



Pamantasan ng Lungsod ng Maynila  
(University of the City of Manila)  
Intramuros, Manila  
College of Engineering and Technology  
Computer Science Department



CSC 0311-2 Automata Theory and Formal Languages

FightOn Compiler

# FIGHTON

Submitted by:

FOURTHCHEERS

**BARBOSA, GIAN KARLO J.**  
2018-10379

**CALUMBIRAN, Christine Mae D.**  
2018-10653

**CAMANGIAN, Kayla Mae M.**  
2018- 10543

**CRUZ, Nicole Anne Marie O.**  
2018-00608

**GALANG, Franc Aldrin M.**  
2018-01692

**GALIMA, Kyle Dominic J.**  
2018-10545

**GONZAGA, Dominique Kyle B.**  
2018-00916

**MANALO, Jade Ann H.**  
2018-10208

**SORIANO, Caira Marija A.**  
2018-00374

# Contents

I.	LANGUAGE OVERVIEW .....	3
II.	GENERAL RULES.....	4
III.	STRUCTURE OF THE PROGRAM .....	5
IV.	SAMPLE CODE .....	5
V.	SPECIFIC RULES.....	6
A.	CONSTANTS.....	6
B.	VARIABLES .....	7
C.	DATA TYPES AND LITERALS .....	8
D.	IDENTIFIER .....	11
E.	OPERATORS .....	12
F.	OPERATOR PRECEDENCE .....	14
G.	EXPRESSIONS.....	15
H.	STATEMENTS .....	19
I.	FUNCTIONS .....	27
J.	ARRAY .....	29
K.	TOWER .....	30
L.	COMMENT .....	33
M.	ESCAPE SEQUENCES .....	34
VI.	RESERVED WORDS .....	35
VII.	RESERVED SYMBOLS .....	37
A.	Arithmetic Operator .....	37
B.	Assignment Operator .....	37
C.	Relational Operator .....	37
D.	Unary Operator.....	38
E.	Other Symbols .....	38
VIII.	REGULAR EXPRESSION .....	40
A.	RESERVED WORDS.....	40
B.	RESERVED SYMBOLS .....	41
C.	LITERALS AND COMMENT .....	42
D.	IDENTIFIER .....	42
IX.	REGULAR DEFINITION .....	43
X.	TRANSITION DIAGRAM .....	45
A.	Reserved Words .....	45
B.	Reserved Symbols .....	46
C.	Clock Literal.....	47
D.	Hover Literal .....	47
E.	Rune and Rope Literal .....	47
F.	Identifiers .....	47
G.	Comments .....	48
XI.	CONTEXT-FREE GRAMMAR .....	49
XII.	FIRST SET.....	52
XIII.	FOLLOW SET .....	53
XIV.	PREDICT SET .....	55
XV.	PREDICT TABLE .....	58
XVI.	TEST SCRIPTS: LEXICAL .....	63
XVII.	TEST SCRIPTS: SYNTAX .....	71

## I. LANGUAGE OVERVIEW

FightØn is a programming language that is based on role-playing games (RPG) and intends to teach people who want to learn programming but are intimidated by it. Game enthusiasts can easily relate to how this language is written since the reserved words used are gaming terms. This allows the language to be more accessible to a wider variety of people as it's intended to be intuitive and familiar. The FightØn programming language is based on the structure of C, C++, and Python languages.

FighØn is a programming language developed by Team Fourth Cheers, a group of 9 fourth year Computer Science students from the College of Engineering and Technology, Pamantasan ng Lungsod ng Maynila.



## II. GENERAL RULES

1. FightØn is a case-sensitive language.
2. The program always starts and ends with the reserved words *START*, and *END* respectively.
3. The main function always starts with the reserved word *PAUSE*.
4. A program can only have one *PAUSE()* function.
5. All reserved words must be written in lowercase except for *STOCK*, *OR*, *AND*, *NOT*, *START*, *PLAY*, *PAUSE*, and *END*.
6. All global variables and user-defined functions must be declared before the *PAUSE()* function.
7. Global variables and user-defined functions are optional.
8. Local variables are declared inside a function, hence global variables are declared outside a function.
9. Same identifiers in the global declaration is invalid.
10. Same identifiers in the local declaration is invalid.
11. Having the same identifier in the global declaration and in the local declaration is valid. However, the value of the identifier in the local declaration will be used.
12. Identifiers have a maximum of 25 characters – composed of letters, numbers and underscores –, and must always start with a letter.
13. All variables must be declared before being used.
14. Variables with the same data types can be declared-initialized on a single line as long as a comma separates it.
15. Multiple declarations/initializations of variables with different data types on the same line is allowed as long as a semicolon separates it.
16. Initialization of variables is done by assigning a value to a variable with the use of the assignment operator (=).
17. The tilde symbol (~) in the beginning of a *clock* (integer) and *hover* (float) literal indicates a negative number.
18. All statements are terminated by a semicolon (;).
19. Statements such as input, output, conditional, looping statements, etc. are placed in the body of a function.
20. Statements can be null EXCEPT in input, output statements, conditional, switch, and loop statements.
21. Single-line comments begin with a single dollar sign (\$).
22. Multi-line comments begin with an open angle bracket together with a forward slash (</), and end with a forward slash together with a close angle bracket (/>).
23. Comments are optional.
24. Escaped sequences are preceded by a backslash sign (\).

### III. STRUCTURE OF THE PROGRAM

```
START
    <global>
    PAUSE():
        <statement/s>;
    PLAY
END
```

where:

```
<global> =      <constant declaration/s>;
                <variable declaration/s>;
                <function definition/s>;
                <tower declaration/s>;
<statements> = <constant declaration/s>;
                <declaration statement/s>;
                <assignment statement/s>;
                <looping statement/s>;
                <conditional statement/s>;
                <input/output statement/s>;
                <return statement/s>;
```

### IV. SAMPLE CODE

```
START
$This line is a comment
quest intro(){
    rope name;
    name = "What's your name, great warrior?";

    listen(name);
    say("Welcome to this Kingdom, ", name);

}

PAUSE():
    intro();
PLAY
END
```

## V. SPECIFIC RULES

### A. CONSTANTS

#### Declaration and Initialization of Constants

1. Declaration of constants can be done globally and locally.
2. Declaration of constants must begin with the reserved word **STOCK**, followed by a data type, an identifier, an assignment operator (=), and its value.
3. Initialization of constants must be done right after its declaration. It is done with the use of assignment operator (=).
4. A single line can have multiple constant declarations as long as they are separated by comma.

#### Syntax

```
STOCK <data type> <identifier> = <value> ;
```

#### Valid Examples

```
STOCK hover floatVar = 12.2;
```

```
STOCK clock intVar1 = 1, intVar2 = 2;
```

```
STOCK coin boolVar= heads;
```

```
STOCK rune charVar = 'c';
```

#### Invalid Examples

```
STOCK floatVar = 12.2;
```

```
STOCK clock intVar = 1, rune charVar= 'a';
```

```
STOCK coin boolVar = heads, STOCK comedy charVar = 'c';
```

## B. VARIABLES

### Declaration and Initialization of Variable

1. Declaration of variables must start with a data type, followed by an identifier.
2. Initialization of variables is done by assigning a value to a variable with the use of the assignment operator (=).

#### Syntax

```
<data type> <identifier> = <value> ;
```

#### Valid Examples

```
rune charVar1 = 'a', charVar2;
```

```
clock intVar1 = 1;
```

```
hover floatVar1 = 1.2; rope strVar1 = "string";  
clock iVar1 = 123;
```

#### Invalid Examples

```
clock intVar = 1;  
clock intVar = 2;
```

```
rune charVar1 = 'a', clock = intVar = 2;
```

```
clock intVar1 == 1;
```

## C. DATA TYPES AND LITERALS

Literals are values written exactly as it's meant to be interpreted while data types determine the type of literals a variable or an identifier can have and what operations can be performed.

Data Type	C++ Language	Proposed Language
Integer	int	clock
Decimal	float	hover
Boolean	bool	coin
Character	char	rune
String	string	rope

1. **clock** - a data type that stores positive or negative whole numbers.

### Rules for Clock Literals

- a. It can only have one (1) up to ten (10) place values.
- b. Leading zero/es will be omitted, i.e., 09 is equivalent to 9.
- c. If clock literal/s only contain zeroes, it will be treated as zero (0), i.e., 00000 is equivalent to 0.

Valid Clock Literal	Invalid Clock Literal	Reason
1234	"1234"	Enclosed with ""
0000	'0'	Enclosed with ''
123000	123000.1	Float value
12345	€10-21	Use of special characters
122222229	12222222100	Place value exceeded

2. **hover** - a data type that stores positive or negative decimal numbers.

### Rules for Hover Literals

- a. It can only have one (1) up to ten (10) place values, and one (1) up to five (5) decimal places.
- b. It must have one (1) decimal point.

- c. Any leading zeros will be omitted, i.e., 001.01 will be treated as 1.01
- d. Any trailing zeros will be omitted, i.e., 001.010 will be treated as 1.01
- e. It cannot start and end with decimal point (.), i.e., .123 and 123. are not allowed.

Valid Hover Literal	Invalid Hover Literal	Reason
123.456	"123.456"	Enclosed with ""
123.0	456.	Starts with a decimal point
0.123	.123	Starts with a decimal point
123.4567	123.45678901	Place value exceeded

3. **coin** - a data type that can only hold two values, true or false.

#### Rules for Coin Literal

- a. It can only contain the reserved words *heads* (true) and *tails* (false).
- b. Reserved words *heads* and *tails* should always be in lowercase.

Valid Coin Literal	Invalid Coin Literal	Reason
heads	Heads	Starts with uppercase 'H'
tails	"tails"	Enclosed with ""
heads	True	Not a reserved word

4. **rune** - a data type that holds a single character.

#### Rules for Rune Literal

- a. It must be enclosed within a pair of single quotations ("").
- b. The rune literal can be any printable ASCII character or a null. If it contains a single quotation ('), a backslash (\) is required to be placed before the single quotation.
- c. If a new line is intended to be used, the escape sequence newline (\n) is needed.
- d. If a tab is intended to be used, the escape sequence add tab (\t) can be used.
- e. If a backslash (\) is intended to be printed, another backslash (\) is needed.

Valid Rune Literal	Invalid Rune Literal	Reason
'A'	A	Missing single quotations(“ ”)
\\"	”	Missing backslash (\)
\\"\\	\\"	Missing backslash (\)

5. **rope** - a data that stores one or more characters.

### Rules for Rope Literal

- a. It must be enclosed with a pair of double quotations (“ ”).
- b. The rope literals can be any printable ASCII character. If it contains a double quotation(“”), a backslash (\) is required to be placed before the double quotation.
- c. If a new line is intended to be used, the escape sequence newline (\n) is needed.
- d. If a tab is intended to be used, the escape sequence add tab (\t) can be used.
- e. If backslash (\) is intended to be printed, another backslash (\) is needed.

Valid Rope Literal	Invalid Rope Literal	Reason
“player”	‘player’	Enclosed in ‘ ’
“player”	player	Not enclosed in “ ”
“\\player \\”	““player””	Missing backslash (\)
“\\player”	“player”	Missing backslash (\)

## D. IDENTIFIER

An identifier is a user-defined name of a program element that can be a function name and a variable.

### Rules for Creating Identifiers

1. Identifiers always start with a letter.
2. Identifiers can be composed of letters, numbers, or an underscore.
3. It is composed of one (1) up to fifteen (25) characters.
4. Using reserved words as identifiers is strictly prohibited.

Valid Identifier	Invalid Identifier	Reason
player	coin	Use of Reserved word
player_Name	player-name	Use Special character
anldentifier	iamaverylongidentifier	Place value exceeded

## E. OPERATORS

An operator is a symbol that represents an action or process that can manipulate a certain value or an operand.

1. **Arithmetic Operators** - a set of symbols used to perform a specific arithmetic operation.

Operator	Name	Example	Description
+	Addition	$x + y$	Used to Add two or more operands
-	Subtraction	$x - y$	Used to Subtract two or more operands
*	Multiplication	$x * y$	Used to Multiply two or more operands
/	Division	$x / y$	Used to Divide one operand with another operand.
%	Modulus	$x \% y$	Used to Find the Remainder of two operands

2. **Assignment Operators** - a set of symbols used to assign value/s to variable/s.

Operator	Name	Example	Description
=	Assignment operator	$x = 5$	Used to assign a value.

Operator	Name	Example	Description
$+=$	Add AND assignment operator	$x += 3$	Used to add operands and assign the sum to a value.
$-=$	Subtract and assignment operator	$x -= 3$	Used to subtract operands and assign the difference to a value.
$*=$	Multiply AND assignment operator	$x *= 3$	Used to multiply operands and assign the product to a value.
$/=$	Divide AND assignment operator	$x /= 3$	Used to divide operands and assign the quotient to a value.
$\%=$	Modulo AND assignment operator	$x \%= 3$	Used to divide operands and assign the remainder to a value.

3. **Logical Operators** - a set of symbols that determine the logic between two entities.

Operator	Name	Example	Description
AND	Logical AND	$x < 5 \text{ AND } x < 10$	Returns false if any of the two entities are false.
OR	Logical OR	$x < 5 \text{ OR } x < 4$	Returns true as long as one of the entities is found to be true.
NOT	Logical NOT	$\text{NOT}(x < 5 \text{ AND } x < 10)$	returns false if its single operand can be converted to true otherwise, returns true.

4. **Relational Operator** - a set of symbols that define the relationship between two entities.

Operator	Name	Example	Description
$==$	Equal	$x == y$	Used to say that two entities are equal.
$!=$	Not Equal	$x != y$	Used to say that two entities are not equal.
$>$	Greater than	$x > y$	Used to say that one entity is greater than another.
$<$	Less than	$x < y$	Used to say that one entity is less than another.

Operator	Name	Example	Description
$\geq$	Greater than or Equal to	$x \geq y$	Used to say that one entity is either greater than or equal to another.
$\leq$	Less than or Equal to	$x \leq y$	Used to say that one entity is either less than or equal to another.

5. **Unary Operators** - a set of symbols that act upon a single operand to produce a new value.

Operator	Name	Example	Description
$++$	Increment	$x++$	Adds 1 to its operand.
$--$	Decrement	$x--$	Subtracts 1 from its operand.
$\sim$	Unary negation	$\sim 123$	Negates the value of the operand.

## F. OPERATOR PRECEDENCE

Operator precedence determines the grouping of terms in an expression and decides which operator is evaluated first in an expression.

Precedence	Operator	Description	Associativity
1	( ) @	Parenthesis Member Selector	Left-to-Right
2	++ --	Postfix Increment Postfix Decrement	Left-to-Right
3	~	Negative	Right-to-Left
4	NOT	Logical NOT	Right-to-Left
5	*	Multiplication	Left-to-Right
	/	Division	
	%	Modulus	
6	+	Addition	Left-to-Right
	-	Subtraction	
7	> < >= =<	Greater than Less than Greater than or equal Less than or equal	Left-to-Right
8	== !=	Equal to Not equal to	Left-to-Right
9	AND	Logical AND	Left-to-Right
10	OR	Logical OR	Left-to-Right
11	= += -= *= /=	Simple assignment Add AND assignment Subtract AND assignment Multiply AND assignment Divide AND Assignment	Right-to-Left

## G. EXPRESSIONS

An expression is a combination of operands, and operators that represent a value.

1. **Arithmetic Expression** - contains arithmetic operators and operands.

### Rules for Arithmetic Expression

- Multiplicative (\* / % ) and additive operators(+ -) are used as infix operators. In other words, they operate on two operands and are always in between operands.
- Only clock (integer) and hover (decimal) values may be used in arithmetic expressions.
- Arithmetic expressions may be followed by an open parenthesis.
- The expression will be evaluated based on PEMDAS (Parenthesis first, followed by Exponent, then Multiplication, next is Division, then Addition, and finally Subtraction).

Syntax
<operand> <arithmetic operator> <operand> where: operand = clock literal hover literal identifier Arithmetic expression

Valid Arithmetic Expression	Invalid Arithmetic Expression	Reason
a + 3 * 3	+a3	Preceded by an arithmetic operator
a + b	a++b	Use of unary operator
1 + (a % 5)	1 (a % 5)	Missing arithmetic operator

2. **Assignment Expression** - used to set a value to a variable name.

### Rules for Assignment Expression

- It must always start with a variable, followed by an assignment operator (=), then an operand.
- An operand can be literal, a constant, a variable, and other expressions.
- Same variable can hold different values at different instants of time based on its scope.
- Operand/s on the right side should have the same data type on the left side of the assignment operator.

## Syntax

```
<identifier> <assignment operator> <operand> ;
<identifier> <assignment operator> <operand> ;
      where: operand = literal
              identifier
              expression
              function calling
```

Valid Assignment Expression	Invalid Assignment Expression	Reason
a = 12;	clock a == 12;	Variable declaration statement
b = 12 + 3;	3 + 5 = b;	Assigning value to an arithmetic expression.

3. **Relational Expression** - evaluates two or more values against each other. The results will either be *heads* (true) or *tails* (false).

### Rules for Relational Expression

- Relational operators are used as infix operators .In other words, they operate on two operands and always are always in between operands.
- The operands being compared should be of the same data type.
- An operand can be literal, a constant, a variable or other expression except the assignment expression.
- If an operand is an arithmetic expression, the arithmetic expression would be evaluated, and the resulting value would be the operand.

## Syntax

```
<operand> <relational operator> <operand>
      where: operand = literal
              identifier
              expression
```

Valid Relational Expression	Invalid Relational Expression	Reason
a < 3	a << 3	Operator followed by another operator.

a == b	a = b	Use of an assignment operator.
--------	-------	--------------------------------

4. **Logical Expression** - combines several expressions into one statement.

### Rules for Logical Expression

- a. Logical operators are used as infix operators. In other words, they operate on two operands and are always in between operands.
- b. An operand can be a literal, an identifier or a relational expression.
- c. It is evaluated based on the precedence of their operators, and returns either the boolean literal *heads* or the boolean literal *tails*.

Syntax
$\langle \text{operand} \rangle \langle \text{logical operator} \rangle \langle \text{operand} \rangle$ <i>where: </i> $\langle \text{operand} \rangle = \langle \text{relational expression} \rangle$ $\quad \quad \quad \langle \text{constant} \rangle$ $\quad \quad \quad \langle \text{identifier} \rangle$

Valid Logical Expression	Invalid Logical Expression	Reason
a AND b	a AND OR b	Operator followed by another operator.
(a > b) OR (c < d)	(1 + 2) AND heads	Use of an arithmetic expression.
NOT heads	NOTheads	Not separated by whitespace

5. **Unary Expression** - that has one operand, and a unary operator that acts upon the operand to produce a new value.

### Rules for Postfix Increment and Decrement

- a. It must consist of one operand and one unary operator.
- b. A postfix unary may only be used in identifiers.
- c. It must begin with an operand, followed by a unary operator.

Syntax
$\langle \text{operand} \rangle \langle \text{unary operator} \rangle$

<b>Valid Unary Expression</b>	<b>Invalid Unary Expression</b>	<b>Reason</b>
a++	a+ +	Space in between ‘+’
a--	--a	Begins with unary operator

### Rules for Unary Negative

- a. It must begin with the unary negative operator (~), can be followed by another negative operator or by a clock literal, hover literal, an identifier, or an arithmetic expression.

Syntax
~ <expression>

<b>Valid Unary Negative Expression</b>	<b>Invalid Unary Negative Expression</b>	<b>Reason</b>
~123	123~	Negation comes after the integer
~~abc	-abc	Use of subtraction operator

## H. STATEMENTS

An instruction that directs the computer to perform a specified action.

1. **Expression Statements** - contains an expression that will be evaluated.

Syntax
<expression>

Example
( kill + assist ) - death toxic - goodDeeds

2. **Return Statements** - used to return a value or simply pass the control to the calling function.

### Rules for Return Statements

- a. It always starts with the reserved word *return*, followed by an expression, literals, identifiers, and then a terminator.
- b. Return statements can have one or no value returned.

Syntax
return; return <expression>;

Example
return; return level - demotionPercent;

3. **Say and Listen Statements** - statements that instructs a computer how to read and process information.

- a. **Say Statements** - display data on a screen.

### Rules for Say Statements

1. It must begin with the reserved word *say*, followed by a parentheses that encloses a literal of any data type, a variable and arithmetic expression, then a terminator.

2. Numerical data, such as *clock* and *hover*, can be displayed without enclosing them with the double quotation marks ("").
3. Say statements need to have at least one parameter.
4. Comma ( , ) is used to separate multiple or a mix of literals, constants, variables, and arithmetic expressions.

### Syntax

```
say( <literal> );
say( <identifier> );
say( <arithmetic expression> );
```

### Example

```
say( "Welcome to my world!" );
say( healthPercentage - damageReceived );
```

- b. **Listen Statements** - store accepted values that the user inputs.

### Rules for Listen Statements

1. It must begin with the reserved word *listen*, followed by parentheses that encloses an identifier, then a terminator.
2. Listen statements can only have one parameter.

### Syntax

```
listen( <identifier> );
```

### Example

```
listen( playerName );
listen( inventory[0][1] );
```

4. **Conditional Statements** - statements that check a condition then execute certain parts of code depending on whether the condition is true or false.

- a. **Move Statements (punch, kick, ultimate)** - compare values and execute the block of statements if the condition is satisfied.

### Rules for Move Statements

1. Move statements must always start with the reserved word *punch* (if), followed by parentheses enclosing a condition, then a pair of curly brackets ({}).
2. The *ultimate* statement can only be initiated after the *punch*.
3. The *kick* statement can only be initiated after the *punch* statement and cannot come after the *ultimate* statement.
4. Conditions in move statements should always be enclosed with parentheses.
5. The body of move statements must always be enclosed by curly brackets ({}).

#### Syntax

```
punch ( <condition> ){
    <statements >
} kick ( <condition> ){
    <statements>
} ultimate {
    <statements>
}
```

#### Example

```
punch ( move1 == heads ){

    enemyHP = enemyHP - 5;

} kick ( move2 == heads ){

    enemyHP = enemyHP - 10;

} ultimate {

    enemyHP = enemyHP - 30;

}
```

- b. **Bag Statements** - check if the value of a variable is in *item* (case) labels, if not the *potion* (default) statement is executed.

### Rules for Bag Statements

1. Bag statements must always start with the reserved word *bag* (switch), followed by parentheses that enclose an expression or an identifier, then a pair of curly brackets ({} ) that encloses the bag's body.
2. The bag statement can be null, no item inside.
3. The *item*, *potion*, and *quit* (break) are optional.
4. The expression is evaluated then compared with the values of each item label. If there is a match, the corresponding code after the matching label will be executed until a *quit* statement is encountered; else, the *potion* block, if any, is executed.
5. Potion case can only appear at the end of the bag statement.
6. Quit is not needed in the potion statement.

#### Syntax

```
bag ( <expression> ) {  
    item <literal> :  
        <statements>  
        quit;  
    potion:  
        <statements>  
}
```

#### Example

```
bag ( weapon ) {  
    item "sword" :  
        say("Sword equipped");  
        quit;  
    item "bow":  
        say("Bow Equipped");  
        quit;  
    item "arrow":  
        say("Arrow Equipped");  
        quit;  
    potion:  
        say("Health potion used");  
}
```

5. **Looping Statements** - execute one or more statements repeatedly until some condition is met.

- a. **Sprint Statements** – execute one or more statements for a specific number of times.

### Rules for Sprint Statements

1. It must always begin with the reserved word `sprint`, then parentheses enclosing the three statements (*initialization*, *condition*, and *iterator*) that are separated by semicolons, and a pair of curly brackets (`{}`) enclosing the body.
2. Putting statements on *initialization*, *condition*, and *iterator* are not required as long as the semicolons are present.
3. Only variable initialization and assignment are allowed in the *initialization* part of the sprint loop.
4. Only one variable can be initialized or assigned to the *initialization* of sprint statements.
5. Only assignment or unary expression can be used in the *iteration* of sprint statements.
6. If the condition is evaluated as *heads* (true), the following will happen:
  - Statement/s inside the sprint statement are executed.
  - Iterator is executed/evaluated.
  - Conditions will be evaluated again.
  - This keeps iterating until the condition is evaluated as false.
7. If the condition is evaluated as false, the sprint loop terminates.
8. Nested sprint loop is allowed.

#### Syntax

```
sprint( <initialization>; <condition>; <iteration> ){
    <statements>
}
```

where:

- initialization = initializes variable
- condition = if true, the body of sprint loop is executed; else, the loop is terminated
- iterator = updates the value of initialized variables and checks the condition again

#### Example

```
sprint( clock rageMode = 100; rageMode > 0 ;{
    punch ( rageMode == 50){
        say("Going back to normal: strength and speed decreased");
        rageMode--;
    }
}
```

- b. **Flight Statements** - continuously loops until the condition is *tails* (false).

### Rules for Flight Statements

1. It must always start with the reserved word *flight*, followed by parentheses that encloses the condition, then a pair of curly brackets ({}).
2. The body of flight statements must always be enclosed by curly brackets ({}).
3. If the condition is evaluated as *heads* (true), the following will happen respectively:
  - Statement/s inside the applause statement are executed.
  - After the execution, the condition is evaluated again.
4. If the condition is evaluated as false, the flight loop terminates.
5. Nested flight loop is allowed.

#### Syntax

```
flight( <condition> ) {  
    <statements>  
}
```

#### Example

```
flight( rageMode > 0 ) {  
  
    punch ( rageMode == 50){  
        say("Going back to normal: strength and speed decreased");  
    }  
    rageMode--;  
}
```

- c. **Take...Flight Statements** - execute at least one regardless of the condition.

### Rules for Take...Flight Statements

1. Take..Flight statements always start with the reserved word *take*, followed by curly brackets ({} that encloses its body, then the flight's head.
2. The body of take..flight statements must always be enclosed by curly brackets ({}).
3. The body is executed first then condition/s is evaluated. If the condition is true, the body is executed; else, the take...flight loop terminates.
4. Nested take...flight loop is allowed.

## Syntax

```
take{  
    <statements>  
} flight( <condition > );
```

## Example

```
take{  
  
    punch ( rageMode == 50){  
        say("Going back to normal: strength and speed decreased");  
    }  
  
    rageMode--;  
  
} flight( rageMode > 0 );
```

- d. **Quit Statements** - statements that terminate a loop.

### Rules for Quit Statements

1. It always starts with the reserved word *quit*, followed by a semicolon.
2. It can be used inside a loop or within bag statements.

## Syntax

```
quit;
```

## Example

```
sprint( clock healthBar = 50; healthBar > 0 ; healthBar--){  
  
    punch (healthBar <= 10){  
        say("Your health bar is too low, you can't enter here.");  
        quit;  
    }  
  
}
```

- e. **Skip Statements** - statements that skip the current iteration.

### Rules for Skip Statements

1. It always starts with the reserved word *skip*, followed by a semicolon.
2. It can only be used inside a loop.

#### Syntax

```
skip;
```

#### Example

```
sprint( clock healthBar = 50; healthBar > 0 ; healthBar--){

    punch (healthBar <= 10){
        say("Your health bar is too low, you can't enter here yet. ");
        quit;
    } kick (healthBar >= 11) {
        skip;
    }

}
```

## I. FUNCTIONS

Function is a block of code that performs a set of instructions

### Rules for Function Creation

1. All user-defined functions must be placed between *START* and *PAUSE()* function.
2. When defining a function, it should start with the reserved word *quest*, followed by its identifier, then a pair of parentheses (()).
3. Parameters can be placed between the parentheses.
4. Parameter list must obey the same declaration rules applied to variable declaration.
5. Parameters are divided by comma (,).
6. Parameters are optional.
7. The body of a function must always be enclosed by curly brackets ({}).
8. The body of a function can be null.
9. Functions can have zero or more return statements.

#### Syntax

```
quest <identifier> ( <parameters> ){  
    <statement/s>  
}  
  
where: parameters = <data type> <identifier>
```

#### Example

```
quest buyItem(clock money){  
  
    punch ( money >= itemPrice){  
        say(itemName, " obtained.");  
    }  
  
}
```

### Rules for Calling a Function

1. Functions are called by its identifier followed by parentheses (()).
2. Value/s or argument/s can be passed to a function, as long as it matches the number, order and data type of the parameters of said function.

## Syntax

```
<identifier> ( <value> )
```

where: value =   
 <identifier>  
 <literal>  
 <expression>

## Example

START

```
quest buyItem(clock money, rope playerName){
```

```
    punch ( money >= itemPrice){  
        say(itemName, " obtained.");  
    }
```

```
}
```

```
PAUSE():
```

```
    clock money= 1000;
```

```
    buyItem(money, "Monkey King" );
```

PLAY

END

## J. ARRAY

An array is a data structure that stores a fixed-size collection of elements of the same data type.

### Rules for Array Declaration

1. Arrays must begin with a specified data type (clock, hover, rune, rope, and coin), followed by an identifier, and the array size enclosed by square brackets ([]).
2. The array size must always be a positive whole number.
3. Array can only store literals with the same type.
4. Changing the size of an array once it is declared is not allowed.
5. The declaration of multiple arrays in one statement is allowed.
6. The declaration of a multidimensional array is allowed.
7. The maximum dimension of an array is 3.
8. The initialization of the value of an array can be done in its declaration.
9. The number of initialized values can be less than the size of the array but not greater than it.

### Rules in Using Array

1. An array always begins with index 0.
2. When assigning a value to an element, the index must be specified.
3. The last index of the array is always one (1) less than its defined length.
4. A whole array is not allowed to be passed into a function.

#### Syntax

```
<data type> <identifier> [<array size>];  
<data type> <identifier> [<array size>]= { <values> };  
<data type> <identifier> [<array size>][<array size>] = { { <values> }, { <values> } };  
  
where: array size = <identifier>  
           <clock literal>  
  
value = <identifier>  
           <literal>
```

#### Example

```
hover extraLevel[9] = { 3.5, 9.5, 10.5};  
rope bosses[2] = { "Draco", "Big Mama", "Elder Dragon" };
```

## K. TOWER

A *tower* (struct or structure) is a user-defined data type that has a structure that stores a fixed-size collection of elements of different data types.

### Rules for Tower Declaration

1. The declaration of *tower* must always begin with the reserved word *tower*, followed by its identifier and is preceded with curly brackets ({} ) in order to indicate the body of the *tower*, then a semicolon (;).
2. Tower declaration can only be done globally.
3. All tower declarations must have at least one element.
4. Each element must only have its data type and identifier. It cannot be initialized.
5. All elements are placed inside the body of the tower.
6. The declaration of multiple tower's elements with the same data type is allowed. To separate multiple elements, use comma (,).
7. The declaration of multiple tower's elements with different data types in a single line is allowed. To separate multiple elements, use semicolon (;).
8. Array declaration is allowed inside the body of the tower.
9. A structure cannot contain another structures, no nested *tower* (structure).
10. A tower member cannot be constant.

#### Syntax

```
tower <identifier> {  
    <variable declaration>  
    .  
    .  
    .  
    <variable declaration>  
};
```

#### Example

```
tower Player {  
    clock id, age;  
    rope name;  
    clock randDMG[10];  
};
```

## Rules for Tower Variables Declaration

1. Declaration of tower variables must start with the reserved word *tower*, followed by a user-defined data type, then an identifier.
2. Initialization of variables is not allowed.

### Syntax

```
<identifier> <identifier>;
```

### Example

START

```
  tower Player {  
    clock id, age;  
    rope name;  
    clock randDMG[10];  
  };
```

PAUSE():

Player player[5];

PLAY

END

## Rules for Accessing Members of a Tower

1. Towers are called through its element's identifier, followed by an at-symbol (@) then the identifier of the structure element.

### Syntax

```
<identifier>@<element>
```

### Example

player[0]@id

## Rules for Passing Tower to a Function

1. Towers can be used as a parameter of a function by using the data type of the element as its tower id.

### Syntax

```
quest <data type> <identifier> ( <tower identifier> <identifier> ){  
}  
}
```

### Example

```
tower Player {  
    clock id;  
    rope name;  
};  
  
quest checkInfo (Player playerInfo){  
    Player playerInfo;  
    playerInfo@id = 201812345;  
  
    say ("Your id is", playerInfo@id);  
}  
  
PAUSE():  
    Player playerInfo;  
    checkInfo(playerInfo);
```

## L. COMMENT

A comment is a set of text in a program that is not executed by the program.

### Rules for Comments

1. Single-line comments must be preceded by a dollar sign (\$).
2. Multi-line comments must start with an open angle bracket and forward slash (</), then end with angle close bracket and forward slash (/>).
3. Nested multi-line comments are not allowed.

Syntax: Single-line Comment

```
$ <comments>
```

Syntax: Multi-line Comment

```
</ <comments> />
```

### Example

```
$ This is a single-line comment
```

```
</
```

```
.
```

```
.
```

```
This is a multi-line comment
```

```
.
```

```
.
```

```
.
```

```
/>
```

## M. ESCAPE SEQUENCES

Escape sequences are a special set of characters that act as a single character to represent an escaped character.

### Rules for Escape Sequences

1. It always starts with the backslash symbol (\), followed by a special character.
2. An escape sequence can only contain two (2) characters and can only be used with *on rope* (string) or *rune* (char) literals.

Escape Sequence	Description
\n	Include new line in a string
\t	Include horizontal tab in a string
\'	Include single quote in a string
\"	Include double quote in a string
\~	Include tilde in a string

## VI. RESERVED WORDS

C++ Language	Proposed Language	Description
&&	AND	Returns true if both statements are heads
	END	
!	NOT	It is used to reverse the logical state of its operand
	OR	Executes a statement if at least one of the operands is heads
main	PAUSE	The function where the program starts its execution
	PLAY	
	START	
const	STOCK	Used to define constant values
switch	bag	A control statement that allows a value to execute one of many blocks
int	clock	A data type that represents integers
boolean	coin	A data type that can only represent heads or tails
while	flight	Executes one or more statements as long as a specified condition is true
1	heads	Boolean value for true
float	hover	A data type that represents decimal numbers
case	item	A section of code inside the bag statement that gets executed when it is equal to the bag's value
else if	kick	Provides another set of conditions to be examined if the first offer statement is proven to be tails
cin	listen	Used for getting user input
default	potion	A block of code inside the bag statement that gets executed when all the choices are not equal to the bag's value
if	punch	Executes a set of statements if the condition is heads
function	quest	A block of code which only runs when it is called
break	quit	Terminates the loop
return	return	Returns a value or simply passes the control to the calling quest

string	rope	A data type that represents a sequence of characters
char	rune	A data type that holds one character
cout	say	Used to output values or print text
continue	skip	Forces the next iteration of the loop to take place, skipping any code in between
for	sprint	Executes one or more statements for a specific number of times
0	tails	Boolean value for false
do	take	Used in take...flight to execute the loop once before the condition is evaluated
struct	tower	Used to make user-defined data types and combine data items of different types
else	ultimate	Specifies a block of code to be executed if the condition is tails

## VII. RESERVED SYMBOLS

### A. Arithmetic Operator

Operator	Description	Example
+	Performs addition	25 + 5 will give 30
-	Performs subtraction	25 - 5 will give 20
*	Performs multiplication	25 * 5 will give 125
/	Performs division	25 / 5 will give 5
%	Yields the remainder after division	25 % 5 will give 0

### B. Assignment Operator

Operator	Description	Example
=	Assignment operator	x = 5
+=	Add and assignment operation	x += 5
-=	Subtract and assignment operation	x -= 5
*=	Multiply and assignment operation	x *= 5
/=	Divide and assignment operation	x /= 5
%=	Modulo and assignment operation	x %= 5

### C. Relational Operator

Operator	Description	Example
==	Tests whether or not the values of two operands are equal or not. Returns heads if equal, otherwise it returns tails	(10 == 11) is Tails
!=	Tests whether or not the values of two operands are equal or not. If not equal it returns heads, otherwise it returns tails	(10 != 11) is Heads
<	Checks if the left operand has a lesser value than the right operand. If the left	(10 < 11) is Tails

	operand is lesser, it returns true	
>	Checks if the left operand has a greater value than the right operand. If the left operand is greater, it returns true	(10 > 11) is Heads
<=	Checks if the left operand is less than or equal to the right operand. If the left operand is less than or equal to the right operand, it returns true	(23 <= 26) is Heads
>=	Checks if the left operand is greater than or equal to the right operand. If the left operand is greater than or equal to the right operand, it returns true	(32 >= 32) is Heads

## D. Unary Operator

Operator	Description	Example
++	Increment a value by one	10++ will give 11
--	Decrement a value by one	10-- will give 9

## E. Other Symbols

Operator	Description
,	Separator
.	Decimal Point
:	Used in a bag statement
~	Used to define a negative number
‘ ’	Used to define a rune
“ ”	Used to define a rope
@	Used to access elements in tower
()	Used to group operations, conditions, parameters, and arguments
[]	Used in declaring arrays
{ }	Used in code blocking
;	Used to terminate a line
\'	An escape sequence used to display the single quotation character

\"	An escape sequence used to display the double quotation character
\t	An escape sequence used to shift the cursor a few spaces to the right on the same line
\n	An escape sequence used to shift the cursor to the new line
\\"	An escape sequence used to display the backslash character
\$	Used to indicate a single-line comment
</>	Used to indicate the start and end of a comment

## VIII. REGULAR EXPRESSION

### A. RESERVED WORDS

C++ Language	Proposed Language	Expression	Token
&&	AND	(A)(N)(D)	AND
	END	(E)(N)(D)	END
!	NOT	(N)(O)(T)	NOT
	OR	(O)(R)	OR
main	PAUSE	(P)(A)(U)(S)(E)	PAUSE
	PLAY	(P)(L)(A)(Y)	PLAY
	START	(S)(T)(A)(R)(T)	START
const	STOCK	(S)(T)(O)(C)(K)	STOCK
case	bag	(b)(a)(g)	bag
int	clock	(c)(l)(o)(c)(k)	clock
boolean	coin	(c)(o)(i)(n)	coin
while	flight	(f)(l)(i)(g)(h)(t)	flight
1	heads	(h)(e)(a)(d)(s)	heads
float	hover	(h)(o)(v)(e)(r)	hover
switch	item	(i)(t)(e)(m)	item
else if	kick	(k)(i)(c)(k)	kick
cin	listen	(l)(i)(s)(t)(e)(n)	listen
default	potion	(p)(o)(t)(i)(o)(n)	potion
if	punch	(p)(u)(n)(c)(h)	punch
function	quest	(q)(u)(e)(s)(t)	quest
break	quit	(q)(u)(i)(t)	quit
return	return	(r)(e)(t)(u)(r)(n)	return
string	rope	(r)(o)(p)(e)	rope
char	rune	(r)(u)(n)(e)	rune
cout	say	(s)(a)(y)	say
continue	skip	(s)(k)(i)(p)	skip
for	sprint	(s)(p)(r)(i)(n)(t)	sprint
0	tails	(t)(a)(i)(l)(s)	tails
do	take	(t)(a)(k)(e)	take
struct	tower	(t)(o)(w)(e)(r)	tower
else	ultimate	(u)(l)(t)(i)(m)(a)(t)(e)	ultimate

## B. RESERVED SYMBOLS

Reserved Symbols	Regular Expression	Token
(	((()	(
)	())	)
[	([)	[
]	(])	]
{	({{})	{
}	(}})	}
+	(+)	+
-	(-)	-
*	(*)	*
/	(/)	/
	(/)(>)	
%	(%)	%
=	(=)	=
+ =	(+)(=)	+ =
- =	(-)(=)	- =
* =	(*)(=)	* =
/ =	(/)(=)	/ =
% =	(%)(=)	% =
==	(=)(=)	==
!=	(!)(=)	!=
>	(>)	>
> =	(>)(=)	> =
<	(<)	<
< /	(<)(/)	< /
< =	(<)(=)	< =
++	(+)(+)	++
--	(-)(-)	--
.	(.)	.
,	(,)	,
@	(@)	@
\$	(\$)	\$
~	(~)	~
\	(\ )	\
:	(;)	;
:	(:)	:
“	(“)	“
‘	(‘)	‘

## C. LITERALS AND COMMENT

Literals and Comment	Regular Expression	Token
clock literal	( ~ / λ ) (nonzero) (number / λ) <sup>9</sup>	clocklit
hover literal	( ~ / λ ) (nonzero) (number / λ) <sup>9</sup> ( . ) (number / λ) <sup>5</sup>	hoverlit
rune literal	( ' ) ( ascii / λ ) ( ' )	runelit
rope literal	( " )( ascii / λ ) ( " )	ropelit
coin literal	( (h) (e) (a) (d) (s) / ((t) (a) (i) (l) (s)) )	coinlit
Comment	( </ \$ ) (ascii / λ) ( /> / λ)	comment

## D. IDENTIFIER

Identifier	Regular Expression	Token
Identifier	(alpha) (alphanumeric/ λ) <sup>24</sup>	id

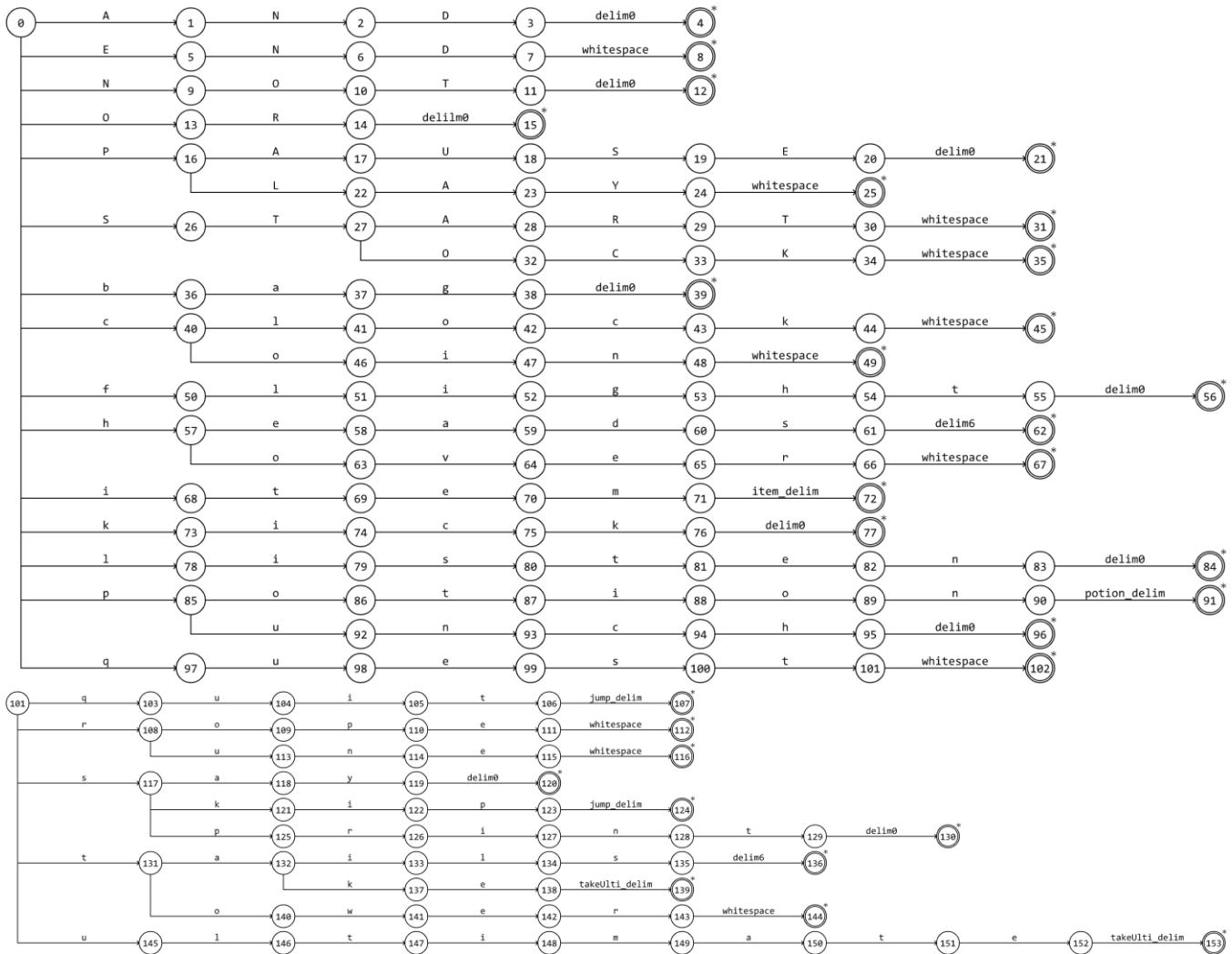
## IX. REGULAR DEFINITION

<b>Delimiter</b>	<b>Value</b>
nonzero	1, 2, 3, 4, 5, 6, 7, 8, 9,
numbers	0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
letters	A-z
lowercase	a-z
alphanumeric	numbers, letters
nonzero_alphanumeric	nonzero, letters
ascii	any printable ascii character
whitespace	space, \t, \n
quote	", '
arithmetic	"+", -, *, %, /
angle_brackets	>, <
not_equal	!, =
esc_seq	t, n, quote, \
colons	:, ;
colon_equal	;, =
ccb_comma	, ,
delim0	whitespace, (
delim1	alphanumeric, delim0
delim2	whitespace, )
delim3	delim1, ~
delim4	delim1, =
delim5	delim3, quote
delim6	delim2, colons, ccb_comma, not_equal
delim7	angle_brackets, arithmetic
delim8	delim1, ;
item_delim	whitespace, ~, quote
return_delim	delim0, quote, ~
unary_delim	delim2, ;
potion_delim	whitespace, :
jump_delim	whitespace, ;
takeUlti_delim	whitespace, {
colon_delim	alphanumeric, whitespace
tilde_delim	nonzero_alphanumeric, delim0, ~
op_delim	delim5, ), ;
cp_delim	delim2, {, }, !, ,, delim7, =, colons
ocb_delim	alphanumeric, whitespace, {, }, ~, (
ccb_delim	}, ,, jump_delim
osb_delim	alphanumeric, whitespace, [

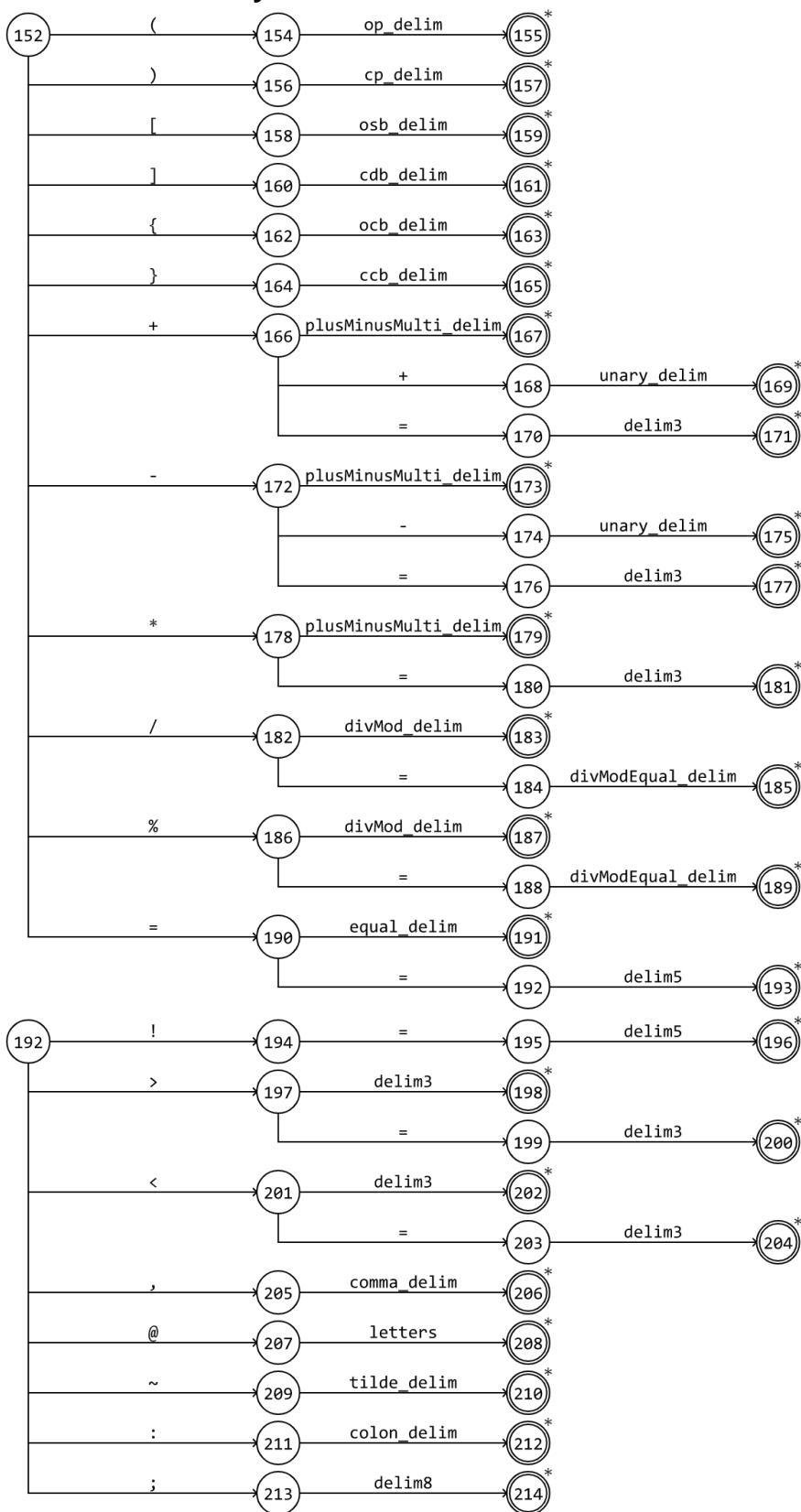
csb_delim	;, }, @, [, not_equal, whitespace, arithmetic, angle_brackets, delim2, ccb_comma
plusMinusMulti_delim	delim0, alphanumeric, ~
divMod_delim	delim0, nonzero_alphanumeric, ~
comma_delim	delim1, {, quote, ~
identifier_delim	tilde_delim
clock_delim	alphanumeric, quote, takeUlti_delim
hover_delim	delim2 (, [ , ], {, ccb_comma, delim7, not_equal, @, ;
escape	n, t, quote
newline	\n

# X. TRANSITION DIAGRAM

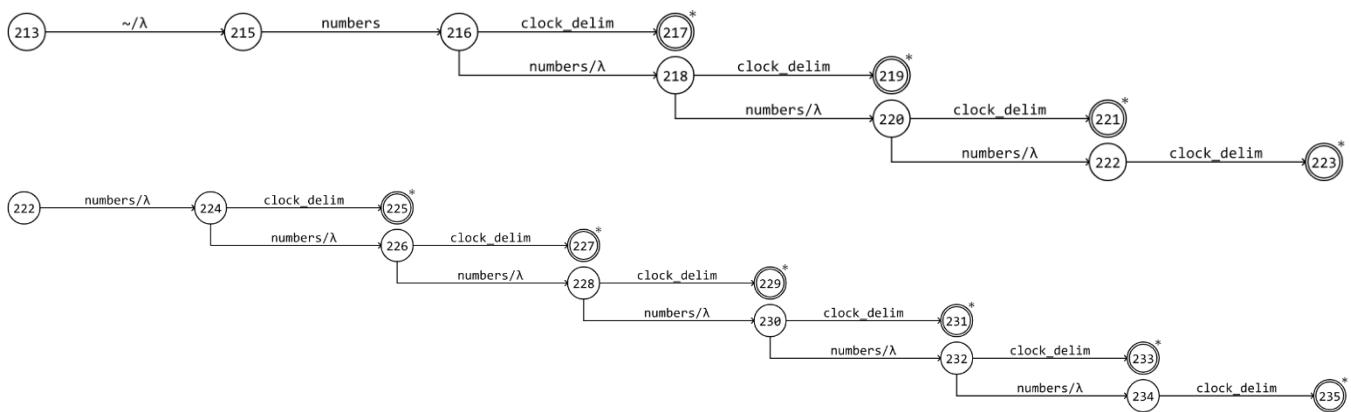
## A. Reserved Words



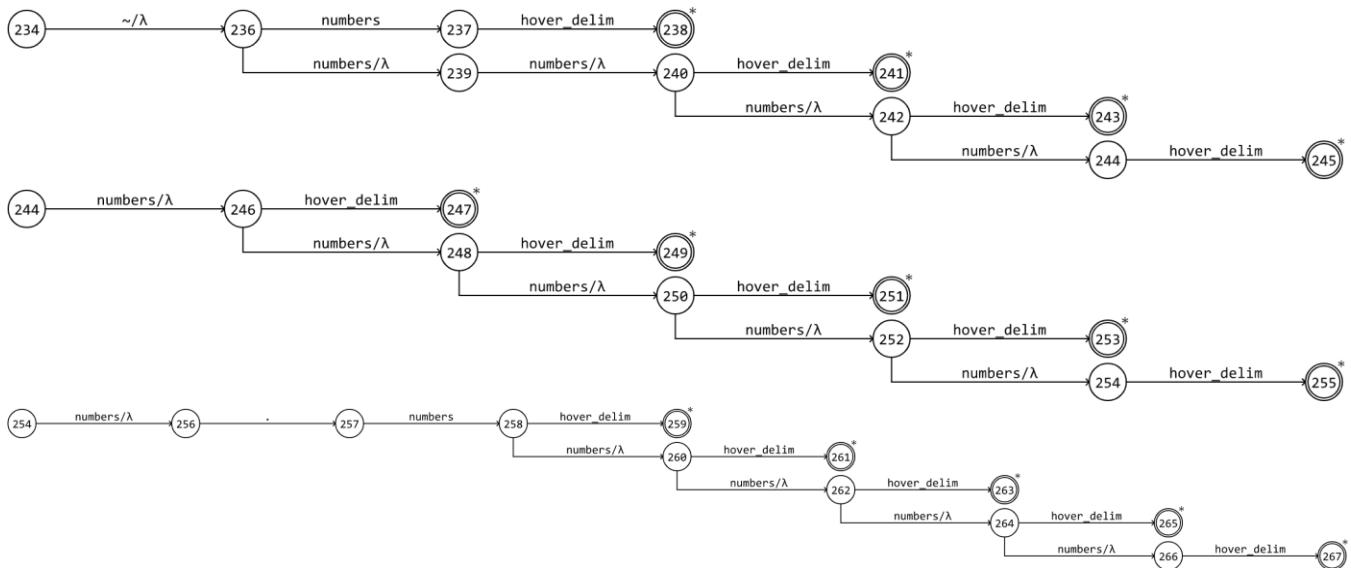
## B. Reserved Symbols



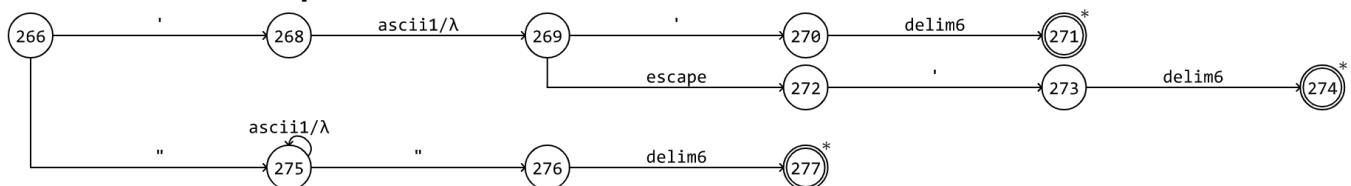
### C. Clock Literal



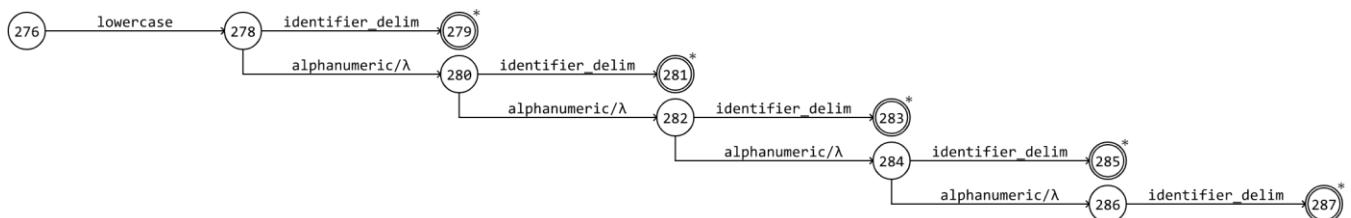
### D. Hover Literal



### E. Rune and Rope Literal

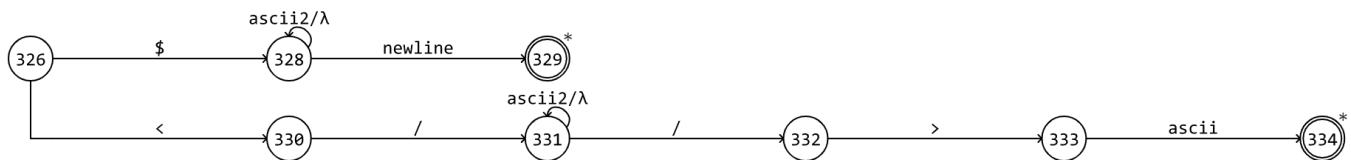


### F. Identifiers





## G. Comments



## XI. CONTEXT-FREE GRAMMAR

#	Production	Production Rule
1	<program>	START <global> PAUSE () : <statements> PLAY END
2	<global>	<global_statement> <global>
3	<global>	$\lambda$
4	<global_statement>	STOCK <type> id <stock_dec> <morestock_dec> ;
5	<global_statement>	tower id { <type> id <array1D_ind> <moretowerMem> ; <moretowerMem_dec> } ;
6	<global_statement>	<type> id <vardec_opt> <morevardec_opt> ;
7	<global_statement>	quest id ( <func_param> ) { <statements> }
8	<type>	clock
9	<type>	hover
10	<type>	coin
11	<type>	rune
12	<type>	rope
13	<literals>	clocklit
14	<literals>	hoverlit
15	<literals>	coinlit
16	<literals>	runelit
17	<literals>	ropelit
18	<assign_oprtr>	=
19	<assign_oprtr>	+=
20	<assign_oprtr>	-=
21	<assign_oprtr>	*=
22	<assign_oprtr>	/=
23	<assign_oprtr>	%=
24	<notAssign_operators>	+
25	<notAssign_operators>	-
26	<notAssign_operators>	*
27	<notAssign_operators>	/
28	<notAssign_operators>	%
29	<notAssign_operators>	AND
30	<notAssign_operators>	OR
31	<notAssign_operators>	<
32	<notAssign_operators>	>
33	<notAssign_operators>	>=
34	<notAssign_operators>	<=
35	<notAssign_operators>	==
36	<notAssign_operators>	!=
37	<unary_oprtr>	++
38	<unary_oprtr>	--
39	<unary_oprtr>	$\lambda$
40	<expr>	<expression> <exprs>

41	<expr>	$\lambda$
42	<exprs>	, <expression> <exprs>
43	<exprs>	$\lambda$
44	<morestock_dec>	, id <stock_dec> <morestock_dec>
45	<morestock_dec>	$\lambda$
46	<stock_dec>	= <expression>
47	<stock_dec>	[ <expression> ] <stock_arrayOpt1>
48	<stock_arrayOpt1>	= <stock_array1D>
49	<stock_arrayOpt1>	[ <expression> ] <stock_arrayOpt2>
50	<stock_arrayOpt2>	= <stock_array2D>
51	<stock_arrayOpt2>	[ <expression> ] = <stock_array3D>
52	<stock_array1D>	{ <expr> }
53	<stock_array2D>	{ <stock_array1D> <morestock_Arr1D> }
54	<stock_array3D>	{ <stock_array2D> <morestock_Arr2D> }
55	<morestock_Arr1D>	, <stock_array1D> <morestock_Arr1D>
56	<morestock_Arr1D>	$\lambda$
57	<morestock_Arr2D>	, <stock_array2D> <morestock_Arr2D>
58	<morestock_Arr2D>	$\lambda$
59	<moretowerMem>	, id <array1D_ind> <moretowerMem>
60	<moretowerMem>	$\lambda$
61	<moretowerMem_dec>	<type> id <array1D_ind> <moretowerMem> ; <moretowerMem_dec>
62	<moretowerMem_dec>	$\lambda$
63	<array1D_ind>	[ <expression> ] <array2D_ind>
64	<array1D_ind>	$\lambda$
65	<array2D_ind>	[ <expression> ] <array3D_ind>
66	<array2D_ind>	$\lambda$
67	<array3D_ind>	[ <expression> ]
68	<array3D_ind>	$\lambda$
69	<expression>	<literals> <expr_ext>
70	<expression>	id <id_suffix> <expr_ext>
71	<expression>	( <expression> ) <expr_ext>
72	<expression>	NOT <expression>
73	<expression>	$\sim$ <expression>
74	<expr_ext>	<notAssign_operators> <expression>
75	<expr_ext>	$\lambda$
76	<id_suffix>	( <expr> )
77	<id_suffix>	<array1D_ind> <id_at> <unary_oprtr>
78	<id_at>	@ id <array1D_ind>
79	<id_at>	$\lambda$
80	<vardec_opt>	= <expression>
81	<vardec_opt>	[ <expression> ] <array1D>
82	<vardec_opt>	$\lambda$
83	<morevardec_opt>	, id <vardec_opt> <morevardec_opt>
84	<morevardec_opt>	$\lambda$
85	<array1D>	[ <expression> ] <array2D>
86	<array1D>	= <stock_array1D>
87	<array1D>	$\lambda$
88	<array2D>	[ <expression> ] <array3D>

89	<array2D>	= <stock_array2D>
90	<array2D>	λ
91	<array3D>	= <stock_array3D>
92	<array3D>	λ
93	<func_param>	<parameter> <parameters>
94	<func_param>	λ
95	<parameter>	<type> id
96	<parameter>	id id
97	<parameters>	, <parameter> <parameters>
98	<parameters>	λ
99	<statements>	<statement> <statements>
100	<statements>	λ
101	<statement>	STOCK <type> id <stock_dec> <morestock_dec> ;
102	<statement>	<type> id <vardec_opt> <morevardec_opt> ;
103	<statement>	id <startswith_idNext> ;
104	<statement>	sprint ( <initialization> ; <expression_null> ; <iteration> ) {<statements> }
105	<statement>	quit ;
106	<statement>	skip ;
107	<statement>	bag ( <expression> ) { <item_statement> <potion_statement> }
108	<statement>	flight ( <expression> ) { <statements> } ;
109	<statement>	take { <statements> } flight ( <expression> ) ;
110	<statement>	punch ( <expression> ) { <statements> } <kick_statement><ultimate_statements>
111	<statement>	say ( <expression> <exprs> ) ;
112	<statement>	listen ( id <array1D_ind> <id_at> ) ;
113	<statement>	return <expression_null> ;
114	<startswith_idNext>	id <array1D_ind> <moretowerMem>
115	<startswith_idNext>	( <expr> )
116	<startswith_idNext>	<array1D_ind> <id_at> <iteration_suffix>
117	<initialization>	<type> id = <expression>
118	<initialization>	id <array1D_ind> <id_at> <assign_oprtr> <expression>
119	<initialization>	λ
120	<expression_null>	<expression>
121	<expression_null>	λ
122	<iteration>	id <array1D_ind> <id_at> <iteration_suffix>
123	<iteration>	λ
124	<iteration_suffix>	++
125	<iteration_suffix>	-
126	<iteration_suffix>	<assign_oprtr> <expression>
127	<kick_statement>	kick ( <expression> ) { <statements> } <kick_statement>
128	<kick_statement>	λ
129	<ultimate_statement>	ultimate { <statements> }
130	<ultimate_statement>	λ
131	<item_statement>	item <literals> : <statements> <item_statement>
132	<item_statement>	λ
133	<potion_statement>	potion : <statements>
134	<potion_statement>	λ

## XII. FIRST SET

#	Production	Production Set
1	<program>	{ START }
2	<global>	{ STOCK, tower, quest, clock, hover, coin, rune, rope, λ }
3	<global_statement>	{ STOCK, tower, quest, clock, hover, coin, rune, rope }
4	<type>	{ clock, hover, coin, rune, rope }
5	<literals>	{ clocklit, hoverlit, coinlit, runelit, ropelit }
6	<assign_oprtr>	{ =, +=, -=, *=, /=, %= }
7	<notAssign_operators>	{ +, -, *, /, %, AND, OR, <, >, >=, <=, ==, != }
8	<unary_oprtr>	{ ++, --, λ }
9	<expr>	{ (, id, clocklit, hoverlit, coinlit, runelit, ropelit, NOT, ~, λ }
10	<exprs>	{ ,, λ }
11	<morestock_dec>	{ ,, λ }
12	<stock_dec>	{ =, [] }
13	<stock_arrayOpt1>	{ =, [] }
14	<stock_arrayOpt2>	{ =, [] }
15	<stock_array1D>	{ {} }
16	<stock_array2D>	{ {} }
17	<stock_array3D>	{ {} }
18	<morestock_Arr1D>	{ ,, λ }
19	<morestock_Arr2D>	{ ,, λ }
20	<moretowerMem>	{ ,, λ }
21	<moretowerMem_dec>	{ clock, hover, coin, rune, rope, λ }
22	<array1D_ind>	{ [ , λ ] }
23	<array2D_ind>	{ [ , λ ] }
24	<array3D_ind>	{ [ , λ ] }
25	<expression>	{ (, id, clocklit, hoverlit, coinlit, runelit, ropelit, NOT, ~ ) }
26	<expr_ext>	{ +, -, *, /, %, AND, OR, <, >, >=, <=, ==, !=, λ }
27	<id_suffix>	{ (, ++, --, [ , @, λ ] }
28	<id_at>	{ @, λ }
29	<vardec_opt>	{ =, [ , λ ] }
30	<morevardec_opt>	{ ,, λ }
31	<array1D>	{ =, [ , λ ] }
32	<array2D>	{ =, [ , λ ] }
33	<array3D>	{ =, λ }
34	<func_param>	{ id, clock, hover, coin, rune, rope, λ }
35	<parameter>	{ id, clock, hover, coin, rune, rope }
36	<parameters>	{ ,, λ }
37	<statements>	{ STOCK, id, clock, hover, coin, rune, rope, sprint, quit, skip, bag, flight, take, punch, say, listen, return, λ }
38	<statement>	{ STOCK, id, clock, hover, coin, rune, rope, sprint, quit, skip, bag, flight, take, punch, say, listen, return }
39	<startswith_idNext>	{ (, id, =, +=, -=, *=, /=, %=, ++, [ , @, - ] }
40	<initialization>	{ id, clock, hover, coin, rune, rope, λ }
41	<expression_null>	{ (, id, clocklit, hoverlit, coinlit, runelit, ropelit, NOT, ~, λ ) }
42	<iteration>	{ id, λ }
43	<iteration_suffix>	{ =, +=, -=, *=, /=, %=, ++, - }

44	<kick_statement>	{ kick, λ }
45	<ultimate_statement>	{ ultimate, λ }
46	<item_statement>	{ item, λ }
47	<potion_statement>	{ potion, λ }

### XIII. FOLLOW SET

#	Production	Production Set
1	<program>	{ λ }
2	<global>	{ PAUSE }
3	<global_statement>	{ PAUSE, STOCK, tower, quest, clock, hover, coin, rune, rope }
4	<type>	{ id }
5	<literals>	{ (), :, ;, }, +, -, *, /, %, AND, OR, <, >, >=, <=, ==, !=, ,, ] }
6	<assign_oprtr>	{ (, id, clocklit, hoverlit, coinlit, runelit, ropelit, NOT, ~ }
7	<notAssign_operators>	{ (, id, clocklit, hoverlit, coinlit, runelit, ropelit, NOT, ~ }
8	<unary_oprtr>	{ (), :, }, +, -, *, /, %, AND, OR, <, >, >=, <=, ==, !=, ,, ] }
9	<expr>	{ (), {} }
10	<exprs>	{ (), {} }
11	<morestock_dec>	{ ; }
12	<stock_dec>	{ ;, , }
13	<stock_arrayOpt1>	{ ;, , }
14	<stock_arrayOpt2>	{ ;, , }
15	<stock_array1D>	{ ;, , , }
16	<stock_array2D>	{ ;, , , }
17	<stock_array3D>	{ ;, , , }
18	<morestock_Arr1D>	{ {}, {} }
19	<morestock_Arr2D>	{ {}, {} }
20	<moretowerMem>	{ ; }
21	<moretowerMem_dec>	{ {}, {} }
22	<array1D_ind>	{ (), :, }, =, +=, -=, *=, /=, %=, +, -, *, /, %, AND, OR, <, >, >=, <=, ==, !=, ++, --, ,, ], @, - }
23	<array2D_ind>	{ (), :, }, =, +=, -=, *=, /=, %=, +, -, *, /, %, AND, OR, <, >, >=, <=, ==, !=, ++, --, ,, ], @, - }
24	<array3D_ind>	{ (), :, }, =, +=, -=, *=, /=, %=, +, -, *, /, %, AND, OR, <, >, >=, <=, ==, !=, ++, --, ,, ], @, - }
25	<expression>	{ (), :, {}, ,, ] }
26	<expr_ext>	{ (), :, {}, ,, ] }
27	<id_suffix>	{ (), :, }, +, -, *, /, %, AND, OR, <, >, >=, <=, ==, !=, ,, ] }
28	<id_at>	{ (), :, }, =, +=, -=, *=, /=, %=, +, -, *, /, %, AND, OR, <, >, >=, <=, ==, !=, ++, --, ,, ], - }
29	<vardec_opt>	{ ;, , }
30	<morevardec_opt>	{ ; }
31	<array1D>	{ ;, , }
32	<array2D>	{ ;, , }
33	<array3D>	{ ;, , }
34	<func_param>	{ () }
35	<parameter>	{ (), , }
36	<parameters>	{ () }

37	<statements>	{ PLAY, }, item, potion }
38	<statement>	{ PLAY, STOCK, id, }, clock, hover, coin, rune, rope, sprint, quit, skip, bag, flight, take, punch, say, listen, return, item, potion }
39	<startswith_idNext>	{ ; }
40	<initialization>	{ ; }
41	<expression_null>	{ ; }
42	<iteration>	{ () }
43	<iteration_suffix>	{ () ; }
44	<kick_statement>	{ PLAY, STOCK, id, }, clock, hover, coin, rune, rope, sprint, quit, skip, bag, flight, take, punch, say, listen, return, ultimate, item, potion }
45	<ultimate_statement>	{ PLAY, STOCK, id, }, clock, hover, coin, rune, rope, sprint, quit, skip, bag, flight, take, punch, say, listen, return, item, potion }
46	<item_statement>	{ }, potion }
47	<potion_statement>	{ () }

## XIV. PREDICT SET

#	Production	Production Set
1	<program>	{ START }
2	<global>	{ STOCK, tower, quest, clock, hover, coin, rune, rope }
3	<global>	{ PAUSE }
4	<global_statement>	{ STOCK }
5	<global_statement>	{ tower }
6	<global_statement>	{ clock, hover, coin, rune, rope }
7	<global_statement>	{ quest }
8	<type>	{ clock }
9	<type>	{ hover }
10	<type>	{ coin }
11	<type>	{ rune }
12	<type>	{ rope }
13	<literals>	{ clocklit }
14	<literals>	{ hoverlit }
15	<literals>	{ coinlit }
16	<literals>	{ runelit }
17	<literals>	{ ropelit }
18	<assign_oprtr>	{ = }
19	<assign_oprtr>	{ += }
20	<assign_oprtr>	{ -= }
21	<assign_oprtr>	{ *= }
22	<assign_oprtr>	{ /= }
23	<assign_oprtr>	{ %= }
24	<notAssign_operators>	{ + }
25	<notAssign_operators>	{ - }
26	<notAssign_operators>	{ * }
27	<notAssign_operators>	{ / }
28	<notAssign_operators>	{ % }
29	<notAssign_operators>	{ AND }
30	<notAssign_operators>	{ OR }
31	<notAssign_operators>	{ < }
32	<notAssign_operators>	{ > }
33	<notAssign_operators>	{ >= }
34	<notAssign_operators>	{ <= }
35	<notAssign_operators>	{ == }
36	<notAssign_operators>	{ != }
37	<unary_oprtr>	{ ++ }
38	<unary_oprtr>	{ -- }
39	<unary_oprtr>	{ ), ;, }, +, -, *, /, %, AND, OR, <, >, >=, <=, ==, !=, ,, ] }
40	<expr>	{ (, id, clocklit, hoverlit, coinlit, runelit, ropelit, NOT, ~ }
41	<expr>	{ (, ) }
42	<exprs>	{ (, ) }
43	<exprs>	{ (, ) }
44	<morestock_dec>	{ (, ) }
45	<morestock_dec>	{ (, ) }

46	<stock_dec>	{ = }
47	<stock_dec>	{ [ }
48	<stock_arrayOpt1>	{ = }
49	<stock_arrayOpt1>	{ [ ] }
50	<stock_arrayOpt2>	{ = }
51	<stock_arrayOpt2>	{ [ ] }
52	<stock_array1D>	{ { } }
53	<stock_array2D>	{ { } }
54	<stock_array3D>	{ { } }
55	<morestock_Arr1D>	{ , }
56	<morestock_Arr1D>	{ { } }
57	<morestock_Arr2D>	{ , }
58	<morestock_Arr2D>	{ { } }
59	<moretowerMem>	{ , }
60	<moretowerMem>	{ ; }
61	<moretowerMem_dec>	{ clock, hover, coin, rune, rope }
62	<moretowerMem_dec>	{ { } }
63	<array1D_ind>	{ [ ] }
64	<array1D_ind>	{ ( ), ;, }, =, +=, -=, *=, /=, %=, +, -, *, /, %, AND, OR, <, >, >=, <=, ==, !=, ++, --, ,, ], @, - }
65	<array2D_ind>	{ [ ] }
66	<array2D_ind>	{ ( ), ;, }, =, +=, -=, *=, /=, %=, +, -, *, /, %, AND, OR, <, >, >=, <=, ==, !=, ++, --, ,, ], @, - }
67	<array3D_ind>	{ [ ] }
68	<array3D_ind>	{ ( ), ;, }, =, +=, -=, *=, /=, %=, +, -, *, /, %, AND, OR, <, >, >=, <=, ==, !=, ++, --, ,, ], @, - }
69	<expression>	{ clocklit, hoverlit, coinlit, runelit, ropelit }
70	<expression>	{ id }
71	<expression>	{ ( ) }
72	<expression>	{ NOT }
73	<expression>	{ ~ }
74	<expr_ext>	{ +, -, *, /, %, AND, OR, <, >, >=, <=, ==, != }
75	<expr_ext>	{ ( ), ;, }, ,, ] }
76	<id_suffix>	{ ( ) }
77	<id_suffix>	{ ( ), ;, }, +, -, *, /, %, AND, OR, <, >, >=, <=, ==, !=, ++, --, ,, [, ], @ }
78	<id_at>	{ @ }
79	<id_at>	{ ( ), ;, }, =, +=, -=, *=, /=, %=, +, -, *, /, %, AND, OR, <, >, >=, <=, ==, !=, ++, --, ,, ], - }
80	<vardec_opt>	{ = }
81	<vardec_opt>	{ [ ] }
82	<vardec_opt>	{ ;, , }
83	<morevardec_opt>	{ , }
84	<morevardec_opt>	{ ; }
85	<array1D>	{ [ ] }
86	<array1D>	{ = }
87	<array1D>	{ ;, , }
88	<array2D>	{ [ ] }
89	<array2D>	{ = }

90	<array2D>	{ ;, , }
91	<array3D>	{ = }
92	<array3D>	{ ;, , }
93	<func_param>	{ id, clock, hover, coin, rune, rope }
94	<func_param>	{ () }
95	<parameter>	{ clock, hover, coin, rune, rope }
96	<parameter>	{ id }
97	<parameters>	{ , }
98	<parameters>	{ () }
99	<statements>	{ STOCK, id, clock, hover, coin, rune, rope, sprint, quit, skip, bag, flight, take, punch, say, listen, return }
100	<statements>	{ PLAY, }, item, potion }
101	<statement>	{ STOCK }
102	<statement>	{ clock, hover, coin, rune, rope }
103	<statement>	{ id }
104	<statement>	{ sprint }
105	<statement>	{ quit }
106	<statement>	{ skip }
107	<statement>	{ bag }
108	<statement>	{ flight }
109	<statement>	{ take }
110	<statement>	{ punch }
111	<statement>	{ say }
112	<statements>	{ listen }
113	<statement>	{ return }
114	<startswith_idNext>	{ id }
115	<startswith_idNext>	{ () }
116	<startswith_idNext>	{ =, +=, -=, *=, /=, %=, ++, [ , @, - ] }
117	<initialization>	{ clock, hover, coin, rune, rope }
118	<initialization>	{ id }
119	<initialization>	{ ; }
120	<expression_null>	{ (, id, clocklit, hoverlit, coinlit, runelit, ropelit, NOT, ~ }
121	<expression_null>	{ ; }
122	<iteration>	{ id }
123	<iteration>	{ () }
124	<iteration_suffix>	{ ++ }
125	<iteration_suffix>	{ -- }
126	<iteration_suffix>	{ =, +=, -=, *=, /=, %= }
127	<kick_statement>	{ kick }
128	<kick_statement>	{ PLAY, STOCK, id, }, clock, hover, coin, rune, rope, sprint, quit, skip, bag, flight, take, punch, say, listen, return, ultimate, item, potion }
129	<ultimate_statement>	{ ultimate }
130	<ultimate_statement>	{ PLAY, STOCK, id, }, clock, hover, coin, rune, rope, sprint, quit, skip, bag, flight, take, punch, say, listen, return, item, potion }
131	<item_statement>	{ item }
132	<item_statement>	{ }, potion }
133	<potion_statement>	{ potion }
134	<potion_statement>	{ () }

## XV. PREDICT TABLE

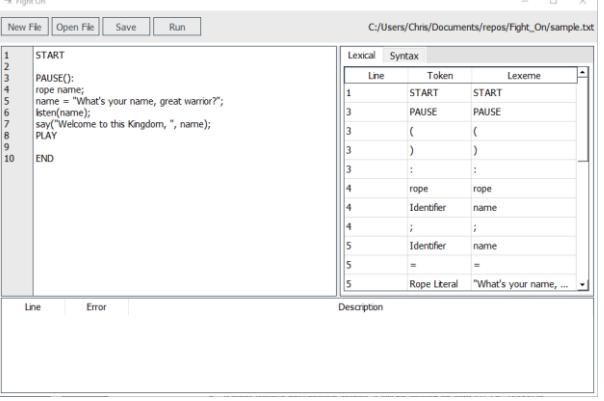


<code>+=</code>	<code>-=</code>	<code>*=</code>	<code>/=</code>	<code>%=</code>	<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>	<code>%</code>	<code>AND</code>	<code>OR</code>	<code>&lt;</code>	<code>&gt;</code>	<code>&gt;=</code>	<code>&lt;=</code>	<code>==</code>	<code>!=</code>	<code>++</code>	<code>--</code>
19	20	21	22	23															
					24	25	26	27	28	29	30	31	32	33	34	35	36		
					39	39	39	39	39	39	39	39	39	39	39	39	39	37	38
64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64
66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66	66
68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68	68
					74	74	74	74	74	74	74	74	74	74	74	74	74	74	
					77	77	77	77	77	77	77	77	77	77	77	77	77	77	77
79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79	79
116	116	116	116	116							116						116		
126	126	126	126	126													124		

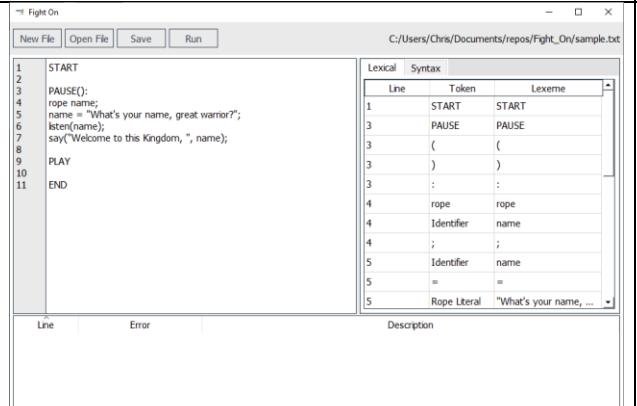
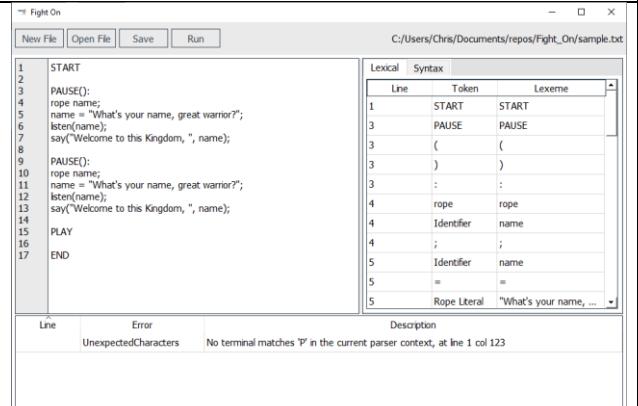


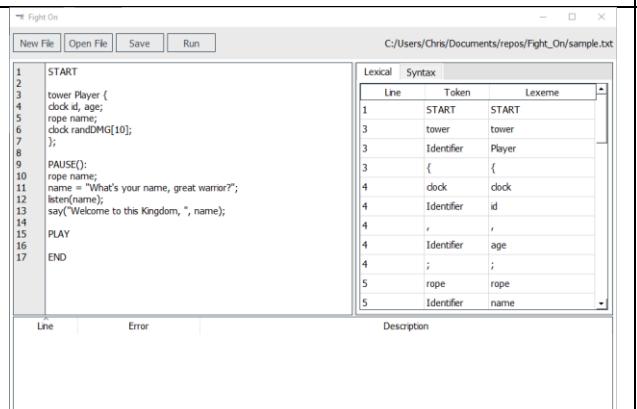
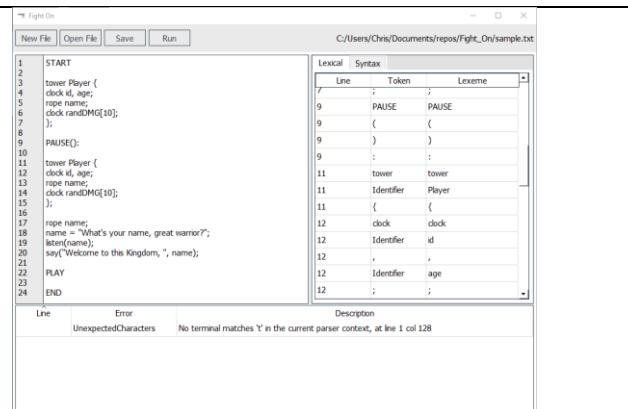


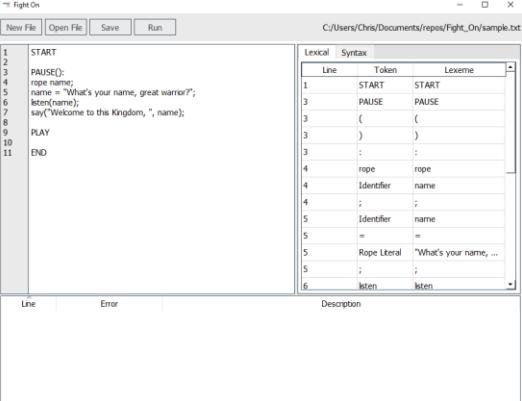
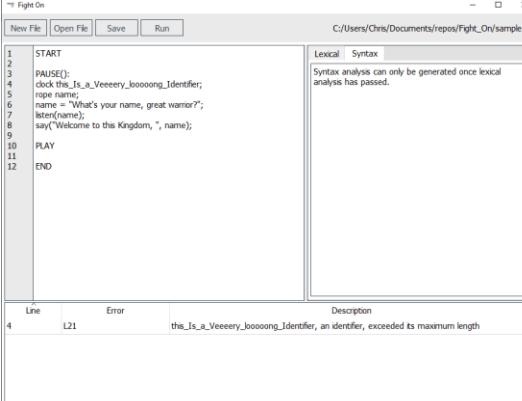
## XVI. TEST SCRIPTS: LEXICAL

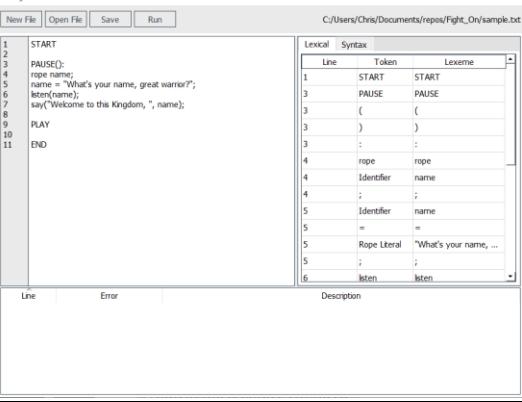
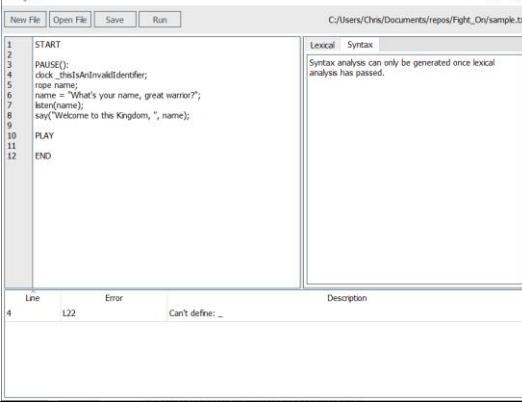
Rule	<p><b>FightOn is a case-sensitive language.</b></p>																							
Sample Input	<pre>START PAUSE(); rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name); PLAY END</pre>	<pre>start PAUSE(); rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name); PLAY END</pre>																						
Expected Output	<p>Lexical analyzer should NOT produce any errors and should generate tokens.</p>	<p>Lexical should NOT produce an error but consider “start” as an identifier not the reserved word “START”.</p>																						
Actual Output	 <table border="1"> <thead> <tr> <th>Line</th> <th>Token</th> <th>Lexeme</th> </tr> </thead> <tbody> <tr><td>1</td><td>START</td><td>START</td></tr> <tr><td>2</td><td>PAUSE</td><td>PAUSE</td></tr> <tr><td>3</td><td>rope</td><td>rope</td></tr> <tr><td>4</td><td>name</td><td>name</td></tr> <tr><td>5</td><td>=</td><td>=</td></tr> <tr><td>6</td><td>Rope Literal</td><td>"What's your name, ...</td></tr> </tbody> </table>			Line	Token	Lexeme	1	START	START	2	PAUSE	PAUSE	3	rope	rope	4	name	name	5	=	=	6	Rope Literal	"What's your name, ...
Line	Token	Lexeme																						
1	START	START																						
2	PAUSE	PAUSE																						
3	rope	rope																						
4	name	name																						
5	=	=																						
6	Rope Literal	"What's your name, ...																						
Remarks	<p>PASSED</p>																							
<p>PASSED</p>																								

**NOTE:** the error comes from the syntax analysis

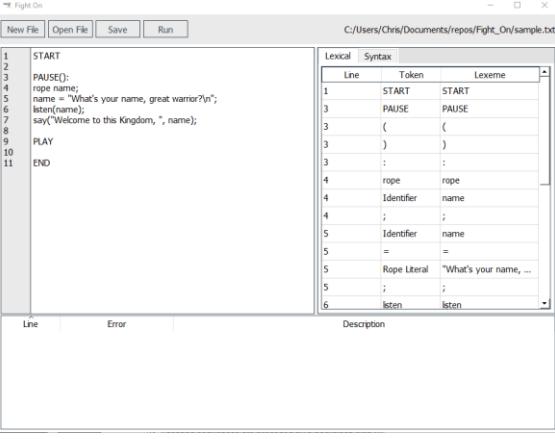
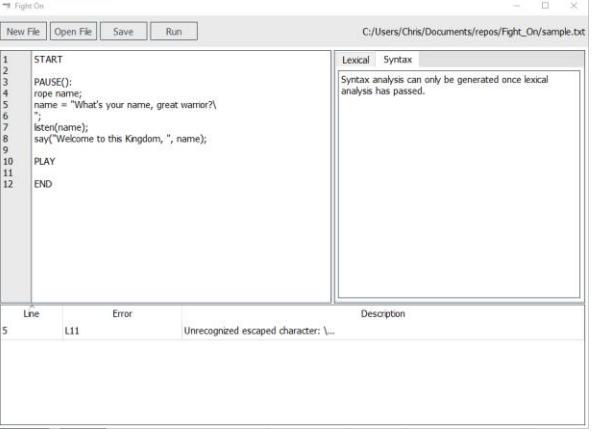
Rule	<p><b>A program can only have one PAUSE() function.</b></p>		
Sample Input	<pre>START PAUSE(): rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name); PLAY END</pre>	<pre>START PAUSE(): rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name); PAUSE(): rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name); PLAY END</pre>	
Expected Output	<p>Lexical analyzer should NOT produce any errors and should generate tokens.</p>	<p>Lexical analyzer should NOT produce any errors and should generate tokens.</p>	
Actual Output	 <p><b>NOTE:</b> the error comes from the syntax analysis</p>	 <p><b>NOTE:</b> the error comes from the syntax analysis</p>	
Remarks	<p>PASSED</p>		

Rule	<p><b>All global variables and user-defined functions must be declared before the PAUSE() function</b></p>																																																																																																																			
Sample Input	<pre> START  tower Player { clock id, age; rope name; clock randDMG[10]; };  PAUSE(): rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name);  PLAY  END </pre>	<pre> START  tower Player { clock id, age; rope name; clock randDMG[10]; };  PAUSE():  tower Player { clock id, age; rope name; clock randDMG[10]; };  rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name);  PLAY  END </pre> <p><b>NOTE:</b> tower can only be declared globally.</p>																																																																																																																		
Expected Output	Lexical analyzer should NOT produce any errors and should generate tokens.	Lexical analyzer should NOT produce any errors and should generate tokens.																																																																																																																		
Actual Output	 <p>The screenshot shows the lexical analysis results for the provided code. The tokens are listed in two columns: Line and Token. The Lexeme column shows the original source code. The tokens are:</p> <table border="1"> <thead> <tr> <th>Line</th> <th>Token</th> <th>Lexeme</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>START</td> <td>START</td> </tr> <tr> <td>3</td> <td>tower</td> <td>tower</td> </tr> <tr> <td>4</td> <td>Identifier</td> <td>Player</td> </tr> <tr> <td>4</td> <td>{</td> <td>{</td> </tr> <tr> <td>4</td> <td>clock</td> <td>clock</td> </tr> <tr> <td>4</td> <td>Identifier</td> <td>id</td> </tr> <tr> <td>4</td> <td>,</td> <td>,</td> </tr> <tr> <td>4</td> <td>Identifier</td> <td>age</td> </tr> <tr> <td>4</td> <td>;</td> <td>;</td> </tr> <tr> <td>5</td> <td>rope</td> <td>rope</td> </tr> <tr> <td>5</td> <td>Identifier</td> <td>name</td> </tr> <tr> <td>9</td> <td>PAUSE()</td> <td></td> </tr> <tr> <td>10</td> <td>rope</td> <td></td> </tr> <tr> <td>11</td> <td>name</td> <td>= "What's your name, great warrior?"</td> </tr> <tr> <td>11</td> <td>listen</td> <td>(name)</td> </tr> <tr> <td>11</td> <td>say</td> <td>("Welcome to this Kingdom, ", name)</td> </tr> <tr> <td>15</td> <td>PLAY</td> <td></td> </tr> <tr> <td>16</td> <td>END</td> <td></td> </tr> </tbody> </table>	Line	Token	Lexeme	1	START	START	3	tower	tower	4	Identifier	Player	4	{	{	4	clock	clock	4	Identifier	id	4	,	,	4	Identifier	age	4	;	;	5	rope	rope	5	Identifier	name	9	PAUSE()		10	rope		11	name	= "What's your name, great warrior?"	11	listen	(name)	11	say	("Welcome to this Kingdom, ", name)	15	PLAY		16	END		 <p>The screenshot shows the lexical analysis results for the provided code. The tokens are listed in two columns: Line and Token. The Lexeme column shows the original source code. The tokens are:</p> <table border="1"> <thead> <tr> <th>Line</th> <th>Token</th> <th>Lexeme</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>START</td> <td>START</td> </tr> <tr> <td>3</td> <td>tower</td> <td>tower</td> </tr> <tr> <td>4</td> <td>Identifier</td> <td>Player</td> </tr> <tr> <td>4</td> <td>{</td> <td>{</td> </tr> <tr> <td>4</td> <td>clock</td> <td>clock</td> </tr> <tr> <td>4</td> <td>Identifier</td> <td>id</td> </tr> <tr> <td>4</td> <td>,</td> <td>,</td> </tr> <tr> <td>4</td> <td>Identifier</td> <td>age</td> </tr> <tr> <td>4</td> <td>;</td> <td>;</td> </tr> <tr> <td>5</td> <td>rope</td> <td>rope</td> </tr> <tr> <td>5</td> <td>Identifier</td> <td>name</td> </tr> <tr> <td>9</td> <td>PAUSE()</td> <td></td> </tr> <tr> <td>10</td> <td>rope</td> <td></td> </tr> <tr> <td>11</td> <td>name</td> <td>= "What's your name, great warrior?"</td> </tr> <tr> <td>11</td> <td>listen</td> <td>(name)</td> </tr> <tr> <td>11</td> <td>say</td> <td>("Welcome to this Kingdom, ", name)</td> </tr> <tr> <td>15</td> <td>PLAY</td> <td></td> </tr> <tr> <td>16</td> <td>END</td> <td></td> </tr> </tbody> </table> <p><b>Line</b>    <b>Error</b>    <b>Description</b></p> <p>Line    Error    Description</p> <p>UnexpectedCharacters    No terminal matches 'Y' in the current parser context, at line 1 col 128</p>	Line	Token	Lexeme	1	START	START	3	tower	tower	4	Identifier	Player	4	{	{	4	clock	clock	4	Identifier	id	4	,	,	4	Identifier	age	4	;	;	5	rope	rope	5	Identifier	name	9	PAUSE()		10	rope		11	name	= "What's your name, great warrior?"	11	listen	(name)	11	say	("Welcome to this Kingdom, ", name)	15	PLAY		16	END	
Line	Token	Lexeme																																																																																																																		
1	START	START																																																																																																																		
3	tower	tower																																																																																																																		
4	Identifier	Player																																																																																																																		
4	{	{																																																																																																																		
4	clock	clock																																																																																																																		
4	Identifier	id																																																																																																																		
4	,	,																																																																																																																		
4	Identifier	age																																																																																																																		
4	;	;																																																																																																																		
5	rope	rope																																																																																																																		
5	Identifier	name																																																																																																																		
9	PAUSE()																																																																																																																			
10	rope																																																																																																																			
11	name	= "What's your name, great warrior?"																																																																																																																		
11	listen	(name)																																																																																																																		
11	say	("Welcome to this Kingdom, ", name)																																																																																																																		
15	PLAY																																																																																																																			
16	END																																																																																																																			
Line	Token	Lexeme																																																																																																																		
1	START	START																																																																																																																		
3	tower	tower																																																																																																																		
4	Identifier	Player																																																																																																																		
4	{	{																																																																																																																		
4	clock	clock																																																																																																																		
4	Identifier	id																																																																																																																		
4	,	,																																																																																																																		
4	Identifier	age																																																																																																																		
4	;	;																																																																																																																		
5	rope	rope																																																																																																																		
5	Identifier	name																																																																																																																		
9	PAUSE()																																																																																																																			
10	rope																																																																																																																			
11	name	= "What's your name, great warrior?"																																																																																																																		
11	listen	(name)																																																																																																																		
11	say	("Welcome to this Kingdom, ", name)																																																																																																																		
15	PLAY																																																																																																																			
16	END																																																																																																																			
Remarks	<p>PASSED</p>																																																																																																																			

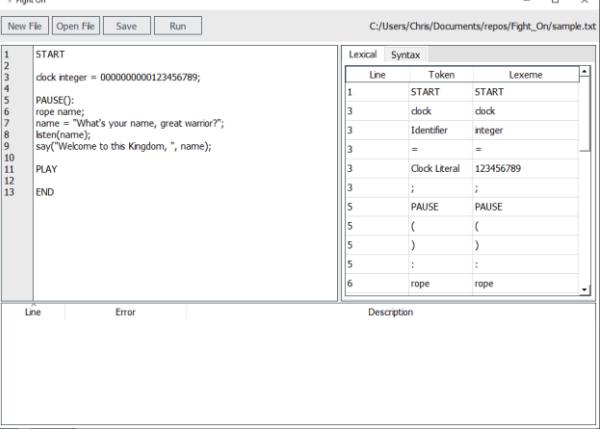
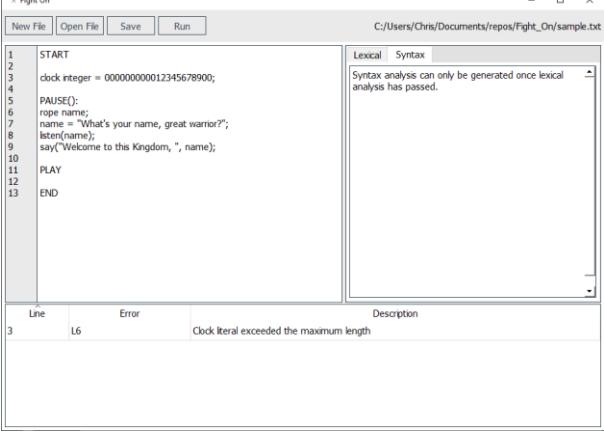
Rule	<p><b>Identifiers have a maximum of 25 characters – composed of letters, numbers and underscores –, and must always start with a letter.</b></p>
Sample Input	<p>START  PAUSE():  rope name;  name = "What's your name, great warrior?";  listen(name);  say("Welcome to this Kingdom, ", name);  PLAY  END</p>
Expected Output	<p>Lexical analyzer should NOT produce any errors and should generate tokens.</p>
Actual Output	 
Remarks	<p>PASSED</p>
Sample Input	<p>START  PAUSE():  rope name;  name = "What's your name, great warrior?";  listen(name);  say("Welcome to this Kingdom, ", name);  PLAY  END</p>
	<p>START  PAUSE():  clock _thisIsAnInvalidIdentifier;  rope name;  name = "What's your name, great warrior?";  listen(name);  say("Welcome to this Kingdom, ", name);  PLAY  END</p>

<b>Expected Output</b>	Lexical analyzer should NOT produce any errors and should generate tokens.	Lexical analyzer should produce an error indicating that it cannot define “_” and prevent the syntax analysis to be generated.																																																						
<b>Actual Output</b>	 <p>The screenshot shows the lexical analysis results for the file C:/Users/Chris/Documents/repos/Fight_On/sample.txt. The tokens are:</p> <table border="1"> <thead> <tr> <th>Line</th> <th>Token</th> <th>Lexeme</th> </tr> </thead> <tbody> <tr><td>1</td><td>START</td><td>START</td></tr> <tr><td>2</td><td>PAUSE()</td><td>PAUSE()</td></tr> <tr><td>3</td><td>rope name;</td><td>rope name;</td></tr> <tr><td>4</td><td>name = "What's your name, great warrior?";</td><td>name = "What's your name, great warrior?";</td></tr> <tr><td>5</td><td>listen(name);</td><td>listen(name);</td></tr> <tr><td>6</td><td>say("Welcome to this Kingdom, ", name);</td><td>say("Welcome to this Kingdom, ", name);</td></tr> <tr><td>7</td><td>PLAY</td><td>PLAY</td></tr> <tr><td>8</td><td>END</td><td>END</td></tr> </tbody> </table>	Line	Token	Lexeme	1	START	START	2	PAUSE()	PAUSE()	3	rope name;	rope name;	4	name = "What's your name, great warrior?";	name = "What's your name, great warrior?";	5	listen(name);	listen(name);	6	say("Welcome to this Kingdom, ", name);	say("Welcome to this Kingdom, ", name);	7	PLAY	PLAY	8	END	END	 <p>The screenshot shows the lexical analysis results for the file C:/Users/Chris/Documents/repos/Fight_On/sample.txt. The tokens are:</p> <table border="1"> <thead> <tr> <th>Line</th> <th>Token</th> <th>Lexeme</th> </tr> </thead> <tbody> <tr><td>1</td><td>START</td><td>START</td></tr> <tr><td>2</td><td>PAUSE()</td><td>PAUSE()</td></tr> <tr><td>3</td><td>rope name;</td><td>rope name;</td></tr> <tr><td>4</td><td>name = "What's your name, great warrior?";</td><td>name = "What's your name, great warrior?";</td></tr> <tr><td>5</td><td>listen(name);</td><td>listen(name);</td></tr> <tr><td>6</td><td>say("Welcome to this Kingdom, ", name);</td><td>say("Welcome to this Kingdom, ", name);</td></tr> <tr><td>7</td><td>PLAY</td><td>PLAY</td></tr> <tr><td>8</td><td>END</td><td>END</td></tr> </tbody> </table> <p>An error message is displayed: "Syntax analysis can only be generated once lexical analysis has passed." This indicates that the lexical analyzer failed to recognize the identifier "name".</p>	Line	Token	Lexeme	1	START	START	2	PAUSE()	PAUSE()	3	rope name;	rope name;	4	name = "What's your name, great warrior?";	name = "What's your name, great warrior?";	5	listen(name);	listen(name);	6	say("Welcome to this Kingdom, ", name);	say("Welcome to this Kingdom, ", name);	7	PLAY	PLAY	8	END	END
Line	Token	Lexeme																																																						
1	START	START																																																						
2	PAUSE()	PAUSE()																																																						
3	rope name;	rope name;																																																						
4	name = "What's your name, great warrior?";	name = "What's your name, great warrior?";																																																						
5	listen(name);	listen(name);																																																						
6	say("Welcome to this Kingdom, ", name);	say("Welcome to this Kingdom, ", name);																																																						
7	PLAY	PLAY																																																						
8	END	END																																																						
Line	Token	Lexeme																																																						
1	START	START																																																						
2	PAUSE()	PAUSE()																																																						
3	rope name;	rope name;																																																						
4	name = "What's your name, great warrior?";	name = "What's your name, great warrior?";																																																						
5	listen(name);	listen(name);																																																						
6	say("Welcome to this Kingdom, ", name);	say("Welcome to this Kingdom, ", name);																																																						
7	PLAY	PLAY																																																						
8	END	END																																																						
<b>Remarks</b>	PASSED	PASSED																																																						

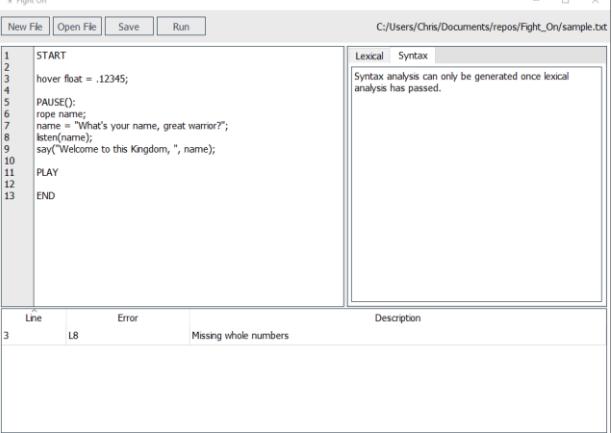
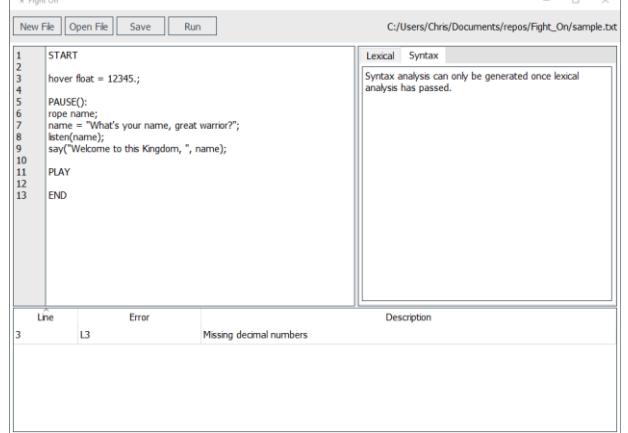
<b>Rule</b>	Escaped sequences are preceded by a backslash sign (\).
<b>Sample Input</b>	<pre> START PAUSE(): rope name; name = "What's your name, great warrior?\n"; listen(name); say("Welcome to this Kingdom, ", name);  PLAY END </pre> <pre> START PAUSE(): rope name; name = "What's your name, great warrior?\\"; // Escaped quote listen(name); say("Welcome to this Kingdom, ", name);  PLAY END </pre>

<b>Expected Output</b>	Lexical analyzer should NOT produce any errors and should generate tokens.	Lexical analyzer should produce an error indicating that an unrecognized escaped character was encountered and prevent the syntax analysis to be generated.																																																
<b>Actual Output</b>	 <p>The screenshot shows a lexical analysis interface with two panes. The left pane displays the input code:</p> <pre> 1 START 2 PAUSE(); 3 rope name; 4 name = "What's your name, great warrior?\n"; 5 listen(name); 6 say("Welcome to this Kingdom, ", name); 7 8 PLAY 9 10 END </pre> <p>The right pane shows the token output:</p> <table border="1"> <thead> <tr> <th>Line</th> <th>Token</th> <th>Lexeme</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>START</td> <td>START</td> </tr> <tr> <td>2</td> <td>PAUSE</td> <td>PAUSE</td> </tr> <tr> <td>3</td> <td>(</td> <td>(</td> </tr> <tr> <td>3</td> <td>)</td> <td>)</td> </tr> <tr> <td>3</td> <td>:</td> <td>:</td> </tr> <tr> <td>4</td> <td>rope</td> <td>rope</td> </tr> <tr> <td>4</td> <td>Identifier</td> <td>name</td> </tr> <tr> <td>4</td> <td>;</td> <td>;</td> </tr> <tr> <td>5</td> <td>Identifier</td> <td>name</td> </tr> <tr> <td>5</td> <td>=</td> <td>=</td> </tr> <tr> <td>5</td> <td>Rope Literal</td> <td>"What's your name, ...</td> </tr> <tr> <td>5</td> <td>;</td> <td>;</td> </tr> <tr> <td>6</td> <td>listen</td> <td>listen</td> </tr> </tbody> </table>	Line	Token	Lexeme	1	START	START	2	PAUSE	PAUSE	3	(	(	3	)	)	3	:	:	4	rope	rope	4	Identifier	name	4	;	;	5	Identifier	name	5	=	=	5	Rope Literal	"What's your name, ...	5	;	;	6	listen	listen	 <p>The screenshot shows a lexical analysis interface with two panes. The left pane displays the input code, identical to the one in the actual output. The right pane shows the token output and an error message:</p> <table border="1"> <thead> <tr> <th>Line</th> <th>Error</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>5</td> <td>L11</td> <td>Unrecognized escaped character: \...</td> </tr> </tbody> </table> <p>A note above the error message states: "Syntax analysis can only be generated once lexical analysis has passed."</p>	Line	Error	Description	5	L11	Unrecognized escaped character: \...
Line	Token	Lexeme																																																
1	START	START																																																
2	PAUSE	PAUSE																																																
3	(	(																																																
3	)	)																																																
3	:	:																																																
4	rope	rope																																																
4	Identifier	name																																																
4	;	;																																																
5	Identifier	name																																																
5	=	=																																																
5	Rope Literal	"What's your name, ...																																																
5	;	;																																																
6	listen	listen																																																
Line	Error	Description																																																
5	L11	Unrecognized escaped character: \...																																																
<b>Remarks</b>	PASSED	PASSED																																																

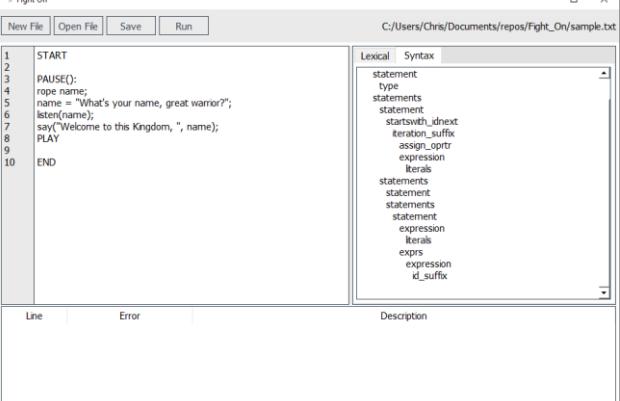
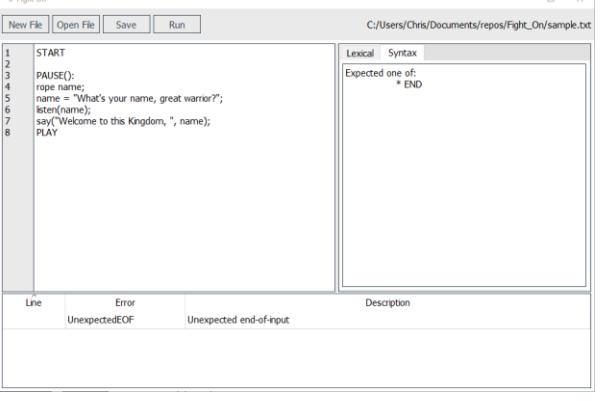
<b>Rule</b>	Clock literals can only have one up to ten place values and leading zeroes are will be omitted.	
<b>Sample Input</b>	<pre> START clock integer = 0000000000123456789; PAUSE(): rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name);  PLAY END </pre>	<pre> START clock integer = 000000000012345678900; PAUSE(): rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name);  PLAY END </pre>

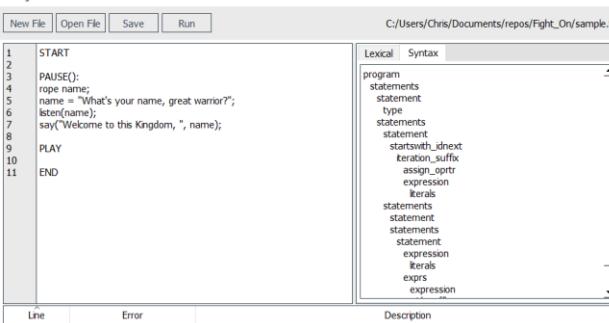
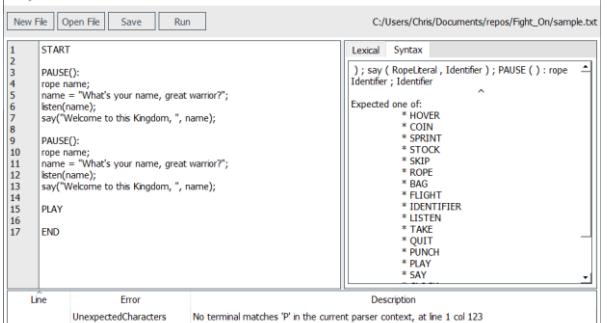
<b>Expected Output</b>	Lexical analyzer should NOT produce any errors and should generate tokens.	Lexical analyzer should produce an error indicating that a clock literal exceeded its maximum length and prevent the syntax analysis to be generated.																																										
<b>Actual Output</b>	 <p>The screenshot shows the lexical analysis interface with the file 'C:/Users/Chris/Documents/repos/Fight_On/sample.txt' loaded. The tokens are listed in a table:</p> <table border="1"> <thead> <tr> <th>Line</th> <th>Token</th> <th>Lexeme</th> </tr> </thead> <tbody> <tr><td>1</td><td>START</td><td>START</td></tr> <tr><td>2</td><td>clock</td><td>clock</td></tr> <tr><td>3</td><td>Identifier</td><td>integer</td></tr> <tr><td>3</td><td>=</td><td>=</td></tr> <tr><td>3</td><td>Clock Literal</td><td>123456789</td></tr> <tr><td>3</td><td>;</td><td>;</td></tr> <tr><td>5</td><td>PAUSE</td><td>PAUSE</td></tr> <tr><td>5</td><td>(</td><td>(</td></tr> <tr><td>5</td><td>)</td><td>)</td></tr> <tr><td>5</td><td>:</td><td>:</td></tr> <tr><td>6</td><td>rope</td><td>rope</td></tr> </tbody> </table>	Line	Token	Lexeme	1	START	START	2	clock	clock	3	Identifier	integer	3	=	=	3	Clock Literal	123456789	3	;	;	5	PAUSE	PAUSE	5	(	(	5	)	)	5	:	:	6	rope	rope	 <p>The screenshot shows the lexical analysis interface with the same file loaded. An error message is displayed: "Syntax analysis can only be generated once lexical analysis has passed." Below it, a specific error is shown:</p> <table border="1"> <thead> <tr> <th>Line</th> <th>Error</th> <th>Description</th> </tr> </thead> <tbody> <tr><td>3</td><td>L6</td><td>Clock literal exceeded the maximum length</td></tr> </tbody> </table>	Line	Error	Description	3	L6	Clock literal exceeded the maximum length
Line	Token	Lexeme																																										
1	START	START																																										
2	clock	clock																																										
3	Identifier	integer																																										
3	=	=																																										
3	Clock Literal	123456789																																										
3	;	;																																										
5	PAUSE	PAUSE																																										
5	(	(																																										
5	)	)																																										
5	:	:																																										
6	rope	rope																																										
Line	Error	Description																																										
3	L6	Clock literal exceeded the maximum length																																										
<b>Remarks</b>	PASSED	PASSED																																										

<b>Rule</b>	Hover literals cannot start and end with decimal point.	
<b>Sample Input</b>	<pre> START hover float = .12345; PAUSE(); rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name); PLAY END </pre>	<pre> START hover float = 12345.; PAUSE(); rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name); PLAY END </pre>

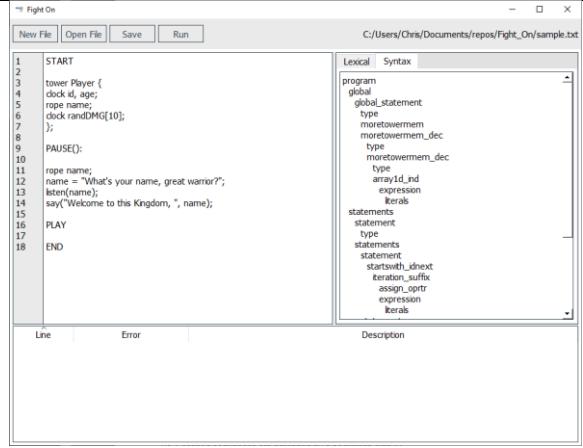
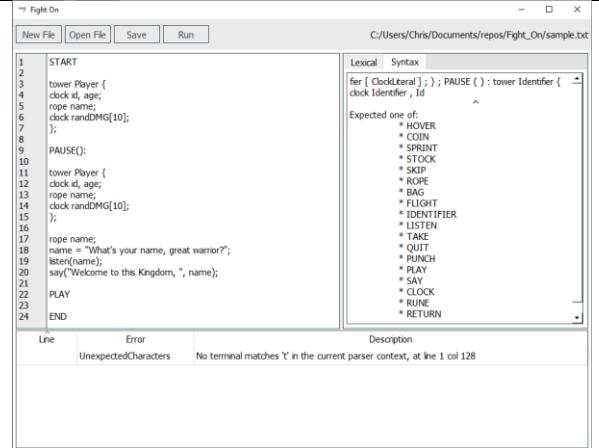
<b>Expected Output</b>	<p>Lexical analyzer should produce an error indicating the whole numbers are missing and prevent the syntax analysis to be generated.</p>	<p>Lexical analyzer should produce an error indicating the decimal numbers are missing and prevent the syntax analysis to be generated.</p>												
<b>Actual Output</b>	 <p>The screenshot shows the 'Fight On' application window. The file path is C:/Users/Chris/Documents/repos/Fight_On/sample.txt. The code editor contains the following pseudocode:</p> <pre> 1 START 2 hover float = .12345; 3 4 PAUSE(); 5 rope name; 6 name = "What's your name, great warrior?"; 7 listen(name); 8 say("Welcome to this Kingdom, ", name); 9 10 PLAY 11 12 END </pre> <p>The syntax analysis panel on the right displays the message: "Syntax analysis can only be generated once lexical analysis has passed." Below the editor, a table shows the error details:</p> <table border="1"> <thead> <tr> <th>Line</th> <th>Error</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>3</td> <td>L8</td> <td>Missing whole numbers</td> </tr> </tbody> </table>	Line	Error	Description	3	L8	Missing whole numbers	 <p>The screenshot shows the 'Fight On' application window. The file path is C:/Users/Chris/Documents/repos/Fight_On/sample.txt. The code editor contains the same pseudocode as the first screenshot.</p> <p>The syntax analysis panel on the right displays the message: "Syntax analysis can only be generated once lexical analysis has passed." Below the editor, a table shows the error details:</p> <table border="1"> <thead> <tr> <th>Line</th> <th>Error</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>3</td> <td>L3</td> <td>Missing decimal numbers</td> </tr> </tbody> </table>	Line	Error	Description	3	L3	Missing decimal numbers
Line	Error	Description												
3	L8	Missing whole numbers												
Line	Error	Description												
3	L3	Missing decimal numbers												
<b>Remarks</b>	PASSED	PASSED												

## XVII. TEST SCRIPTS: SYNTAX

<b>Rule</b> <p>The program always starts and ends with the reserved words START, and END respectively.</p>			
<b>Sample Input</b> <pre>START PAUSE(); rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name); PLAY END</pre>	<pre>START PAUSE(); rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name); PLAY</pre>		
<b>Expected Output</b> <p>Syntax analyzer should NOT produce any errors and must generate the parse tree.</p>		<p>Syntax analyzer should produce an End-of-File (EOF) Error and display the expected token.</p>	
<b>Actual Output</b> 			
<b>Remarks</b> <p>PASSED</p>	<p>PASSED</p>		
<b>Rule</b> <p>A program can only have one PAUSE() function.</p>			

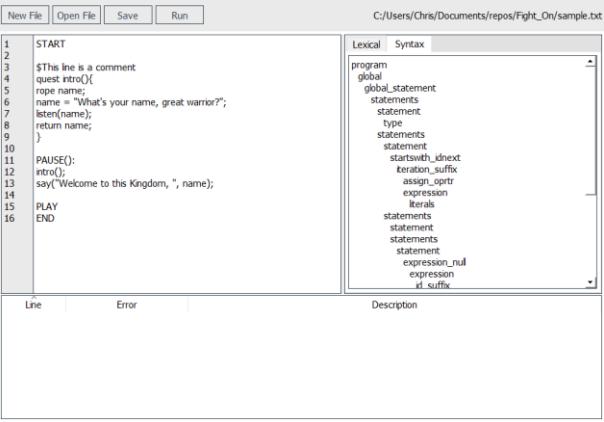
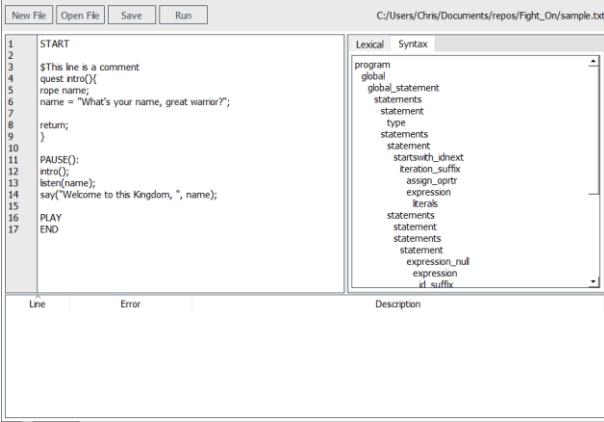
Sample Input	<pre> START PAUSE(): rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name); PLAY END </pre>	<pre> START PAUSE(): rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name);  PAUSE(): rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name);  PLAY  END </pre>
Expected Output	<p>Syntax analyzer should NOT produce any errors and should generate the parse tree.</p>	<p>Syntax analyzer should produce an Unexpected Characters Error and display the expected token.</p>
Actual Output		
Remarks	PASSED	PASSED

Rule	All global variables and user-defined functions must be declared before the PAUSE() function
------	--

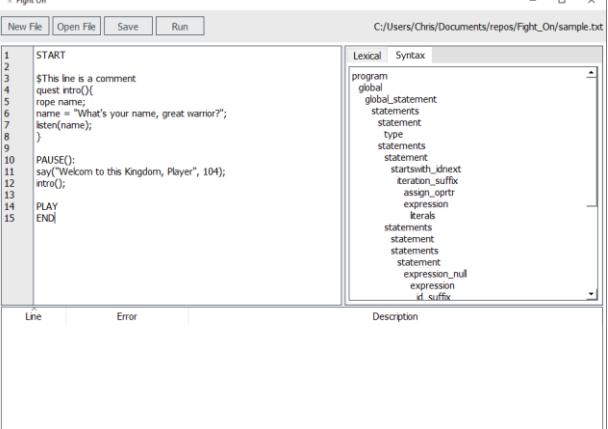
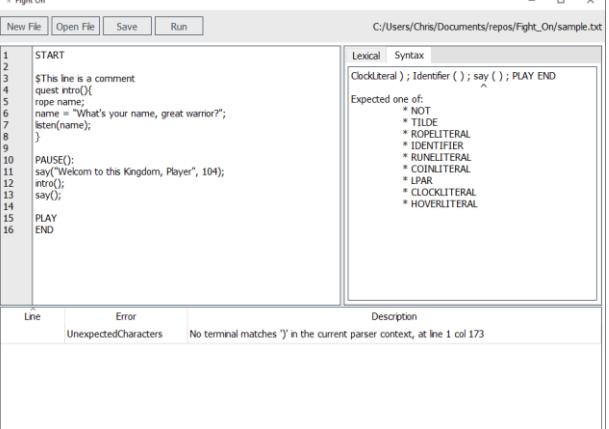
Sample Input	<pre> START  tower Player {     clock id, age;     rope name;     clock randDMG[10]; };  PAUSE(): rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name);  PLAY  END </pre>	<pre> START  tower Player {     clock id, age;     rope name;     clock randDMG[10]; };  PAUSE():  tower Player {     clock id, age;     rope name;     clock randDMG[10]; };  rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name);  PLAY  END </pre> <p><b>NOTE:</b> tower can only be declared globally.</p>
Expected Output	Syntax analyzer should NOT produce any errors and should generate the parse tree.	Syntax analyzer should produce an Unexpected Characters Error and display the expected token.
Actual Output	 <p>The screenshot shows the 'Lexical' and 'Syntax' tabs of the parser interface. The lexical tab displays tokens corresponding to the input code. The syntax tab shows a hierarchical parse tree starting from the 'program' node, which contains 'global_statement', 'type', 'moretowermen', and 'moretowermen_dec' nodes. The 'moretowermen_dec' node further expands into 'type', 'moretowermen', and 'statements' nodes.</p>	 <p>The screenshot shows the 'Lexical' and 'Syntax' tabs. The lexical tab shows tokens. The syntax tab shows a parse tree starting from 'program'. A specific node in the tree is highlighted with an error message: 'Expected one of: * HOVER * COIN * SPRINT * CLOCK * SKIP * ROPE * BAG * FIGHT * IDENTIFIER * LISTEN * TAKE * QUIT * FIGHT * PUNCH * PLAY * SAY * CLOCK * RUINE * RETURN'. Below the tree, an error message states: 'UnexpectedCharacters No terminal matches 'Y' in the current parser context, at line 1 col 128'.</p>
Remarks	PASSED	PASSED

Rule	<b>Initialization of constants must be done right after its declaration. It is done with the use of assignment operator (=).</b>
------	--

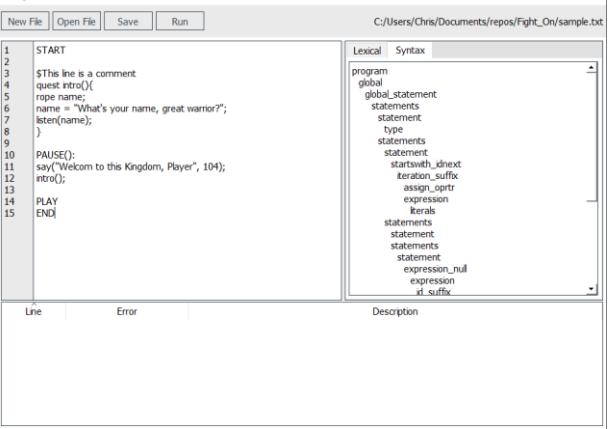
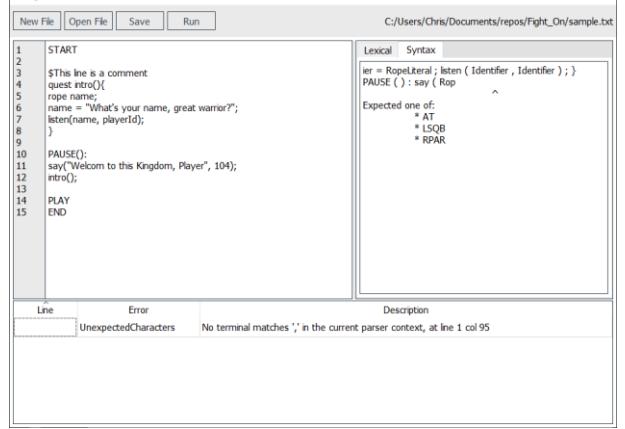
Sample Input	<pre> START  STOCK hover floatVar = 12.2;  PAUSE(): rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name);  PLAY  END </pre>	<pre> START  STOCK hover floatVar;  PAUSE(): rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name);  PLAY  END </pre>
Expected Output	Syntax analyzer should NOT produce any errors and should generate the parse tree.	Syntax analyzer should produce an Unexpected Characters Error and display the expected token.
Actual Output		
Remarks	PASSED	PASSED
Rule	<p><b>Return statements can have one or no value returned.</b></p>	

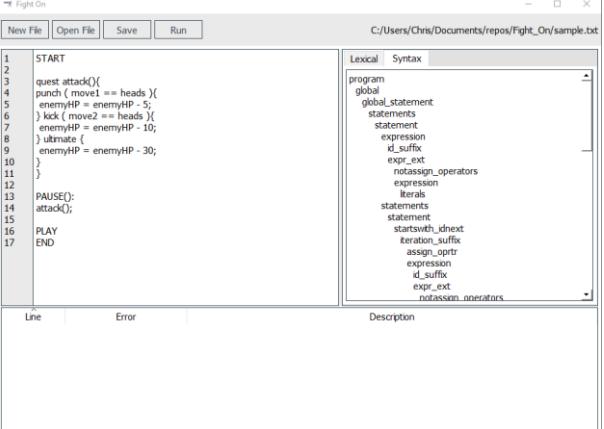
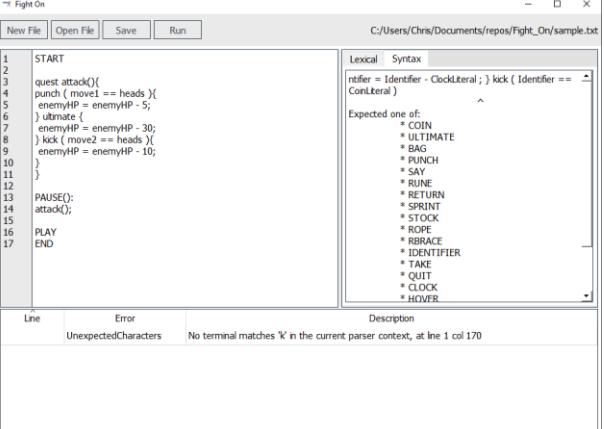
<b>Sample Input</b>	<pre> START  \$This line is a comment quest intro(){ rope name; name = "What's your name, great warrior?"; listen(name); return name; }  PAUSE(): intro(); say("Welcome to this Kingdom, ", name);  PLAY END </pre>	<pre> START  \$This line is a comment quest intro(){ rope name; name = "What's your name, great warrior?";  return; }  PAUSE(): intro(); listen(name); say("Welcome to this Kingdom, ", name);  PLAY END </pre>
<b>Expected Output</b>	Syntax analyzer should NOT produce any errors and should generate the parse tree.	Syntax analyzer should NOT produce any errors and should generate the parse tree.
<b>Actual Output</b>	 <p>The screenshot shows a window titled 'Fight On' with tabs for 'Lexical' and 'Syntax'. The 'Syntax' tab is active, displaying a parse tree for the provided code. The root node is 'program', which contains 'global', 'global_statement', 'statements', 'statement', 'type', 'statements', 'statement', and 'statements'. The 'statement' node under 'statements' has children 'startwith_idnext', 'iteration_suffix', 'assign_optr', 'expression', and 'literals'. The 'statements' node under 'statements' has children 'statement', 'statements', 'statement', 'expression_nul', 'expression_nul', and 'id_sufix'. The code in the editor is identical to the expected output.</p>	 <p>The screenshot shows a window titled 'Fight On' with tabs for 'Lexical' and 'Syntax'. The 'Syntax' tab is active, displaying a parse tree for the provided code. The root node is 'program', which contains 'global', 'global_statement', 'statements', 'statement', 'type', 'statements', 'statement', and 'statements'. The 'statement' node under 'statements' has children 'startwith_idnext', 'iteration_suffix', 'assign_optr', 'expression', and 'literals'. The 'statements' node under 'statements' has children 'statement', 'statements', 'statement', 'expression_nul', 'expression_nul', and 'id_sufix'. The code in the editor is identical to the expected output.</p>
<b>Remarks</b>	PASSED	PASSED

<b>Rule</b>	<p><b>Say statements need to have at least one parameter.</b></p>
-------------	---

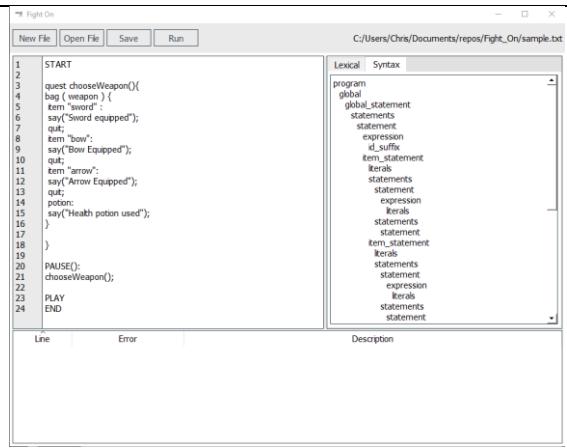
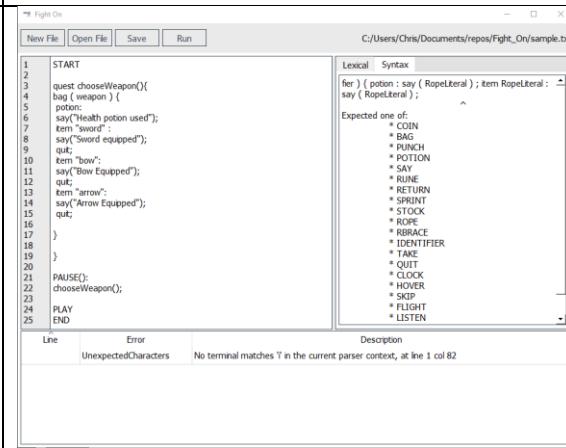
<b>Sample Input</b>	<pre> START  \$This line is a comment quest intro(){ rope name; name = "What's your name, great warrior?"; listen(name); }  PAUSE(): say("Welcom to this Kingdom, Player", 104); intro(); say();  PLAY END </pre>	<pre> START  \$This line is a comment quest intro(){ rope name; name = "What's your name, great warrior?"; listen(name); }  PAUSE(): say("Welcom to this Kingdom, Player", 104); intro(); say();  PLAY END </pre>
<b>Expected Output</b>	Syntax analyzer should NOT produce any errors and should generate the parse tree.	Syntax analyzer should produce an Unexpected Characters Error and display the expected token.
<b>Actual Output</b>		
<b>Remarks</b>	PASSED	PASSED

<b>Rule</b>	<p><b>Listen statements can only have one parameter.</b></p>
-------------	--

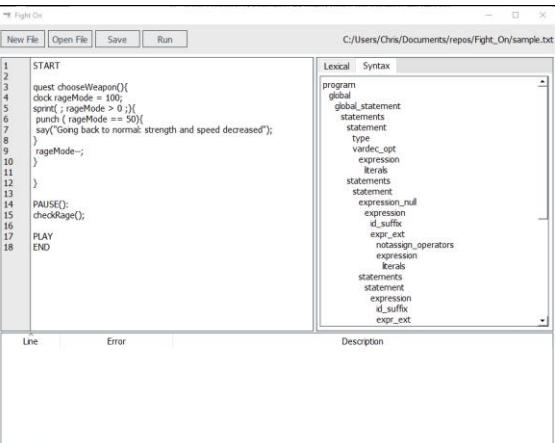
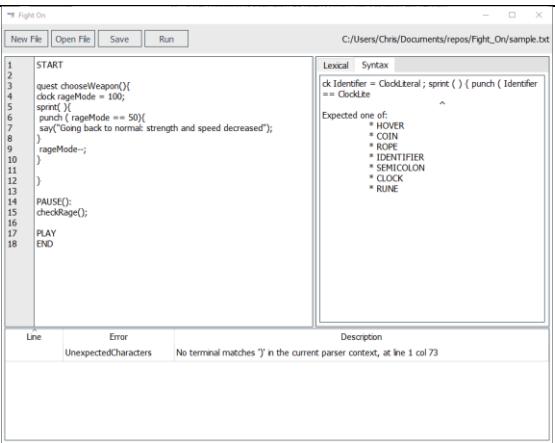
Sample Input	<pre> START  \$This line is a comment quest intro(){ rope name; name = "What's your name, great warrior?"; listen(name); }  PAUSE(): say("Welcom to this Kingdom, Player", 104); intro();  PLAY END </pre>	<pre> START  \$This line is a comment quest intro(){ rope name; name = "What's your name, great warrior?"; listen(name, playerId); }  PAUSE(): say("Welcom to this Kingdom, Player", 104); intro();  PLAY END </pre>
Expected Output	Syntax analyzer should NOT produce any errors and should generate the parse tree.	Syntax analyzer should produce an Unexpected Characters Error and display the expected token.
Actual Output		
Remarks	PASSED	PASSED
Rule	<p>The kick and ultimate statements can only be initiated after the punch statement, and kick statement cannot come after the ultimate statement.</p>	

Sample Input	<pre> START  quest attack(){ punch ( move1 == heads ){     enemyHP = enemyHP - 5; } kick ( move2 == heads ){     enemyHP = enemyHP - 10; } ultimate {     enemyHP = enemyHP - 30; } }  PAUSE(): attack();  PLAY END </pre>	<pre> START  quest attack(){ punch ( move1 == heads ){     enemyHP = enemyHP - 5; } ultimate {     enemyHP = enemyHP - 30; } kick ( move2 == heads ){     enemyHP = enemyHP - 10; } }  PAUSE(): attack();  PLAY END </pre>
Expected Output	Syntax analyzer should NOT produce any errors and should generate the parse tree.	Syntax analyzer should produce an Unexpected Characters Error and display the expected token.
Actual Output		
Remarks	PASSED	PASSED

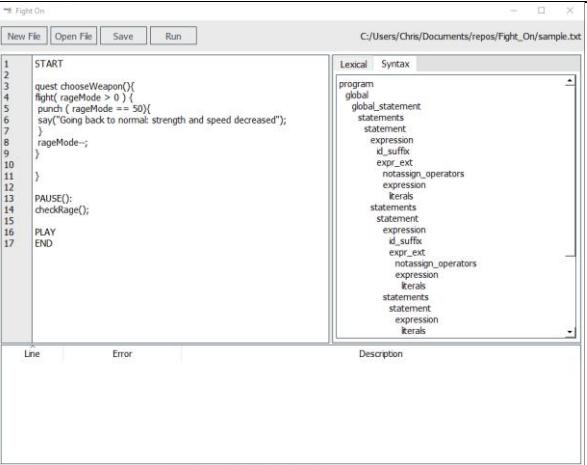
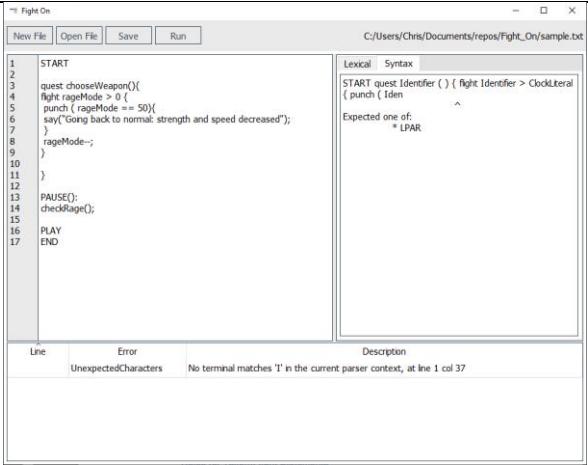
Rule	Potion case can only appear at the end of the bag statement.
------	--

<b>Sample Input</b>	<pre> START  quest chooseWeapon(){ bag ( weapon ) { item "sword" : say("Sword equipped"); quit; item "bow": say("Bow Equipped"); quit; item "arrow": say("Arrow Equipped"); quit; potion: say("Health potion used"); }  PAUSE(): chooseWeapon();  PLAY END </pre>	<pre> START  quest chooseWeapon(){ bag ( weapon ) { potion: say("Health potion used"); item "sword" : say("Sword equipped"); quit; item "bow": say("Bow Equipped"); quit; item "arrow": say("Arrow Equipped"); quit;  PAUSE(): chooseWeapon();  PLAY END </pre>
<b>Expected Output</b>	<p>Syntax analyzer should NOT produce any errors and should generate the parse tree.</p>	<p>Syntax analyzer should produce an Unexpected Characters Error and display the expected token.</p>
<b>Actual Output</b>		

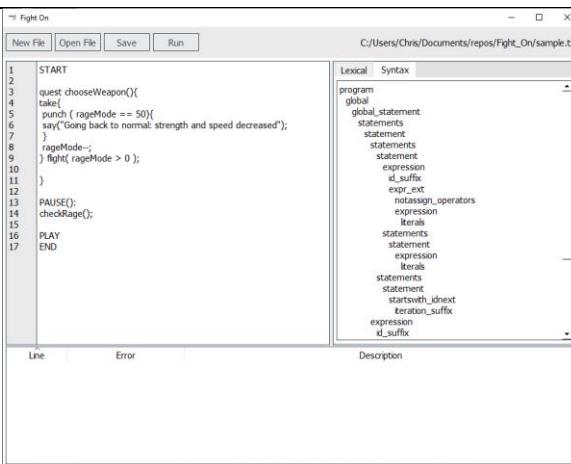
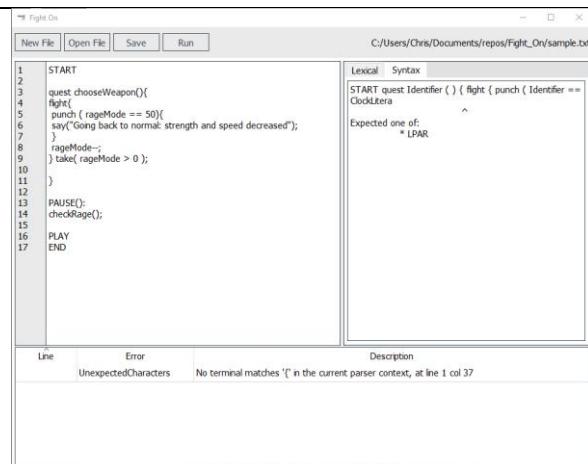
Remarks	PASSED	PASSED
---------	--------	--------

Rule	<b>Putting statements on initialization, condition, and iterator part of sprint statements are not required as long as the semicolons are present.</b>	
Sample Input	<pre> START  quest chooseWeapon(){ clock rageMode = 100; sprint( ; rageMode &gt; 0 ;){ punch ( rageMode == 50){ say("Going back to normal: strength and speed decreased"); } rageMode--; }  PAUSE(); checkRage();  PLAY END </pre>	<pre> START  quest chooseWeapon(){ clock rageMode = 100; sprint( ){ punch ( rageMode == 50){ say("Going back to normal: strength and speed decreased"); } rageMode--; }  PAUSE(); checkRage();  PLAY END </pre>
Expected Output	Syntax analyzer should NOT produce any errors and should generate the parse tree.	Syntax analyzer should produce an Unexpected Characters Error and display the expected token.
Actual Output	 <p>The screenshot shows a syntax analysis interface with two tabs: 'Lexical' and 'Syntax'. The 'Syntax' tab is active, displaying a parse tree for the provided code. The root node is 'program', which contains 'global', 'statements', and 'statements'. The first 'statements' node contains 'statement' (type 'expr'), 'vardec', 'cut', 'expression', and 'Iterals'. The second 'statements' node contains 'statement' (type 'expr'), 'expression', 'null', and 'expression'. The code itself is identical to the sample input above.</p>	 <p>The screenshot shows a syntax analysis interface with two tabs: 'Lexical' and 'Syntax'. The 'Syntax' tab is active, displaying an error message. It shows the first few lines of the code and highlights a closing brace '}' with a red box. To the right, a 'Expected one of:' list is shown, including tokens like 'HOVER', 'COIN', 'RUNE', 'IDENTIFIER', 'SEMICOLON', 'CLOCK', and 'RUNE'. Below the code, a 'Description' section states: 'UnexpectedCharacters No terminal matches '}' in the current parser context, at line 1 col 73'.</p>

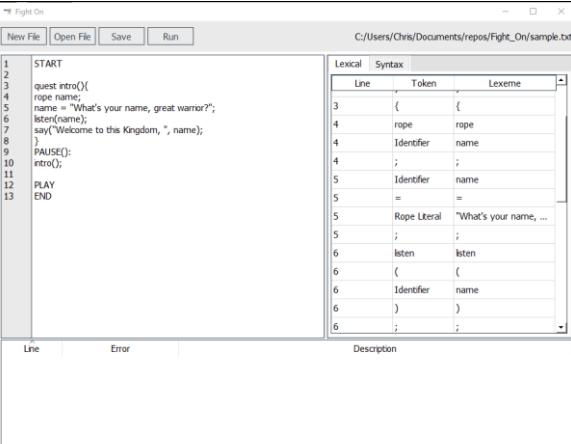
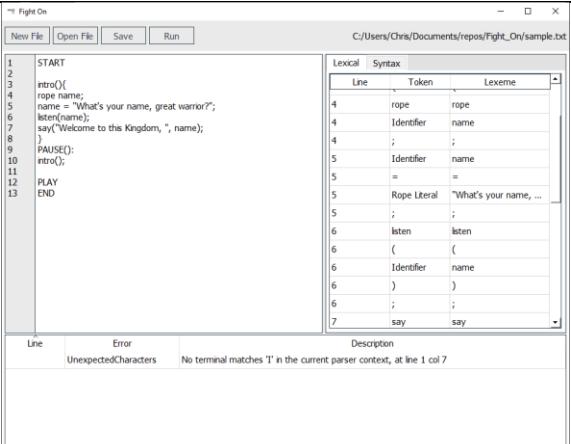
Remarks	PASSED	PASSED
---------	--------	--------

Rule	<b>Flight statements must always start with the reserved word flight, followed by parentheses that enclose the condition, then a pair of curly brackets ({}).</b>	
Sample Input	<pre> START  quest chooseWeapon(){ flight( rageMode &gt; 0 ) { punch ( rageMode == 50){ say("Going back to normal: strength and speed decreased"); } rageMode--; }  PAUSE(): checkRage();  PLAY END </pre>	<pre> START  quest chooseWeapon(){ flight rageMode &gt; 0 { punch ( rageMode == 50{ say("Going back to normal: strength and speed decreased"); } rageMode--; }  PAUSE(): checkRage();  PLAY END </pre>
Expected Output	Syntax analyzer should NOT produce any errors and should generate the parse tree.	Syntax analyzer should produce an Unexpected Characters Error and display the expected token.
Actual Output	 <p>The screenshot shows the Fight On IDE interface. The left pane displays the source code:</p> <pre> 1 START 2 quest chooseWeapon(){ 3   flight rageMode &gt; 0 { 4     punch ( rageMode == 50{ 5       say("Going back to normal: strength and speed decreased"); 6     } 7     rageMode--; 8   } 9 } 10 11 12 13 PAUSE(); 14 checkRage(); 15 16 PLAY 17 END </pre> <p>The right pane shows the lexical and syntax analysis results. The lexical pane lists tokens like 'program', 'global', 'statements', etc. The syntax pane shows a parse tree structure for the entire script.</p>	 <p>The screenshot shows the Fight On IDE interface. The left pane displays the source code, identical to the one above:</p> <pre> 1 START 2 quest chooseWeapon(){ 3   flight rageMode &gt; 0 { 4     punch ( rageMode == 50{ 5       say("Going back to normal: strength and speed decreased"); 6     } 7     rageMode--; 8   } 9 } 10 11 12 13 PAUSE(); 14 checkRage(); 15 16 PLAY 17 END </pre> <p>The right pane shows the lexical and syntax analysis results. The syntax pane highlights an error at line 1, column 37, indicating an unexpected character 'I' with the message 'No terminal matches 'I'' in the parser context.</p>

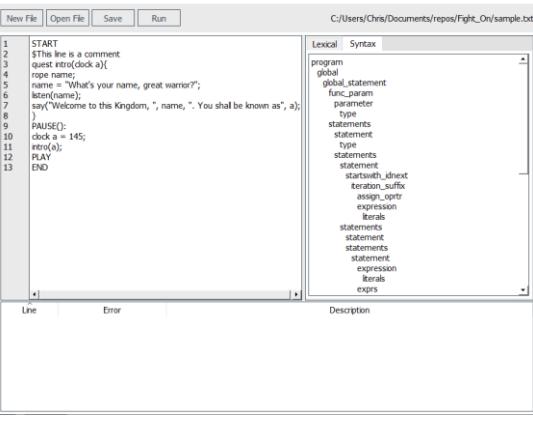
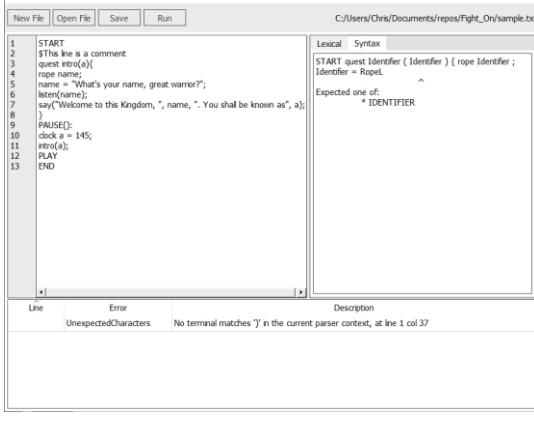
Remarks	PASSED	PASSED
---------	--------	--------

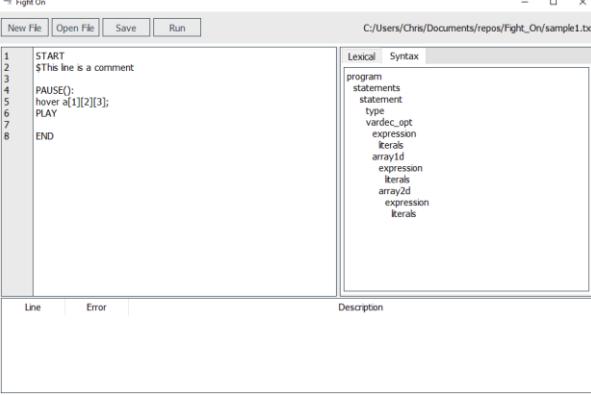
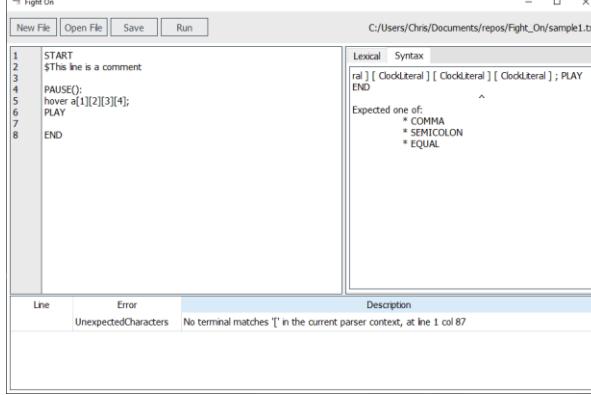
Rule	<b>Take..Flight statements always start with the reserved word take, followed by curly brackets ({} ) that encloses its body, then the flight's head.</b>	
Sample Input	<pre> START  quest chooseWeapon(){ take{ punch ( rageMode == 50){ say("Going back to normal: strength and speed decreased"); } rageMode--; } flight( rageMode &gt; 0 );  }  PAUSE(): checkRage();  PLAY END </pre>	<pre> START  quest chooseWeapon(){ flight{ punch ( rageMode == 50){ say("Going back to normal: strength and speed decreased"); } rageMode--; } take( rageMode &gt; 0 );  }  PAUSE(): checkRage();  PLAY END </pre>
Expected Output	Syntax analyzer should NOT produce any errors and should generate the parse tree.	Syntax analyzer should produce an Unexpected Characters Error and display the expected token.
Actual Output	 <p>The screenshot shows the Fight On IDE interface. The code editor contains the provided sample input. The Syntax tab displays a hierarchical parse tree for the code. The tree starts with a program node, which contains a global_statement node. This node has a statements child, which further contains statement nodes for 'quest', 'take', 'punch', 'say', 'rageMode--', and 'flight' blocks. The 'flight' block's body is enclosed in curly braces {}, indicating it is a Take..Flight statement.</p>	 <p>The screenshot shows the same IDE interface. The code editor contains the same sample input. In the Syntax tab, an error message is displayed: "START quest Identifier () { flight { punch ( Identifier == ClockUltra ^ Expected one of: * LPAR". Below the tabs, an "Error" table is shown with a single entry: "UnexpectedCharacters" at line 1 col 37, with the description "No terminal matches '(' in the current parser context, at line 1 col 37".</p>

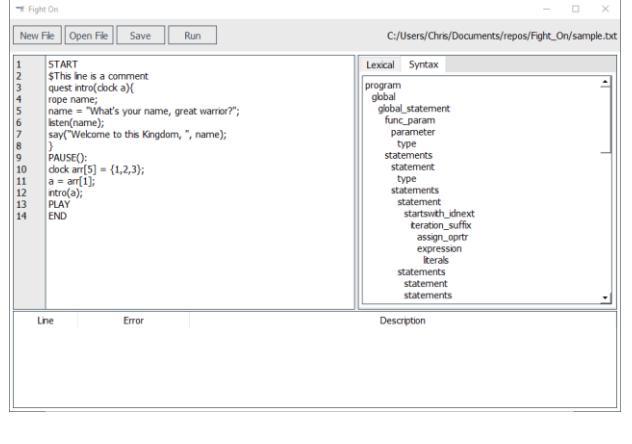
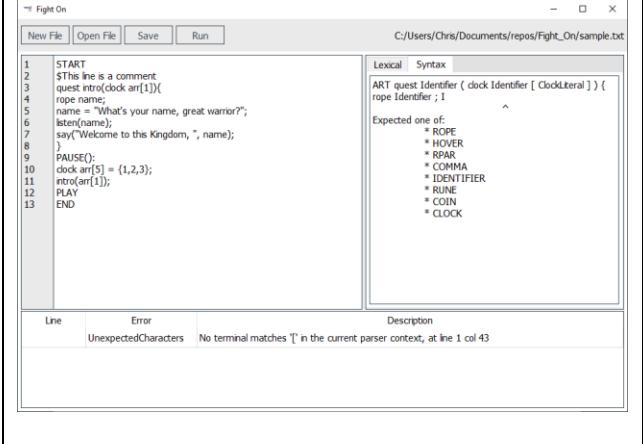
Remarks	PASSED	PASSED
---------	--------	--------

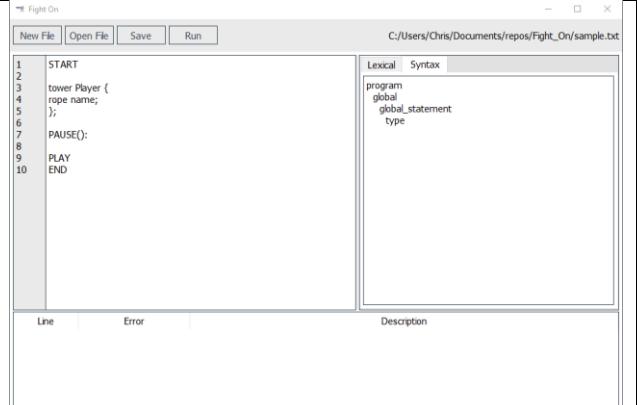
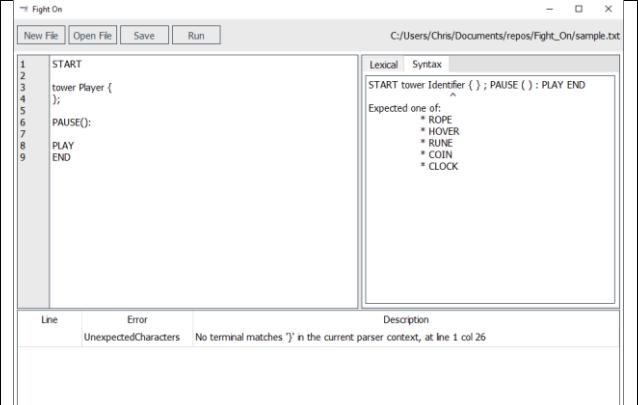
Rule	<b>When defining a function, it should start with the reserved word quest, followed by its identifier, then a pair of parentheses ()).</b>	
Sample Input	<p>START</p> <pre>quest intro(){ rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name); } PAUSE(): intro();</pre> <p>PLAY END</p>	<p>START</p> <pre>intro(){ rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name); } PAUSE(): intro();</pre> <p>PLAY END</p>
Expected Output	Syntax analyzer should NOT produce any errors and should generate the parse tree.	Syntax analyzer should produce an Unexpected Characters Error and display the expected token.
Actual Output		

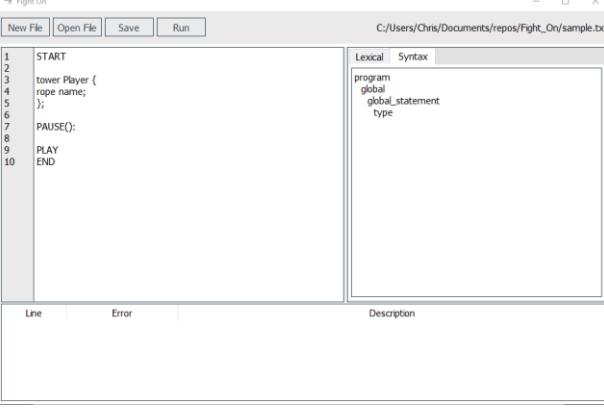
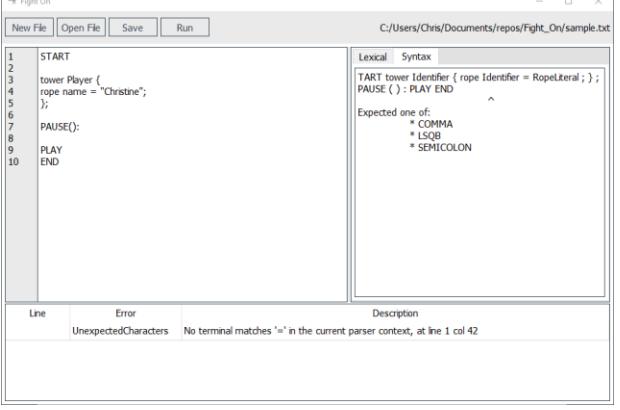
Remarks	PASSED	PASSED
---------	--------	--------

Rule	<b>Parameter list must obey the same declaration rules applied to variable declaration.</b>	
Sample Input	START \$This line is a comment quest intro(clock a){ rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name, ". You shall be known as", a); } PAUSE(): clock a = 145; intro(a); PLAY END	START \$This line is a comment quest intro(a){ rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name, ". You shall be known as", a); } PAUSE(): clock a = 145; intro(a); PLAY END
Expected Output	Syntax analyzer should NOT produce any errors and should generate the parse tree.	Syntax analyzer should produce an Unexpected Characters Error and display the expected token.
Actual Output	 <p>The screenshot shows the IDE interface with the file 'sample.txt' open. The lexical analysis pane shows tokens like 'START', 'quest', 'intro', 'clock', 'a', etc. The syntax analysis pane shows a tree structure for the program, global statements, function parameters, and so on.</p>	 <p>The screenshot shows the IDE interface with the file 'sample.txt' open. The lexical analysis pane shows tokens. The syntax analysis pane shows a parse tree that fails at the end of the parameter list, indicating an expected identifier was not found.</p>
Remarks	PASSED	

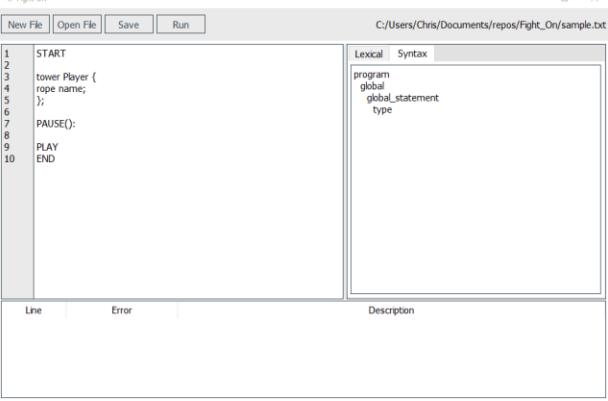
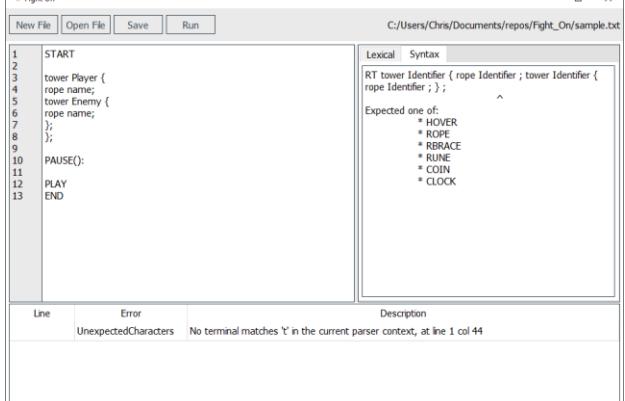
<b>Rule</b>	<b>The maximum dimension of an array is 3.</b>		
<b>Sample Input</b>	<pre>START \$This line is a comment  PAUSE(): hover a[1][2][3]; PLAY  END</pre>	<pre>START \$This line is a comment  PAUSE(): hover a[1][2][3][4]; PLAY  END</pre>	
<b>Expected Output</b>	Syntax analyzer should NOT produce any errors and should generate the parse tree.	Syntax analyzer should produce an Unexpected Characters Error and display the expected token.	
<b>Actual Output</b>	 		
<b>Remarks</b>	PASSED		
<b>Rule</b>	<b>A whole array is not allowed to be passed into a function</b>		

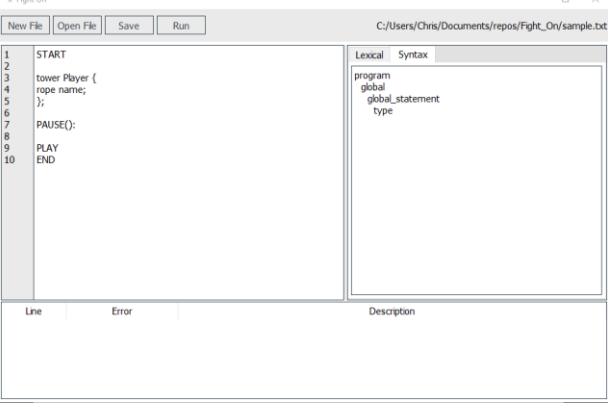
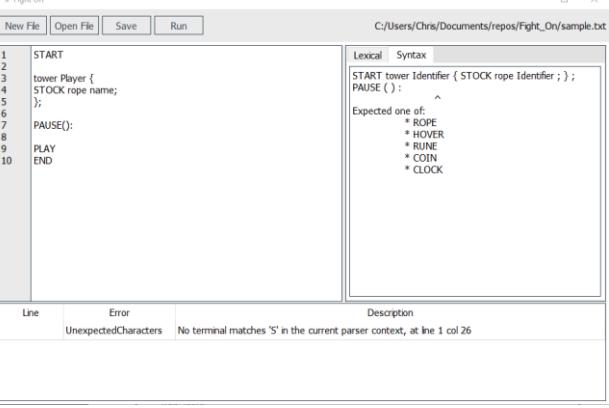
Sample Input	<pre> START \$This line is a comment quest intro(clock a){ rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name); } PAUSE(): clock arr[5] = {1,2,3}; a = arr[1]; intro(a); PLAY END </pre>	<pre> START \$This line is a comment quest intro(clock arr[1]){ rope name; name = "What's your name, great warrior?"; listen(name); say("Welcome to this Kingdom, ", name); } PAUSE(): clock arr[5] = {1,2,3}; intro(arr[1]); PLAY END </pre>
Expected Output	Syntax analyzer should NOT produce any errors and should generate the parse tree.	Syntax analyzer should produce an Unexpected Characters Error and display the expected token.
Actual Output	 <p>The screenshot shows a window titled 'Fight On' with tabs for 'Lexical' and 'Syntax'. The 'Syntax' tab is active, displaying a parse tree for the provided code. The root node is 'program', which contains 'global', 'statement', 'func_param', 'parameter', 'type', 'statements', and 'statement'. The 'statement' node under 'statements' contains 'start', 'idext', 'iteration-suffix', 'assign-optn', 'expression', 'literals', 'statement', and 'statements'. The code itself is identical to the sample input.</p>	 <p>The screenshot shows a window titled 'Fight On' with tabs for 'Lexical' and 'Syntax'. The 'Syntax' tab is active, displaying an error message. It shows the first few lines of the code and highlights the character ']' at line 1, column 4. A tooltip box lists expected tokens: ROPE, HOVER, PAUSE, COMMA, IDENTIFIER, RUNE, COIN, and CLOCK.</p>
Remarks	PASSED	PASSED
Rule	<p><b>All tower declarations must have at least one element.</b></p>	

Sample Input	<pre> START tower Player { rope name; };  PAUSE():  PLAY END </pre>	<pre> START tower Player { };  PAUSE():  PLAY END </pre>						
Expected Output	<p>Syntax analyzer should NOT produce any errors and should generate the parse tree.</p>	<p>Syntax analyzer should produce an Unexpected Characters Error and display the expected token.</p>						
Actual Output	 <p>The screenshot shows a window titled "Fight On" with tabs for "Lexical" and "Syntax". The "Syntax" tab displays a parse tree for the input code. The code consists of tokens: START, tower, Player, {, rope, name, ;, PAUSE, (), :, PLAY, END. The parse tree structure is as follows:</p> <pre> graph TD     START --&gt; tower     tower --&gt; Player     Player --&gt; brace1["{"]     brace1 --&gt; rope     brace1 --&gt; name     brace1 --&gt; semicolon1[";"]     semicolon1 --&gt; PAUSE     PAUSE --&gt; parentheses1["("]     parentheses1 --&gt; colon1[":"]     colon1 --&gt; PLAY     PLAY --&gt; END </pre> <p>Below the parse tree, there is a table with columns "Line", "Error", and "Description". The table is empty.</p>	 <p>The screenshot shows a window titled "Fight On" with tabs for "Lexical" and "Syntax". The "Syntax" tab displays an error message. The input code is the same as above. The error message is:</p> <pre> START tower Identifier { ; PAUSE ( ) : PLAY END ^ Expected one of: * ROPE * HOVER * RUNE * COIN * CLOCK </pre> <p>Below the error message, there is a table with columns "Line", "Error", and "Description". The table contains one row:</p> <table border="1"> <thead> <tr> <th>Line</th> <th>Error</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td></td> <td>UnexpectedCharacters</td> <td>No terminal matches '}' in the current parser context, at line 1 col 26</td> </tr> </tbody> </table>	Line	Error	Description		UnexpectedCharacters	No terminal matches '}' in the current parser context, at line 1 col 26
Line	Error	Description						
	UnexpectedCharacters	No terminal matches '}' in the current parser context, at line 1 col 26						
Remarks	PASSED	PASSED						
Rule	<p><b>Each element must only have its data type and identifier. It cannot be initialized.</b></p>							

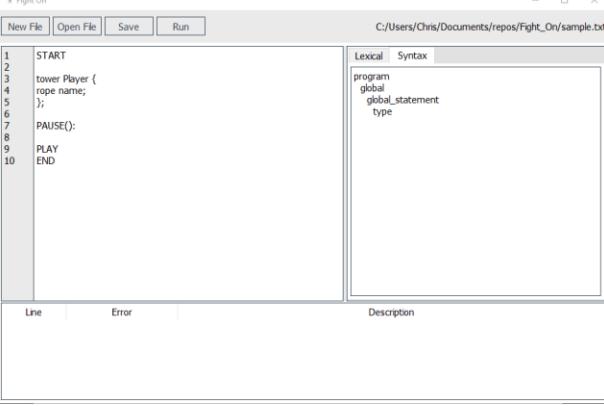
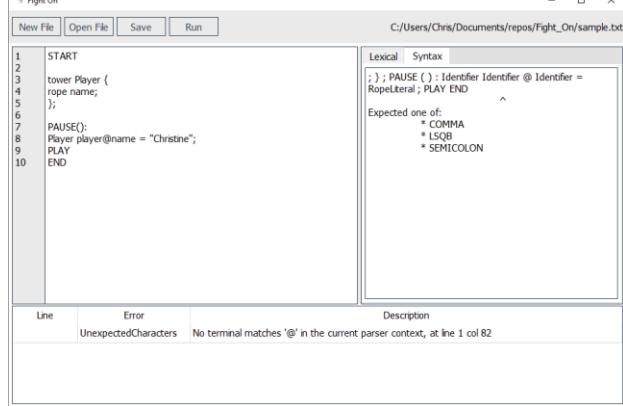
Sample Input	<pre> START tower Player { rope name; };  PAUSE():  PLAY END </pre>	<pre> START tower Player { rope name = "Christine"; };  PAUSE():  PLAY END </pre>
Expected Output	<p>Syntax analyzer should NOT produce any errors and should generate the parse tree.</p>	<p>Syntax analyzer should produce an Unexpected Characters Error and display the expected token.</p>
Actual Output	 <p>The screenshot shows a window titled 'Fight On' with tabs for 'New File', 'Open File', 'Save', and 'Run'. The file path is 'C:/Users/Chris/Documents/repos/Fight_On/sample.txt'. The code input area contains:</p> <pre> 1 START 2 tower Player { 3    rope name; 4 }; 5 PAUSE(); 6 7 PLAY 8 END </pre> <p>The 'Syntax' tab displays tokens: program, global, global_statement, type. The 'Description' tab is empty.</p>	 <p>The screenshot shows a window titled 'Fight On' with tabs for 'New File', 'Open File', 'Save', and 'Run'. The file path is 'C:/Users/Chris/Documents/repos/Fight_On/sample.txt'. The code input area contains:</p> <pre> 1 START 2 tower Player { 3    rope name = "Christine"; 4 }; 5 PAUSE(); 6 7 PLAY 8 END </pre> <p>The 'Syntax' tab displays tokens: TART, tower, Identifier, {, rope, Identifier, =, RopeLiteral, ;, PAUSE, (, ), PLAY, END. The 'Description' tab shows an error: 'Expected one of: * COMMA * LSQB * SEMICOLON'. Below the tabs, it says 'Line 1 Error UnexpectedCharacters No terminal matches '=' in the current parser context, at line 1 col 42'.</p>
Remarks	PASSED	PASSED

Rule	<p><b>A structure cannot contain another structures, no nested tower (structure).</b></p>
------	---

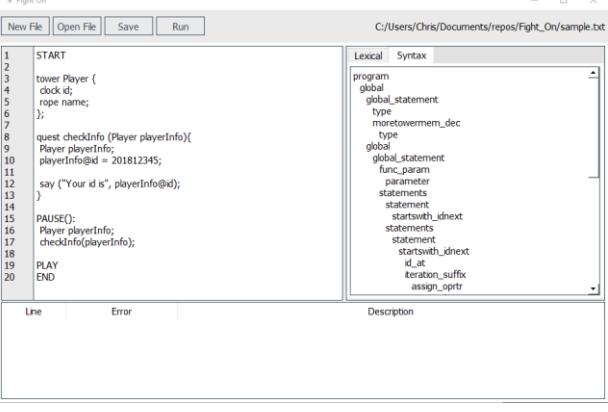
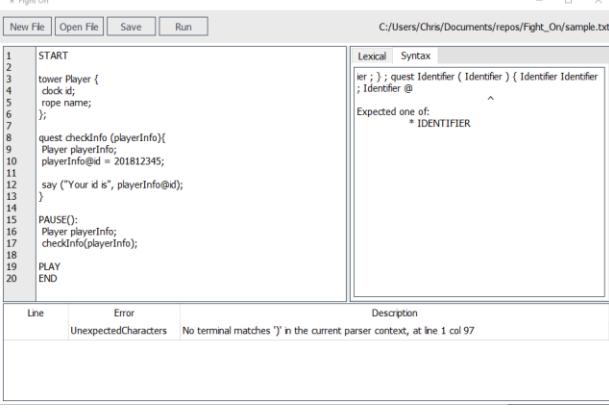
Sample Input	<pre> START  tower Player { rope name; };  PAUSE():  PLAY END </pre>	<pre> START  tower Player { rope name; tower Enemy { rope name; }; };  PAUSE():  PLAY END </pre>
Expected Output	Syntax analyzer should NOT produce any errors and should generate the parse tree.	Syntax analyzer should produce an Unexpected Characters Error and display the expected token.
Actual Output		
Remarks	PASSED	PASSED
Rule	<p><b>A tower member cannot be constant.</b></p>	

<b>Sample Input</b>	<pre> START tower Player { rope name; };  PAUSE():  PLAY END </pre>	<pre> START tower Player { STOCK rope name; };  PAUSE():  PLAY END </pre>
<b>Expected Output</b>	<p>Syntax analyzer should NOT produce any errors and should generate the parse tree.</p>	<p>Syntax analyzer should produce an Unexpected Characters Error and display the expected token.</p>
<b>Actual Output</b>		
<b>Remarks</b>	PASSED	PASSED

<b>Rule</b>	<p><b>Initialization of tower member is during tower variable declaration is not allowed.</b></p>
<b>Sample Input</b>	<pre> START tower Player { rope name; };  PAUSE():  PLAY END </pre>

<b>Expected Output</b>	Syntax analyzer should NOT produce any errors and should generate the parse tree.	Syntax analyzer should produce an Unexpected Characters Error and display the expected token.
<b>Actual Output</b>	 <p>The screenshot shows the 'Lexical' and 'Syntax' tabs. The Lexical tab lists tokens: program, global, global_statement, type. The Syntax tab shows a parse tree structure. Below the tabs is a table with columns 'Line', 'Error', and 'Description'.</p>	 <p>The screenshot shows the 'Lexical' and 'Syntax' tabs. The Lexical tab lists tokens: program, global, global_statement, type. The Syntax tab shows a parse tree structure. Below the tabs is a table with columns 'Line', 'Error', and 'Description'. The 'Error' column contains 'UnexpectedCharacters' and the 'Description' column contains 'No terminal matches '@' in the current parser context, at line 1 col 82.'</p>
<b>Remarks</b>	PASSED	PASSED

<b>Rule</b>	Initialization of tower member is during tower variable declaration is not allowed.
<b>Sample Input</b>	<pre> START tower Player {     clock id;     rope name; };  quest checkInfo (Player playerInfo){     Player playerInfo;     playerInfo@id = 201812345;      say ("Your id is", playerInfo@id); }  PAUSE():     Player playerInfo;     checkInfo(playerInfo);  PLAY END </pre> <pre> START tower Player {     clock id;     rope name; };  quest checkInfo (playerInfo){     Player playerInfo;     playerInfo@id = 201812345;      say ("Your id is", playerInfo@id); }  PAUSE():     Player playerInfo;     checkInfo(playerInfo);  PLAY END </pre>

<b>Expected Output</b>	Syntax analyzer should NOT produce any errors and should generate the parse tree.	Syntax analyzer should produce an Unexpected Characters Error and display the expected token.
<b>Actual Output</b>	 <p>The screenshot shows a window titled "Fight On" with tabs for "Lexical" and "Syntax". The code area contains a valid script:</p> <pre> 1 START 2 tower Player { 3   clock id; 4   rope name; 5 } 6 7 quest checkInfo (Player playerInfo){ 8   Play playerInfo; 9   playerInfo@id = 201812345; 10 11   say ("Your id is", playerInfo@id); 12 } 13 14 PAUSE(); 15 Player playerInfo; 16 checkInfo(playerInfo); 17 18 PLAY 19 END </pre> <p>The "Syntax" tab displays a parse tree structure for the code.</p>	 <p>The screenshot shows a window titled "Fight On" with tabs for "Lexical" and "Syntax". The code area contains an invalid script:</p> <pre> 1 START 2 tower Player { 3   clock id; 4   rope name; 5 } 6 7 quest checkInfo (playerInfo){ 8   Play playerInfo; 9   playerInfo@id = 201812345; 10 11   say ("Your id is", playerInfo@id); 12 } 13 14 PAUSE(); 15 Player playerInfo; 16 checkInfo(playerInfo); 17 18 PLAY 19 END </pre> <p>The "Syntax" tab shows an error message: "Expected one of: * IDENTIFIER".</p>
<b>Remarks</b>	PASSED	PASSED