İhsan Doğramacı Bilkent University

Department of Computer Engineering

CS315 - PROGRAMMING LANGUAGES

Project 1

Language Name: Egg

Sebahattin Utku Sezer - 21802798

Gökhan Taş - 21802136

Bedirhan Sakinoğlu - 21802576

Instructor: H. Altay Güvenir

Teaching Assistants: Irmak Türköz, Duygu Durmuş

Content

BNF Description of Egg Language

1. Initial Program

```
<program> ::= begin <stmts> end

<stmts> ::= <stmt> | <stmt> <semicolon> <stmts>

<stmt> ::= <matched_stmt> <end_stmt>
         | <unmatched_stmt> <end_stmt>
         | <non_if_stmt> <end_stmt>

<end_stmt> ::= <semicolon>
```

2. Variable Identifier

```
<term> ::= <var> | <constant>

<var> ::= <identifier>

<identifier> ::= <word> | <word> <identifier>
               | <identifier> <digit>

<constant> ::= <constant_identifier> <var>

<constant_identifier> ::= const
```

3. Types

```
<types> ::= bool | int | string | double | char

<assignment_values> ::= <bool_stmt> | <int_stmt>
                      | <string_stmt> | <double_stmt>
                      | <char_stmt>

<bool_stmt> ::= <true_stmt> | <false_stmt>


<int_stmt> ::= <sign> <int_stmt> | <int_stmt> <digit>
```

```
              | <digit>

<string_stmt> ::= <string_identifier> <sentence>
<string_identifier>

<double_stmt> ::= <int_stmt> <dot> <digits> | <int_stmt>

<char_stmt> ::= <char_identifier> <digit> <char_identifier>

                | <char_identifier> <character>
     <char_identifier>
```

4. Assignment Operator

```
<assignment_operator> ::= <assign>
```

5. Declaration and Initialization

```
<declaration> ::= <types> <term> <semicolon>

<initialization> ::= <term> <assignment_operator>
<assingment_value> <semicolon>

<declaration_and_initilization> ::= <types> <term>
<assignment_operator> <assignment_value> <semicolon>
```

6. Boolean values

```
<true_stmt> ::= true

<false_stmt> ::= false
```

7. Expressions

```
<factor> ::= <term>

<max> ::= max <LP> <int_stmt> <comma> <int_stmt> <RP>

        | max <LP> <double_stmt> <comma> <double_stmt> <RP>

<min> ::= min <LP> <int_stmt> <comma> <int_stmt> <RP>

        | min <LP> <double_stmt> <comma> <double_stmt> <RP>
```

8. Precedence, Associativity of The Operators

```
<arithmetic_operation> ::= <addition> | <subtraction>
                         | <division> | <multiplication>
                         | <modulo> | <power>
<addition> ::= <addition> + <term> | <term> | ( <addition>)
<subtraction> ::= <subtraction> - <term> | <term> |
(<subtraction>)
<division> ::= <division> / <factor> | <factor> | (<division>)
<multiplication> ::= <multiplication> * <factor> | <factor>
                   | ( <multiplication>)
<modulo> ::= <factor> % <factor>
<power> ::= pow <LP> <int_stmt> <comma> <int_stmt> <RP>
          | pow <LP> <int_stmt> <comma> <double_stmt> <RP>
          | pow <LP> <double_stmt> <comma> <int_stmt> <RP>
          | pow <LP> <double_stmt> <comma> <double_stmt> <RP>
```

9. Loops

```
<loops> ::= <while_loops> | <for_loop> | <do_while_loop>
<while_loop> ::= while <LP> <logical_expression> <RP>
<LCB><stmts> <RCB>
<for_loop> ::= for <LP> <loop_initialization>
               <semicolon><logical_expression> <semicolon>
               <arithmetic_operation> <RP> <LCB> <stmts> <RCB>
<do_while_loop> ::= do <LCB> <stmts> <RCB> while <LP>
                    <logical_expression> <RP> <semicolon>
<loop_initialization> ::= <initialization>
                        | <declaration_and_initilization>
```

10. Conditional statements

```
<matched_stmt> ::= if (<logical_expression>) <stmt>
                 | elif (<logical_expression>) <stmt>
                 | <non_if_statement>
<unmatched_stmt> ::= if (<logical_expression>) <stmt>
          | elif (<logical_expression>)<stmt>
          | if(<logical_expression>)<matched> else <unmatched>
<non_if_stmt> ::= <loops> <non_if_stmt>
                | <arithmetic_operations>  <non_if_stmt>
                | <function_call> <non_if_stmt>
                | <function_declaration><non_if_stmt>
                | <declaration> <non_if_stmt>
                | <initialization> <non_if_stmt>
                | <declaration_and_initilization><non_if_stmt>
                | <input_stmt> <non_if_stmt>
                | <output_stmt> <non_if_stmt>
                | <comment> <non_if_stmt>
                | <double_comment> <non_if_stmt>
                | <loops>
                | <arithmetic_operations>
                | <function_call>
                | <function_declaration>
                | <declaration>
                | <initialization>
                | <declaration_and_initilization>
                | <input_stmt>
                | <output_stmt> | <comment> | <double_comment>
```

```
<logical_expression> ::= <single_expression>

                       | <recursive_expression>

<single_expression> ::=  <term> <logical_operator> <term>

                       | <bool_stmt>

                       | <not> <bool_stmt>

<recursive_expression> ::= <recursive_expression>
<logical_connector> <single_expression>

                         | <single_expression>

<logical_operator> ::= <equal> | <not_equal> | <not>

                     | <greater_than> | <less_than>

                     | <greater_or_equal> | <less_or_equal>

<logical_connector> ::= <and> | <or>
```

11. Statements For Input / Output

```
<input_stmt> ::= eggin <LP> <input_context> <RP>

<output_stmt> ::= eggout <LP> <output_context> <RP>

<input_context> ::= <term> + <input_context> | <term>

<output_context> ::= <term> | <assignment_values>

                   | <term> + <output_context>

                   | <assignment_values> + <output_context>
```

12. Function Definitions and Function Calls

```
<function_types> ::= <types> | void

<parameter> ::= <parameter>, <types> <identifier>

              | <types> <identifier> | <empty>
```

```
<function_declaration> ::= <function_types> funct_<function_name>
( <parameter> ) {<stmts>}

          | <function_types> funct_<function_name> ( <parameter> )
          {<stmts> return <assignment_values> }

          | <function_types> funct_<function_name> (<parameter> )
          {<stmts> return void}

<function_call> ::= funct_<function_name> (<identifier>);
```

13. Symbols

```
<sentence> ::= <word> | <digit> | <digit> <sentence>

             | <sentence> <word> | <space>

             | <sentence> <space> | <sentence> <digit>

             | <sentence> <symbol> | <symbol>

<word> ::= <lower_characters> | <lower_characters> <word>

         | <lower_characters> <character>

         | <lower_characters> <digits>

         | <lower_characters> <under_score>

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<digits> ::= <digit> <digits> | <digits>

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

<character> ::= 'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|
'k'|'l'|'m'|'n'|'o'|'p'|'q'|'r'|'s'|'t'|'u'|'v'|'w'|'x'|'y'|'z
'|'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M'|'N'|'O'|
'P'|'Q'|'R'|'S'|'T'| 'U'|'V'|'W'|'X'|'Y'|'Z'

<symbols> ::= '(' | ')' | '{' | '}' | '[' | ']' | '.' | ',' |
';' | '=' | '"' | '`' '+' | '-' | '#' | '\n' | '&' | '|' | '!'
| `_’ | `£’ | `^’ | `*’ | `$’ | `%’ | `/’ | `\’ | `ß’ | `æ’ |
'~' | '□' | '€' | '@' | '<' | '>' | '½'

<LP> ::= (

<RP> ::= )
```

<LCB> ::= {

<RCB> ::= }

<LSB> ::= [

<RSB> ::=]

<dot> ::= .

<comma> ::= ,

<semicolon> ::= ;

<assign> ::= =

<string_identifier> ::= “

<char_identifier> ::= ‘

<sign> ::= +|-

<hashtag> ::= #

<double_hashtag> ::= ##

<newline> ::= \n

<and> ::= &

<or> ::= |

<not> ::= !

<equal> ::= ==

<not_equal> ::= !=

<greater_than> ::= >

<less_than> ::= <

<greater_or_equal> ::= >=

<less_or_equal> ::= <=

<underscore>::= _

<empty> ::= “”

14. Comments

<comment> ::= <hashtag> <sentence> <hashtag>

<line_comment> ::= <double_hashtag> <sentence> <newline>

Explanation of BNF Description

1. Initial Program

<program> ::= begin <stmts> end

This non-terminal indicates that our language is used and it starts with the “begin” keyword and ends with the “end” keyword.

<stmts> ::= <stmt> | <stmt> <semicolon> <stmts>

This non-terminal indicates that our language consists of statements and users can concatenate them.

<stmt> ::= <matched_stmt> <end_stmt> | <unmatched_stmt> <end_stmt> | <non_if_stmt> <end_stmt>

This non-terminal indicates that our statements can be either matched, unmatched or non-if statements. It will be ended by the end statement. All of them are explained below.

<end_stmt> ::= <semicolon>

This non-terminal indicates the end of the written statement(s).

2. Variable Identifier

<term> ::= <var> | <constant>

This non-terminal indicates that our terms can be either variable or constants. They are explained below in this section.

<var> ::= <identifier>

This non-terminal indicates that our variables are identifiers. Identifiers are explained below.

<identifier> ::= <word> | <word> <identifier> | <identifier> <digit>

This non-terminal indicates that our identifiers can be words, or words followed by a digit or underscore. Identifiers are used to name such variables.

<constant> ::= <constant_identifier> <var>

This non-terminal indicates that constants are defined with their own identifier and a variable

<constant_identifier> ::= const

This non-terminal indicates that identifier of constants is “const” reserved word.

3. Types

<types> ::= bool | int | string | double | char

This non-terminal indicates the primitive data types that Egg language consists of. “bool” represents the Boolean type, “int” represents the Integer type, “string” represents the String type, “double” represents the Double type, and “char” represents the Character type. These abbreviations of primitive data types are used to indicate the type of the variable.

<assingment_values> ::= <bool_stmt> | <int_stmt> | <string_stmt> | <double_stmt> | <char_stmt>

This non-terminal defines each primitive data type statement that Egg language includes.

<bool_stmt> ::= <true_stmt> | <false_stmt>

This non-terminal is for boolean type statements. A boolean only consists of the true statement and false statement only. The difference from some other languages is that Egg language does not count ‘1’ and ‘0’ as boolean statements.

<int_stmt> ::= <sign> <int_stmt> | <int_stmt> <digit> | <digit>

This non-terminal is for integer type statements. An integer can consist of a stream of digits or a single digit. An integer can also have a single sign in front of itself (‘+’ or ‘-’).

<double_stmt> ::= <int_stmt> <dot> <digits> | <int_stmt>

This non-terminal is for double type statements. A double can consist of an integer. It also can consist of an integer following with a dot and stream of digits. So, a double has can have a floating part.

<char_stmt> ::= <char_identifier> <digit> <char_identifier> |
<char_identifier> <character> <char_identifier>

This non-terminal is for character type statements. A character includes a character between single quote marks. In single quote marks there can be a letter or a digit.

4. Assignment Operator

<assignment_operator> ::= <assign>

This non-terminal is for assignment operation. The assignment operator is basically an equal sign which indicates the right hand side of the equal sign will be assigned to the left hand side of the equal sign.

5. Declaration and Initialization

<declaration> ::= <types> <term> <semicolon>

This non-terminal is for the declaration of a variable. The declaration can be done by stating the term’s type and the term itself after that. In Egg programming language, in order to do a declaration, there should be a space between the primitive type and the term itself, and there should be a semicolon after the term.-

<initialization> ::= <term> <assignment_operator> <assingment_value> <semicolon>

This non-terminal is for the initialization of a term. The initialization can be done by stating the term itself, assignment operator after that, and assignment value to assign this value to the term itself. There should be a semicolon in the end.

<declaration_and_initilization> ::= <types> <term> <assignment_operator> <assingment_value> <semicolon>

This non-terminal is for declaration and initialization together. It simply does the combination of what declaration non-terminal and initialization non-terminal.

6. Boolean Values

<true_stmt> ::= true

This non-terminal is a part of <bool_stmt>. This indicates the boolean term's true state, opposite of false.

<false_stmt> ::= false

This non-terminal is a part of <bool_stmt>. This indicates the boolean term's false state, opposite of true.

7. Expressions

<factor> ::= <term>

This non-terminal is to extend the parse tree for operation priorities. Calculations that have the priority (i.e. division, multiplication, modulo) have the term factor to extend the parse tree and to be prioritized.

<max> ::= max <LP> <int_stmt> <comma> <int_stmt> <RP> | max <LP> <double_stmt> <comma> <double_stmt> <RP>

This non-terminal is to determine the maximum value among two integers or two doubles. If it is written according to its syntax with two integers or two doubles, it returns the maximum value among them.

<min> ::= min <LP> <int_stmt> <comma> <int_stmt> <RP> | min <LP> <double_stmt> <comma> <double_stmt> <RP>

This non-terminal is to determine the minimum value among two integers or two doubles. If it is written according to its syntax with two integers or two doubles, it returns the minimum value among them.

8. Precedence, Associativity of The Operators

<arithmetic_operation> ::= <addition> | <subtraction> | <division> | <multiplication> | <modulo> | <power>

This non-terminal represents all of the arithmetic operations in Egg Language.

<addition> ::= <addition> + <term> | <term> | (<addition>)

This non-terminal shows the syntax of addition operation in our language. Since add operation is left associative, addition is left recursive in our language. This situation is to make our language do addition in the order from left to right when precedence is equal. Also, (<addition>) allows the addition in parentheses to have a higher precedence

<subtraction> ::= <subtraction> - <term> | <term> | (<subtraction>)

This non-terminal shows the syntax of subtraction operation in our language. Since subtraction operation is left associative, it is left recursive in our language. This situation is to make our language do subtraction in the order from left to right when precedence is equal. Also, (<addition>) allows the subtraction in parentheses to have a higher precedence

<division> ::= <division> / <factor> | <factor> | (<division>)

This non-terminal shows the syntax of division operation in our language. <factor> non-terminal is to make this operation be at a lower level in the parse tree which results in division operation to have a higher precedence than addition and subtraction operations. Also, (<division>) allows the subtraction in parentheses to have a higher precedence

<multiplication> ::= <multiplication> * <factor> | <factor> | (<multiplication>)

This non-terminal shows the syntax of multiplication operation in our language. <factor> non-terminal is to make this operation be at a lower level in the parse tree which results in multiplication operation to have a higher precedence than addition and subtraction operations. Also, (<multiplication>) allows the subtraction in parentheses to have a higher precedence.

<modulo> ::= <factor> % <factor>

<modulo> non-terminal is used to demonstrate the syntax of mod operation which calculates the remainder of a value. Its precedence must be the same with the multiplication and division operations therefore we used <factor> non-terminals to make it at the same level with multiplication and division operations.

<power> ::= pow <LP> <int_stmt> <comma> <int_stmt> <RP> | pow <LP> <int_stmt> <comma> <double_stmt> <RP> | pow <LP> <double_stmt> <comma> <int_stmt> <RP> | pow <LP> <double_stmt> <comma> <double_stmt> <RP>

This non-terminal represents the syntax of power operation in Egg Language.It can be calculated by using double and integer numbers.

9. Loops

<loops> ::= <while_loops> | <for_loop> | <do_while_loop>

This non-terminal represents all loops.

<while_loop> ::= while <LP> <logical_expression> <RP> <LCB> <stmts> <RCB>

This non-terminal shows the syntax of while loop statements in Egg Language. If logical expression in LP and RP is true while loop continues to repeat statements in it. However, if the logical expression in parenthesis is false, statements are not performed.

<for_loop> ::= for <LP> <loop_initialization> <semicolon> <logical_expression> <semicolon> <arithmetic_operation> <RP> <LCB> <stmts> <RCB>

This non-terminal demonstrates the syntax of for loop in our language. Loop initialization is to define and initialize an integer. While logical expression is true, statements between curly brackets are performed. After every time statements are performed, arithmetic operation is performed and this continues until the logical expression becomes false.

<do_while_loop> ::= do <LCB> <stmts> <RCB> while <LP> <logical_expression> <RP> <semicolon>

This non-terminal is to demonstrate do-while loop. At the first pass, it performs statements in curly brackets without considering logical expression. After that, while the logical expression is true, it performs statements inside curly brackets until the logical expression becomes false.

<loop_initialization> ::= <initialization> | <declaration_and_initialization>

This non-terminal demonstrates the initialization in loops. It allows programmers to only initialize a previously declared variable and also it enables programmers to initialize and declare a variable.

10. Conditional Statements

<matched_stmt> ::= if (<logical_expression>) <stmt> | elif (<logical_expression>) <stmt> | <non_if_statement>

This non-terminal solves the ambiguity between if and else statements according to matched or unmatched situations. At the end, it is converted to a non-if statement which is explained below.

<unmatched_stmt> ::= if (<logical_expression>) <stmt> | elif (<logical_expression>) <stmt>| if (<logical_expression>) <matched> else <unmatched>

This non-terminal is similar above. It solves the ambiguity between if and else statements according to matched or unmatched situations. At the end, it is converted to a non-if statement which is explained below.

<non_if_stmt> ::= <loops> <non_if_stmt> | <arithmetic_operations> <non_if_stmt> | <function_call> <non_if_stmt> | <function_declaration><non_if_stmt> | <declaration> <non_if_stmt> | <initialization> <non_if_stmt> | <declaration_and_initilization> <non_if_stmt> | <input_stmt> <non_if_stmt> | <output_stmt> <non_if_stmt> | <comment> <non_if_stmt> | <double_comment> <non_if_stmt> | <loops> | <arithmetic_operations> | <function_call> | <function_declaration> | <declaration> | <initialization> | <declaration_and_initilization> | <input_stmt> | <output_stmt> | <comment> | <double_comment>

This non-terminal indicates that if a statement is not if-else, it will be treated as a non-if statement which includes all other non-terminals. Eventually, every statement becomes a non-terminal statement. Therefore, every operation will be decided in this non-terminal.

<logical_expression> ::= <single_expression> | <recursive_expression>

This non-terminal indicates that our logical expression can be single or more than one whose results are connected to each other to decide the final result.

<single_expression> ::= <term> <logical_operator> <term> | <bool_stmt> | <not> <bool_stmt>

This non-terminal is to demonstrate the syntax of of a single logical expression which does not contain any logical connector like & and |.

<recursive_expression> ::= <recursive_expression> <logical_connector> <single_expression> | <single_expression>

This non-terminal indicates the syntax of logical expressions that can contain more than one condition so that the final result is decided according to these multiple expressions. These multiple expressions are connected with logical connectors which are & and |.

<logical_operator> ::= <equal> | <not_equal> | <not> | <greater_than> | <less_than> | <greater_or_equal> | <less_or_equal>

This non-terminal includes logical operators to compare two logical statements or can be used in front of a logical statement as <not> non-terminal.

<logical_connector> ::= <and> | <or>

This non-terminal includes the “and”, “or” non-terminals that are used to connect two or more expressions.

11. Statements For Input / Output

<input_stmt> ::= eggin <LP> <input_context> <RP>

This non-terminal is to get input from the user. When this non-terminal is used, the program stops to get input from the user to continue. It declares the input to value entered in brackets.

<output_stmt> ::= eggout <LP> <output_context> <RP>

This non-terminal is to print out the value entered in brackets as output.

<input_context> ::= <term> + <input_context> | <term>

This non-terminal indicates the input that is used in <input_stmt>. It can be used with different terms with a separation of a ‘+’ sign. The inputs that are entered by the user will be assigned to the terms respectively.

<output_context> ::= <term> | <assignment_values> | <term> + <output_context> | <assignment_values> + <output_context>

This non-terminal indicates the output that is used in <output_stmt>. It can be used with different terms or assignment values with a separation of ‘+’ sign. The outputs that are separated from each other with ‘+’ sign will be displayed to the user respectively.

12. Function Definitions and Function Calls

<function_types> ::= <types> | void

This non-terminal indicates that function’s type can be either given types or void which indicates it does not return any types.

<function_declaration> ::= <function_types> funct_<function_name> (<parameter>) {<stmts>} | <function_types> funct_<function_name> (<parameter>) {<stmts> return <assignment_values> } | <function_types> funct_<function_name> (<parameter>) {<stmts> return void}

This non-terminal indicates how to define a function. First the type of function is required. Then the user should add “funct_” before the name of the function in order to make a function. After that, s/he can add parameters if it is needed. Finally the user needs to define the body of the function.

<function_call> ::= funct_<function_name> (<identifier>);

This non-terminal indicates how to call a defined function. Users should write “funct_” before the name of the function. S/he must write the parameters of the function if it has parameters accordingly.

13. Symbols

<sentence> ::= <word> | <digit> | <digit> <sentence> | <sentence> <word> | <space> | <sentence> <space> | <sentence> <digit> | <sentence> <symbol> | <symbol>

This non-terminal <sentence> demonstrates groups of letters, numbers and symbols. It is used in strings and comments.

<word> ::= <lower_characters> | <lower_characters> <word> | <lower_characters> <character> | <lower_characters> <digits> | <lower_characters> <under_score>

This non-terminal word is used to create type identifiers. For that purpose, it only takes lower case letters for the first character and after that it can take all letters, digits and underscore.

14. Comments

<comment> ::= <hashtag> <sentence> <hashtag>

This non-terminal indicates that comments with more than one line are defined between two hashtags. Everything written between hashtags is considered a sentence that is a non-terminal.

<line_comment> ::= <double_hashtag> <sentence> <newline>

This non-terminal indicates that one line comments start with a double hashtag and end with a new line.

Explanation of Non-Trivial Tokens

begin: This non-trivial token is a keyword to start the program.

end: This non-trivial token is a keyword to end the program.

main: This non-trivial token is reserved to the main function which contains all executions.

funct_: This non-trivial token is reserved to make functions in Egg language.

void: This non-trivial token is reserved to return void or decide the type of the function as void.

return: This non-trivial token is reserved to return a function.

max: This non-trivial token is reserved to get maximum among two numbers. Returns an integer.

min: This non-trivial token is reserved to get a minimum among two numbers. Returns an integer.

pow: This non-trivial token is reserved to calculate the power of an integer. Returns an integer.

eggin: This non-trivial token is reserved to get an input from the user.

eggout: This non-trivial token is reserved to get an output from the user.

const: This non-trivial token is reserved to demonstrate a variable is declared as a constant variable.

if: This non-trivial token is reserved for if statements.

else: This non-trivial token is reserved for else statements.

elif: This non-trivial token is reserved for else-if statements.

while: This non-trivial token is reserved for while loops.

do: This non-trivial token is reserved for do-while loops.

for: This non-trivial token is reserved for “for” loops.

true: This non-trivial token is reserved for booleans that return true.

false: This non-trivial token is reserved for booleans that return false.

int: This non-trivial token is reserved for integer numbers.

double: This non-trivial token is reserved for floating numbers.

bool: This non-trivial token is reserved for booleans.

string: This non-trivial token is reserved for strings that contain characters and other written stuff.

char: This non-trivial token is reserved for characters.

 # … # : This non-trivial token is reserved for comments that are not limited to one line. In order to use this comment, it should start with '#' and end with '#' as well.

: This non-trivial token is reserved for one line comments. In order to write one line comment the user should start with double "#" and end with a new line.

Evaluation of Egg Programming Language

A. Readability

Overall Simplicity

Since Egg is designed for games, it contains limited features and constructs which makes it very simple. Operators are also enough to make operations for a game. Needed arithmetic operations can be done in one way which make our language plain. However, we use curly brackets and semicolons which makes our language hard to read slightly.

Orthogonality

Since Egg only contains primitive types such as int, bool, etc. it is easy to read and write. Moreover, lots of combinations are possible and exceptional cases are nearly zero.

Data Types

In Egg, all needed primitive types are defined. However, we do not have any complex data types which reduce the functionality.

Syntax Considerations

In Egg, lots of identifier combinations are allowed. Also, there are reserved words that are very similar to other languages which makes our language easy to be adapted. We have new meaningful new keywords to make our language readable and adaptable such as “begin” is a keyword to start the program.

B. Writability

Simplicity and Orthogonality

Egg programming language designed and developed for game developers. This is why Egg does not have complex constructs and a large number of primitives. Since Egg may not be useful for other intentions than developing a game, it is simpler and easier to write than many other languages. Although it has fewer complex structures, it also has a very similar syntax to many well-known programming languages.

Support for Abstraction

In Egg, complex structures are not supported. However, details are ignored if it is needed. For instance, a user can easily write a sentence with all given symbols, numbers, characters inside two quotation marks.

Expressivity

Since operations are done in one way in Egg, it reduces our writability while it increases readability. However, our number of operations are enough to do mid level complex operations. We predefined some functions in Egg but It is not enough to make complex operations. On the other hand, users can easily define their own functions for their specific games which makes our language easy to write.

C. Reliability

Readability and Writability

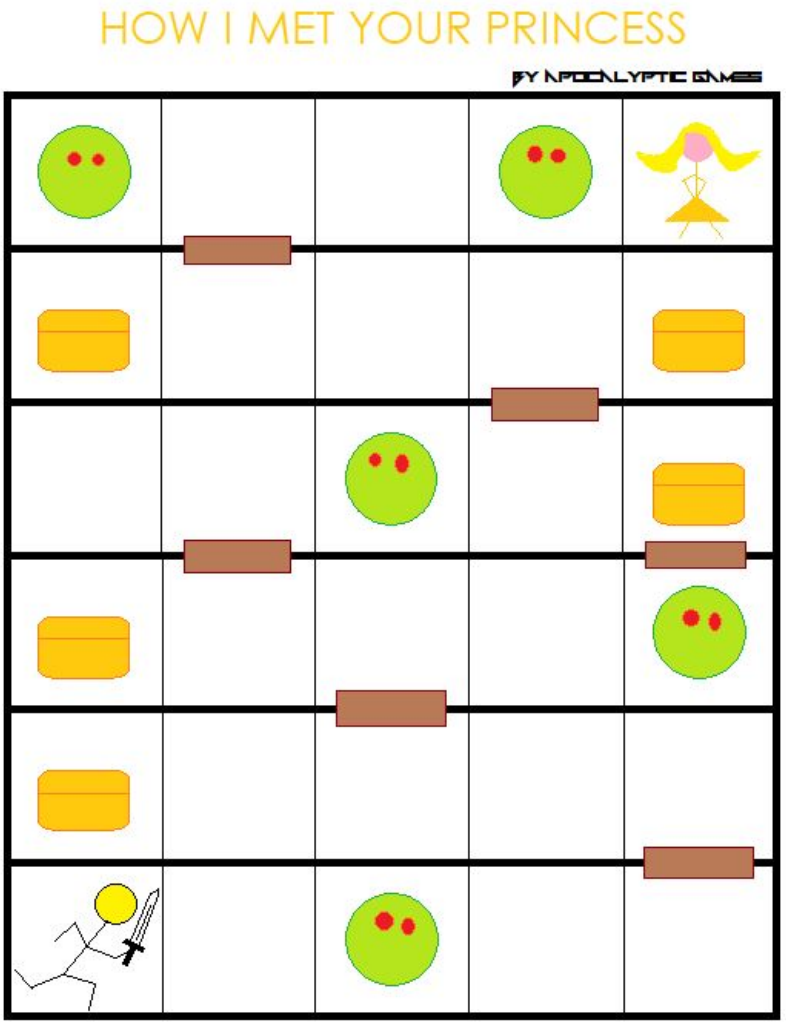
Egg Programming Language is a simple and plain language therefore it is easy to read and write in Egg. As it is readable and writable, this situation makes it easy to detect bugs and errors. However, it might not be enough to make high-level complex operations with built-in functions. The users can easily define their own functions.

Adventure Game Written in Egg

Appendix 1: eggGame.txt (Code of the game can be found along with the Project Report)

How to Play: This is a simple game in the Action-RPG genre. The main goal of this game is to save the Princess from the castle.

The player starts as a Knight shown in the bottom left corner of the picture. The player can walk in every direction s/he wants (up, down, left, right). Green circles are enemies, if Player encounters any one of them, s/he needs to fight to pass them. Every enemy's difficulty depends on what stage (floor) they are at. Every time Player successfully skips a stage, s/he gets an opportunity to shop and buy some gear to increase the Knight's damage. If the Player encounters yellow chests around the map, Knight eats food which will regenerate his strength, Knight gets extra gold which will increase his wealth, Knight will level up which will increase his damage. The brown bricks are doors. Every time Knight opens a door, he gets an opportunity to shop and skips the stage that he is in.



Test Program 2: Fibonacci Calculator in Egg

Appendix 2: eggFibo.txt (Code of the program can be found as a txt file)

Fibonacci Calculator calculates the nth Fibonacci number. It exercises most of the constructs in Egg Programming Language. Firstly, it takes the user name as an input and initializes a constant string variable with the user's name. After that, the program gets 2 integer inputs from the user and calculates max and min and then calculates the max^min and prints it to the user. This part was just for exercising max, min and power constructs. After that, the program gets an input for the value of n variable, and then, it calculates the nth Fibonacci number. If the input is less than 1, it generates an error message. After every calculation, the program asks the user whether s/he wants to calculate another nth Fibonacci number or exit.