifier and an expression. The while loop consists of a boolean expression and a command to be executed.

The for loop consists of an identifier, a range of numbers, and a command to be executed. The for loop can also be a for loop with an initialization, condition, and increment, followed by a command to be executed. The if-else statement consists of a boolean expression, a command to be executed if the expression is true, and a command to be executed if the expression is false. The ternary operator consists of a boolean expression and two values to be returned based on the boolean expression's value.

The boolean expression rule consists of an expression compared with another expression using operators such as less than, greater than, equal to, and not equal to. The boolean expression can also be a boolean not, boolean true, or boolean false.

The parser uses a table for term, expression, and declaration rules to reduce the number of comparisons necessary to parse the input. By using the table, the parser avoids left recursion, which can cause infinite recursion.

The evaluator.pl file

The code represents an interpreter for a simple programming language. The interpreter takes a program as input, which is first parsed into a syntax tree using a parser. The parser generates an abstract syntax tree, which is then passed to the evaluator.

The evaluator has several predicates, each of which evaluates a different aspect of the program. The `eval_program` predicate evaluates the entire program by calling `eval_block` on the program's syntax tree. `eval_block` evaluates a block of code, which may contain declarations and commands, by calling `eval_dec` and `eval_cmd` respectively. `eval_dec` evaluates declarations, which can either be a single declaration or a multiple declaration. `eval_singledec` evaluates a single declaration, which consists of a variable name, an optional type, and an expression. `eval_cmd` evaluates commands, which can be either a single command or a sequence of commands. `eval_cmd1` evaluates a single command, which can be an assignment, an empty command, or an if-else statement.

The `bool_eval` predicate evaluates boolean expressions, which can be either true or false. The `eval_exp` predicate evaluates expressions, which can be arithmetic expressions, variable references, or numeric literals. The arithmetic expressions can be addition, subtraction, multiplication, division, or parentheses. The interpreter also includes two helper functions: `lookup` and `update`. `lookup` searches for a variable in the environment and returns its value.

Steps to run a program in our language:



Step 1: The only requirement is to run this: `pip3 install -r requirements.txt`

Step 2: `python src/lexer.py data/iffelse.xyz`

Step 3: `python src/lexer.py data/forrange.xyz`

Step 4: `python src/lexer.py data/ternary.xyz`

Step 5: `python src/lexer.py data/add.xyz`

Step 6: `python src/lexer.py data/div.xyz`

Step 7: `python src/lexer.py data/subtract.xyz`

Step 8: `python src/lexer.py data/multiplication.xyz`

Step 9: `python src/lexer.py data/while.xyz`

Working of add,subtract, multiplication and division operator

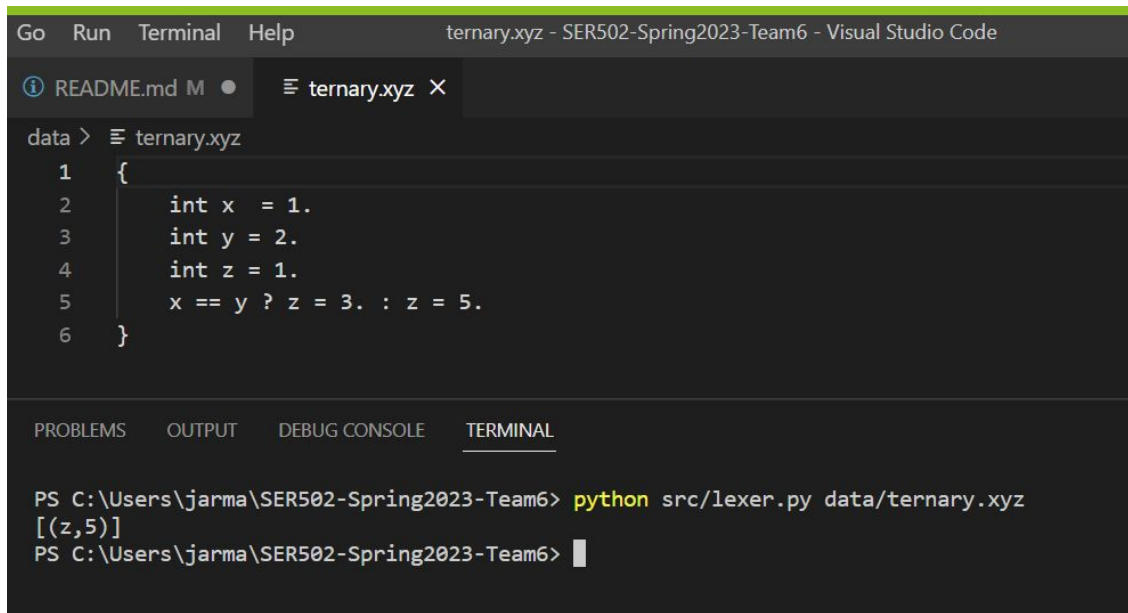
```
Go Run Terminal Help add.xyz - SER502-Spring2023-Team6 - Visual Studio Code
① README.md M • add.xyz X
data > add.xyz
1 {
2   int x = 0.
3   int y = 1.
4   int z = 1.
5   z = x + y .
6 }
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\jarma\SER502-Spring2023-Team6> python src/lexer.py data/add.xyz
[(x,0),(y,1),(z,1)]
PS C:\Users\jarma\SER502-Spring2023-Team6>
```

```
Go Run Terminal Help subtract.xyz - SER502-Spring2023-Team6 - Visual Studio Code
① README.md M contribution.txt M add.xyz subtract.xyz U X
data > subtract.xyz
1 {
2   int x = 9.
3   int y = 5.
4   int z = 1.
5   z = x - y .
6 }
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\jarma\SER502-Spring2023-Team6> python src/lexer.py data/subtract.xyz
[(x,9),(y,5),(z,4)]
PS C:\Users\jarma\SER502-Spring2023-Team6>
```

```
Go Run Terminal Help multiplication.xyz - SER502-Spring2023-Team6 - Visual Studio Code
① README.md M subtract.xyz U add.xyz multiplication.xyz U X
data > multiplication.xyz
1 {
2   int x = 0.
3   int y = 1.
4   int z = 1.
5   z = x * y .
6 }
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\jarma\SER502-Spring2023-Team6> python src/lexer.py data/multiplication.xyz
[(x,0),(y,1),(z,0)]
PS C:\Users\jarma\SER502-Spring2023-Team6>
```

```
Go Run Terminal Help div.xyz - SER502-Spring2023-Team6 - Visual Studio Code
① README.md M subtract.xyz U add.xyz multiplication.xyz U div.xyz U X
data > div.xyz
1 {
2   int x = 4.
3   int y = 2.
4   int z = 1.
5   z = x / y .
6 }
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\jarma\SER502-Spring2023-Team6> python src/lexer.py data/div.xyz
[(x,4),(y,2),(z,2)]
PS C:\Users\jarma\SER502-Spring2023-Team6>
```

Working of ternary operator:



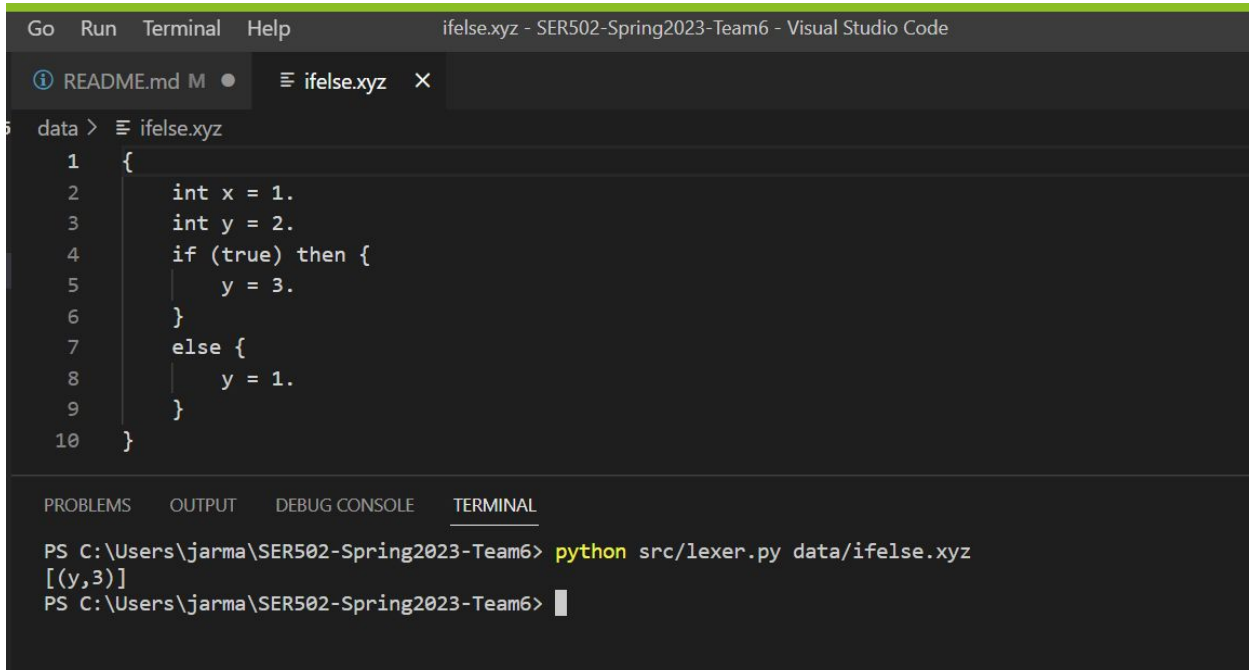
The screenshot displays the Visual Studio Code interface. The top menu bar includes 'Go', 'Run', 'Terminal', and 'Help'. The title bar reads 'ternary.xyz - SER502-Spring2023-Team6 - Visual Studio Code'. The Explorer sidebar on the left shows 'README.md' and 'ternary.xyz'. The main editor area displays the following C code in 'ternary.xyz':

```
data > ternary.xyz
1 {
2     int x = 1.
3     int y = 2.
4     int z = 1.
5     x == y ? z = 3. : z = 5.
6 }
```

The bottom panel shows the 'TERMINAL' tab with the following output:

```
PS C:\Users\jarma\SER502-Spring2023-Team6> python src/lexer.py data/ternary.xyz
[(z,5)]
PS C:\Users\jarma\SER502-Spring2023-Team6> |
```

Working of ifelse condition



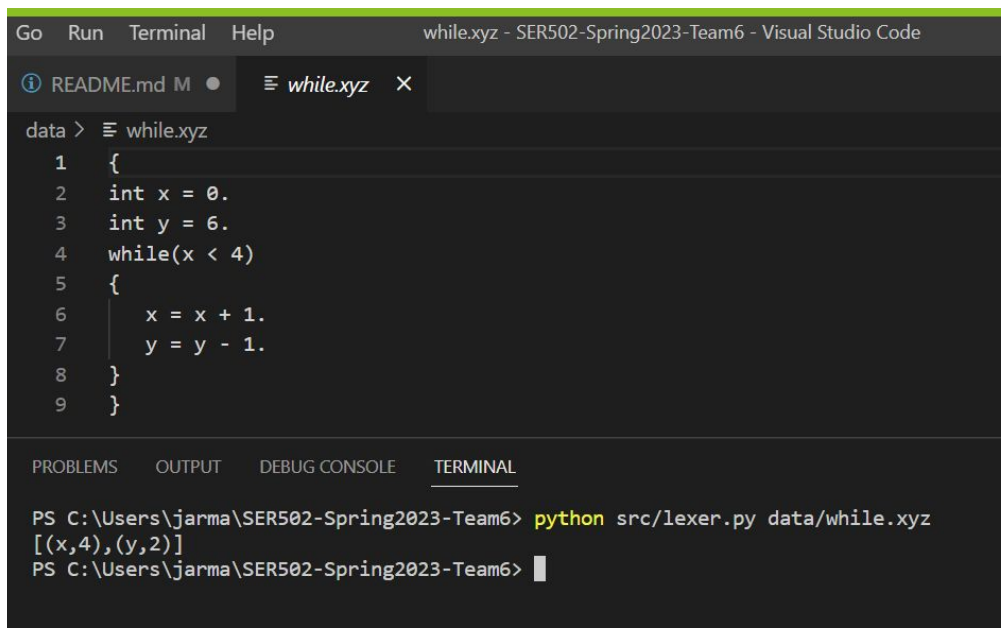
The screenshot shows the Visual Studio Code interface. The top bar indicates the file is 'ifelse.xyz - SER502-Spring2023-Team6 - Visual Studio Code'. The editor window displays a Python script in 'ifelse.xyz' with the following code:

```
1 {  
2     int x = 1.  
3     int y = 2.  
4     if (true) then {  
5         y = 3.  
6     }  
7     else {  
8         y = 1.  
9     }  
10 }
```

The bottom panel shows the 'TERMINAL' tab with the following commands and output:

```
PS C:\Users\jarma\SER502-Spring2023-Team6> python src/lexer.py data/ifelse.xyz  
[(y,3)]  
PS C:\Users\jarma\SER502-Spring2023-Team6> 
```

Working of while condition:



The screenshot shows a Visual Studio Code window with a file named `while.xyz` open. The code in the editor is a Python script that initializes `x` to 0 and `y` to 6, then enters a `while` loop that runs as long as `x` is less than 4. Inside the loop, `x` is incremented by 1 and `y` is decremented by 1. The terminal at the bottom shows the command `python src/lexer.py data/while.xyz` being executed, which outputs `[(x,4),(y,2)]`.

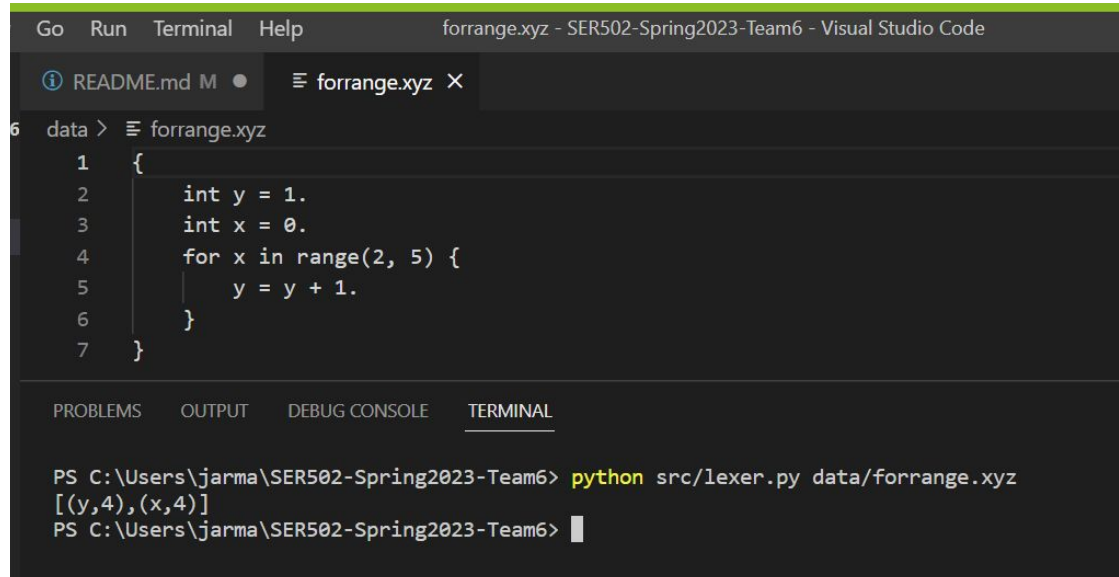
```
Go Run Terminal Help while.xyz - SER502-Spring2023-Team6 - Visual Studio Code

data > while.xyz
1 {
2   int x = 0.
3   int y = 6.
4   while(x < 4)
5   {
6     x = x + 1.
7     y = y - 1.
8   }
9 }

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

PS C:\Users\jarma\SER502-Spring2023-Team6> python src/lexer.py data/while.xyz
[(x,4),(y,2)]
PS C:\Users\jarma\SER502-Spring2023-Team6>
```

Working of forrange condition



The screenshot shows the Visual Studio Code interface. The top bar indicates the file is 'forrange.xyz' in the workspace 'SER502-Spring2023-Team6'. The editor displays a Python script with the following code:

```
data > forrange.xyz
1 {
2     int y = 1.
3     int x = 0.
4     for x in range(2, 5) {
5         y = y + 1.
6     }
7 }
```

The bottom panel shows the 'TERMINAL' tab with the following output:

```
PS C:\Users\jarma\SER502-Spring2023-Team6> python src/lexer.py data/forrange.xyz
[(y,4),(x,4)]
PS C:\Users\jarma\SER502-Spring2023-Team6> 
```




Links:

Github Link: <https://github.com/skuma251/SER502-Spring2023-Team6/>

Youtube Video Link: <https://youtu.be/XGsdB8p3TSA>