Table of Contents

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 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 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
anIdentifier	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 AND x < 10	Returns false if any of the two entities are false.
OR	Logical OR	x < 5 OR x < 4	Returns true as long as one of the entities is found to be true.
NOT	Logical NOT	NOT(x < 5 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
>=	Greater than or Equal to	x >= y	Used to say that one entity is either greater than or equal to another.
<=	Less than or Equal to	x <= 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.
~	Unary negation	~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 Division Modulus	Left-to-Right
6	+ -	Addition Subtraction	Left-to-Right
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

a. Multiplicative (* / %) and additive operators(+ -) are used as infix operators. In other words, they operate on two operands and are always in between operands.
b. Only clock (integer) and hover (decimal) values may be used in arithmetic expressions.
c. Arithmetic expressions may be followed by an open parenthesis.
d. 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

a. It must always start with a variable, followed by an assignment operator (=), then an operand.
b. An operand can be literal, a constant, a variable, and other expressions.
c. Same variable can hold different values at different instants of time based on its scope.
d. 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

a. Relational operators are used as infix operators .In other words, they operate on two operands and always are always in between operands.
b. The operands being compared should be of the same data type.
c. An operand can be literal, a constant, a variable or other expression except the assignment expression.
d. 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
<operand> <logical operator> <operand> *where: <operand> = <relational expression>* *<coin literal>* *<identifier>*

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
<operand> <unary operator>

Valid Unary Expression	Invalid Unary Expression	Reason
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>

Valid Unary Negative Expression	Invalid Unary Negative Expression	Reason
~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 must 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 happened 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.
6. 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 {
int 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)^{9}	clocklit
hover literal	(~ / λ) (nonzero) (number / λ)^{9} (.) (number / λ)^{5}	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)^{24}	id

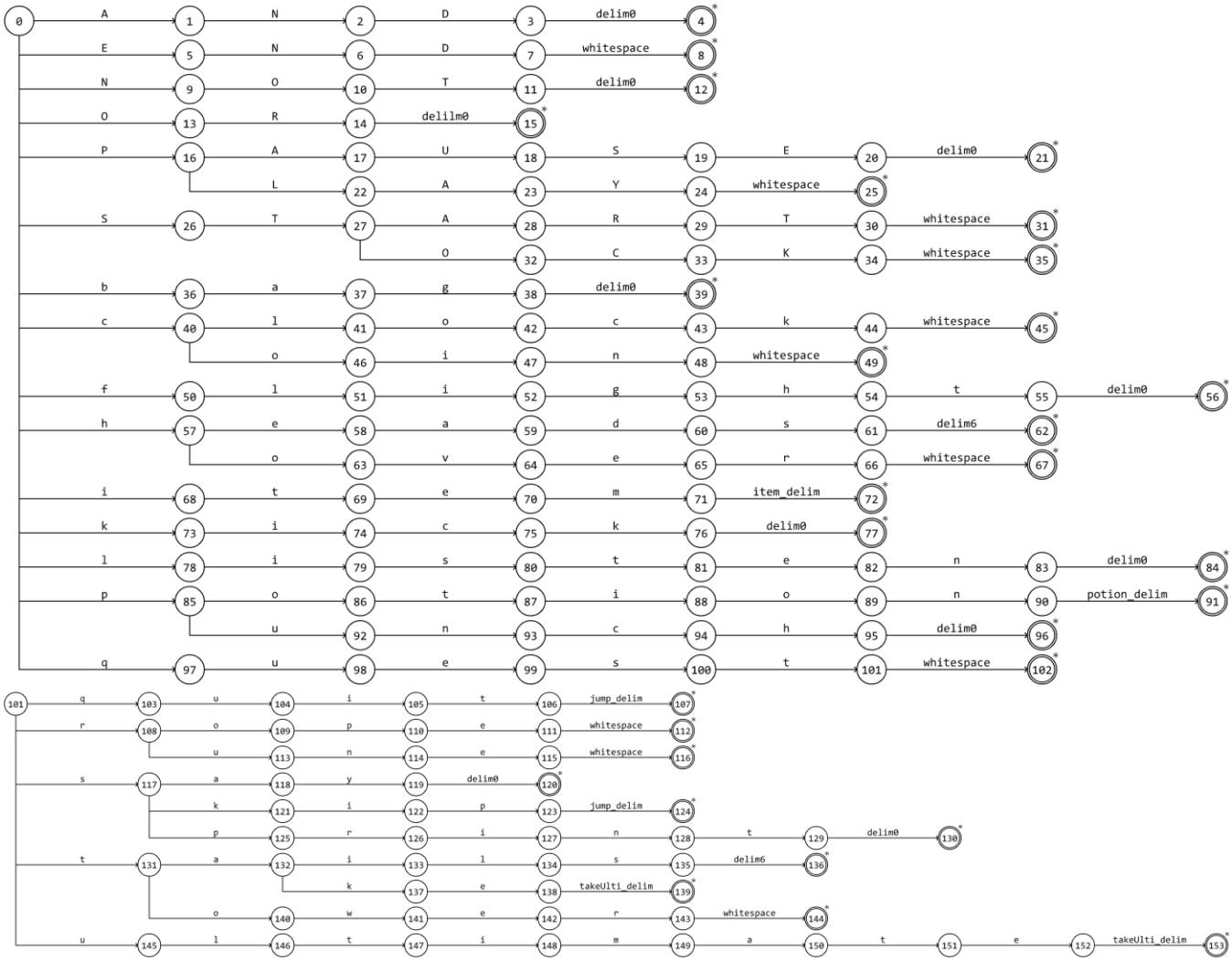
IX. REGULAR DEFINITION

Delimiter	Value
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_alpanumeric, delim0, ~
op_delim	delim5,), ;
cp_delim	delim2, {, }, !, ,, delim7, =, colons
ocb_delim	alphanumeric, whitespace, {, }, ~, (
ccb_delim	}, ,, jump_delim
osb_delim	alphanumeric, whitespace, [

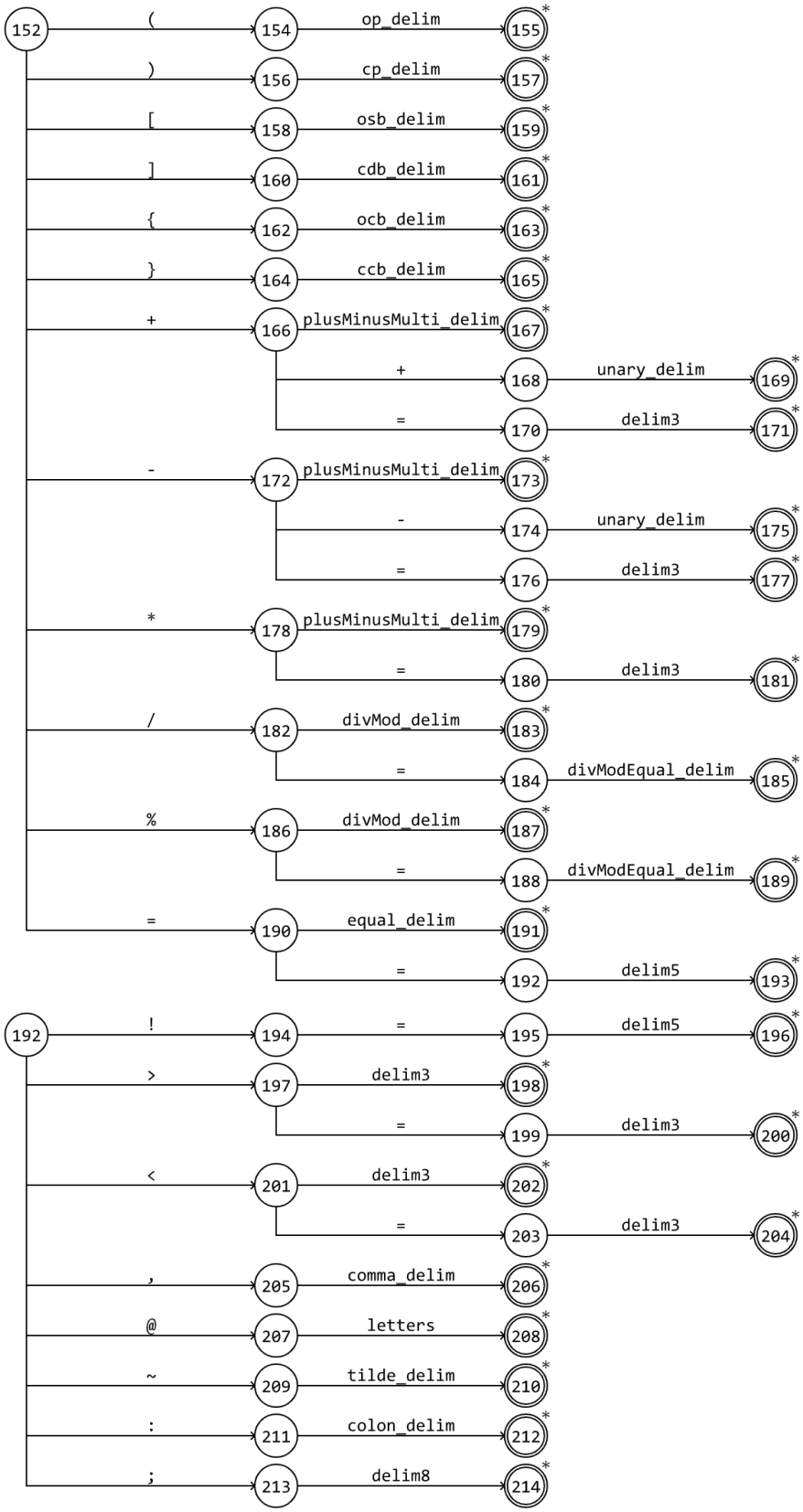
csb_delim	;, }, @, [, not_equal, whitespace, arithmetic, angle_brackets, delim2, ccb_comma
plusMinusMulti_delim	delim0, alphanumeric, ~
divMod_delim	delim0, nonzero_alpanumeric, ~
comma_delim	delim1, {, quote, ~
identifier_delim	tilde_delim
clock_delim	alpanumeric, quote, takeUlti_delim
hover_delim	delim2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

X. TRANSITION DIAGRAM

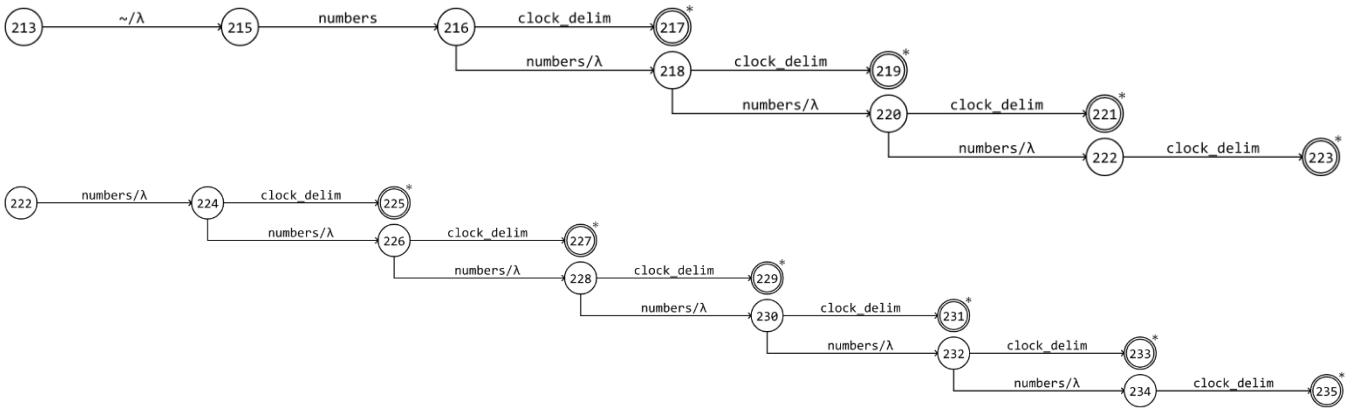
A. Reserved Words



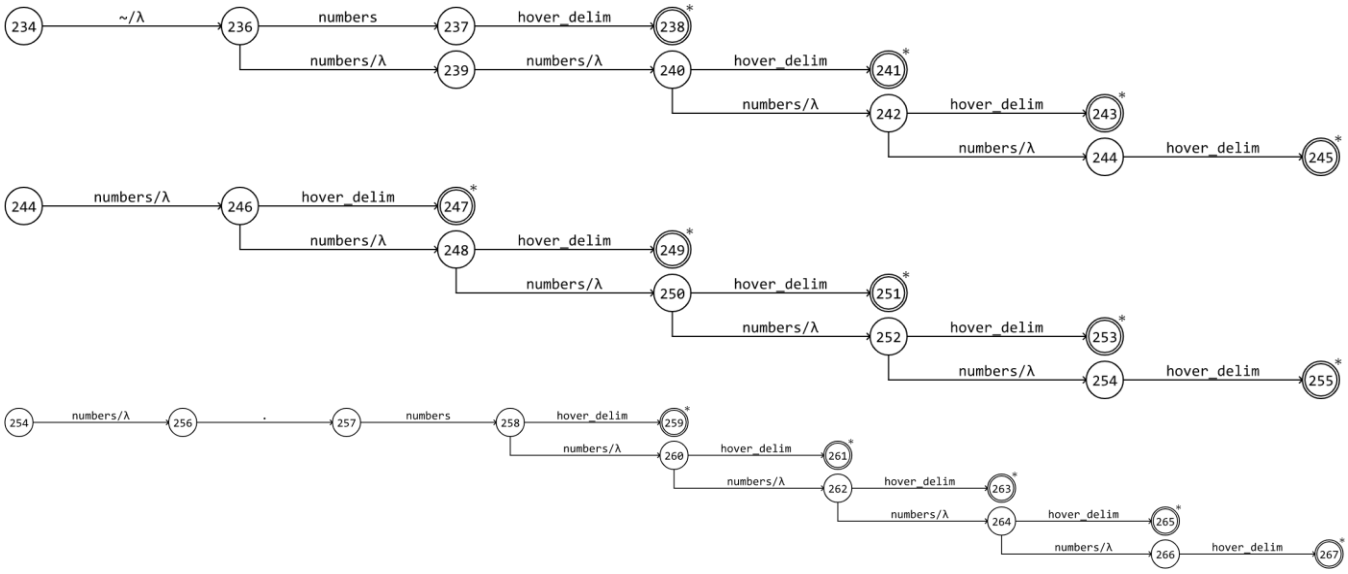
B. Reserved Symbols



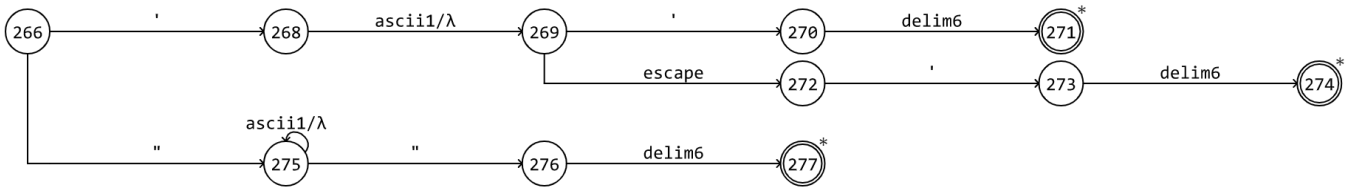
C. Clock Literal



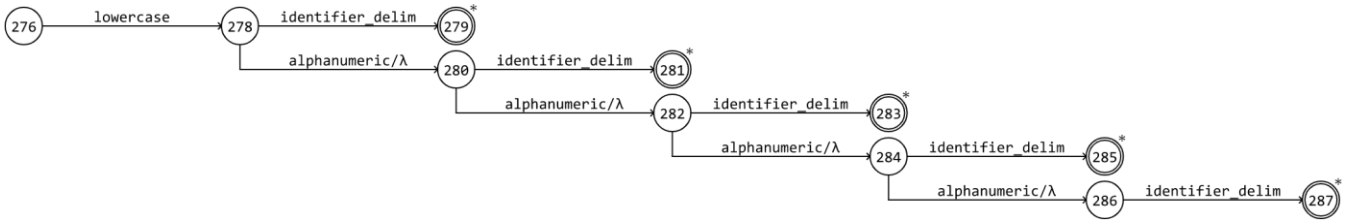
D. Hover Literal

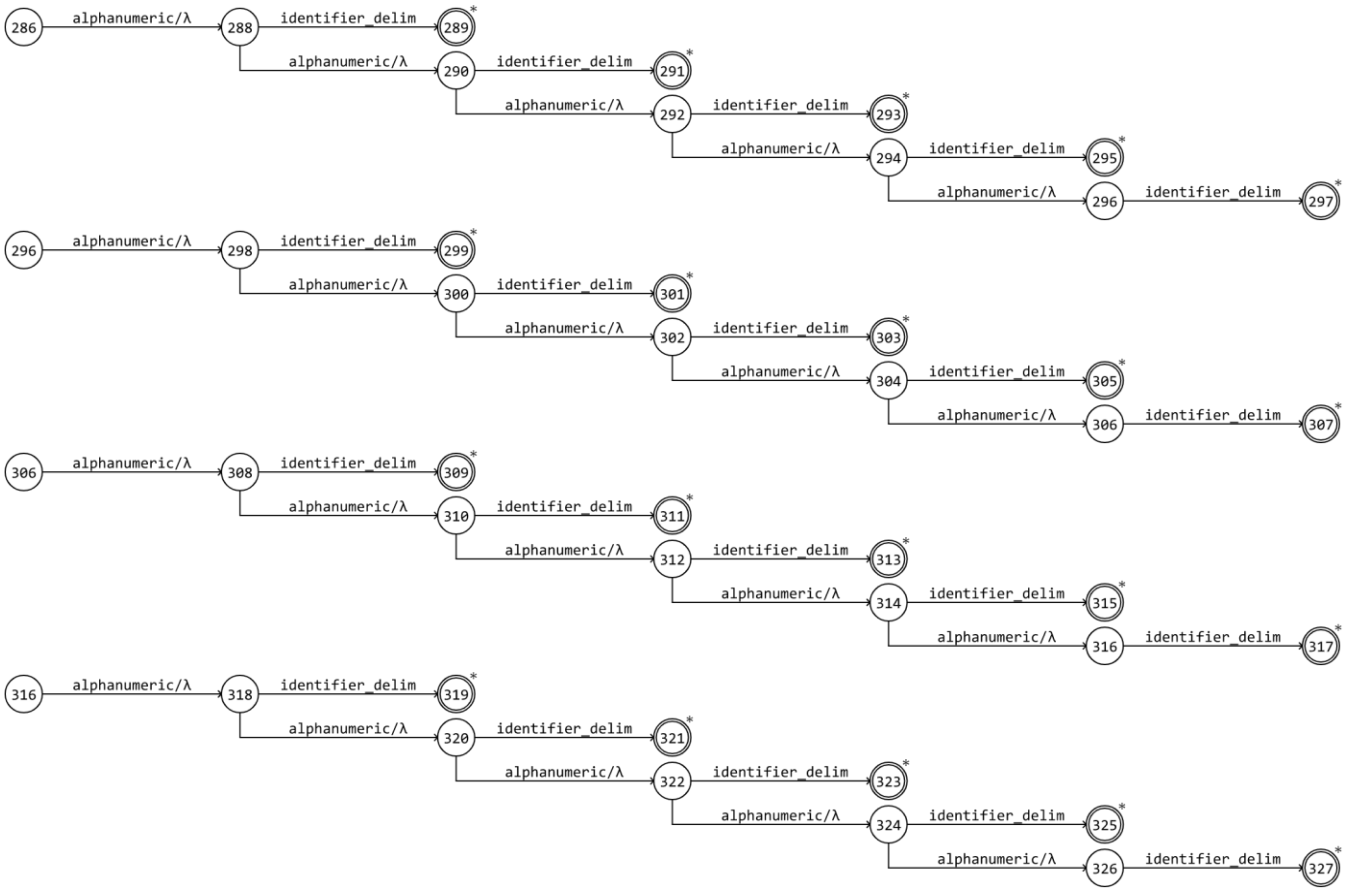


E. Rune and Rope Literal

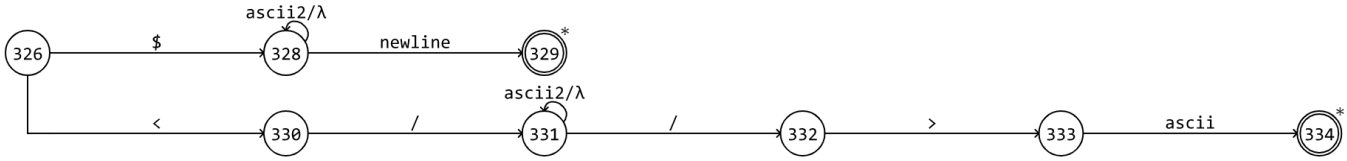


F. Identifiers




286
alphanumeric/λ
288
identifier_delim
289
alphanumeric/λ
290
identifier_delim
291
alphanumeric/λ
292
identifier_delim
293
alphanumeric/λ
294
identifier_delim
295
alphanumeric/λ
296
identifier_delim
297
296
alphanumeric/λ
298
identifier_delim
299
alphanumeric/λ
300
identifier_delim
301
alphanumeric/λ
302
identifier_delim
303
alphanumeric/λ
304
identifier_delim
305
alphanumeric/λ
306
identifier_delim
307
306
alphanumeric/λ
308
identifier_delim
309
alphanumeric/λ
310
identifier_delim
311
alphanumeric/λ
312
identifier_delim
313
alphanumeric/λ
314
identifier_delim
315
alphanumeric/λ
316
identifier_delim
317
316
alphanumeric/λ
318
identifier_delim
319
alphanumeric/λ
320
identifier_delim
321
alphanumeric/λ
322
identifier_delim
323
alphanumeric/λ
324
identifier_delim
325
alphanumeric/λ
326
identifier_delim
327

G. Comments


ascii2/λ
326
$
328
newline
329
ascii2/λ
<
330
/
331
/
332
>
333
ascii
334

XI. CONTEXT-FREE GRAMMAR

#	Production	Production Rule
1	<program>	START <global> PAUSE ( ) : <statements> PLAY END
2	<global>	<global_statement> <global>
3	<global>	λ
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>	λ
40	<expr>	<expression> <exprs>

41	<expr>	λ
42	<exprs>	, <expression> <exprs>
43	<exprs>	λ
44	<morestock_dec>	, id <stock_dec> <morestock_dec>
45	<morestock_dec>	λ
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>	λ
57	<morestock_Arr2D>	, <stock_array2D> <morestock_Arr2D>
58	<morestock_Arr2D>	λ
59	<moretowerMem>	, id <array1D_ind> <moretowerMem>
60	<moretowerMem>	λ
61	<moretowerMem_dec>	<type> id <array1D_ind> <moretowerMem> ; <moretowerMem_dec>
62	<moretowerMem_dec>	λ
63	<array1D_ind>	[<expression>] <array2D_ind>
64	<array1D_ind>	λ
65	<array2D_ind>	[<expression>] <array3D_ind>
66	<array2D_ind>	λ
67	<array3D_ind>	[<expression>]
68	<array3D_ind>	λ
69	<expression>	<literals> <expr_ext>
70	<expression>	id <id_suffix> <expr_ext>
71	<expression>	(<expression>) <expr_ext>
72	<expression>	NOT <expression>
73	<expression>	~ <expression>
74	<expr_ext>	<notAssign_operators> <expression>
75	<expr_ext>	λ
76	<id_suffix>	(<expr>)
77	<id_suffix>	<array1D_ind> <id_at> <unary_oprtr>
78	<id_at>	@ id <array1D_ind>
79	<id_at>	λ
80	<vardec_opt>	= <expression>
81	<vardec_opt>	[<expression>] <array1D>
82	<vardec_opt>	λ
83	<morevardec_opt>	, id <vardec_opt> <morevardec_opt>
84	<morevardec_opt>	λ
85	<array1D>	[<expression>] <array2D>
86	<array1D>	= <stock_array1D>
87	<array1D>	λ
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_statement>
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	<statement>	{ 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

Production Rule	START	PAUSE	(	)	:	PLAY	END	STOCK	id	;	tower	{
<program>	1											
<global>		3						2			2	
<global_statement>								4			5	
<type>												
<literals>												
<assign_oprtr>												
<notAssign_operators>												
<unary_oprtr>				39						39		
<expr>			40	41					40			
<exprs>				43								
<morestock_dec>										45		
<stock_dec>												
<stock_arrayOpt1>												
<stock_arrayOpt2>												
<stock_array1D>												52
<stock_array2D>												53
<stock_array3D>												54
<morestock_Arr1D>												
<morestock_Arr2D>												
<moretowerMem>										60		
<moretowerMem_dec>												
<array1D_ind>				64						64		
<array2D_ind>				66						66		
<array3D_ind>				68						68		
<expression>			71						70			
<expr_ext>				75						75		
<id_suffix>			76	77						77		
<id_at>				79						79		
<vardec_opt>										82		
<morevardec_opt>										84		
<array1D>										87		
<array2D>										90		
<array3D>										92		
<func_param>				94					93			
<parameter>									96			
<parameters>				98								
<statements>						100		99	99			
<statement>								101				
<statement>			115						103			
<startswith_idNext>									114			
<initialization>			120						118	119		
<expression_null>									120	121		
<iteration>				123					122			
<iteration_suffix>												
<kick_statement>						128		128	128			
<ultimate_statement>						130		130	130			
<item_statement>												
<potion_statement>												

}	quest	clock	hover	coin	rune	rope	clocklit	hoverlit	coinlit	runelit	ropelit	=
	2	2	2	2	2	2						
	7	6	6	6	6	6						
		8	9	10	11	12						
							13	14	15	16	17	
												18
39												
41							40	40	40	40	40	
43												
												46
												48
												50
56												
58												
62		61	61	61	61`	61						
64												64
66												66
68												68
							69	69	69	69	69	
75												
79												79
												80
												86
												89
												91
		93	93	93	93	93						
		95	95	95	95	95						
100		99	99	99	99	99						
		102	102	102	102	102						
												116
		117	117	117	117	117						
							120	120	120	120	120	
												126
128		128	128	128	128	128						
130			130	130	130	130						
132												
134												

+=	-=	*=	/=	%=	+	-	*	/	%	AND	OR	<	>	>=	<=	==	!=	++	--
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		
					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	78, 79	79	79	79	79	79	79
116	116	116	116	116									116					116	
126	126	126	126	126														124	

,	[	]	NOT	~	@	sprint	quit	skip	bag	flight	take	punch	say	listen
39		39												
			40	40										
42														
44														
	47													
	49													
	51													
55														
57														
59														
64	63	64												
66	65	66												
68	67	68												
			72	73										
75		75												
77	77	77												
79		79												
82	81													
83														
87	85													
90	88													
92														
97														
						99	99	99	99	99	99	99	99	99
						104	105	106	107	108	109	110	111	112
	116													
			120	120										
						128	128	128	128	128	128	128	128	128
						130	130	130	130	130	130	130	130	130

<id_at>	return	–	kick	<expression>	ultimate	item	potion
64		64					
66		66					
68		68					
		79					
	99					100	100
	113						
		116					
		125					
	128				128	128	128
	130				129	130	130
						131	133
							133