# Examination Generation Grading Language (EGGL)

Language Reference Manual

Gordon Hew (CVN) (gh2242@columbia.edu)

COMS W4115: Programming Languages and Translators
Fall 2010, Professor Stephen Edwards

## Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>Lexical Convention</td>
<td>3</td>
</tr>
<tr>
<td>Comments</td>
<td>3</td>
</tr>
<tr>
<td>Identifiers</td>
<td>3</td>
</tr>
<tr>
<td>Constants</td>
<td>3</td>
</tr>
<tr>
<td>Reserved Keywords</td>
<td>3</td>
</tr>
<tr>
<td>Separators</td>
<td>4</td>
</tr>
<tr>
<td>White-spaces</td>
<td>4</td>
</tr>
<tr>
<td>Types</td>
<td>4</td>
</tr>
<tr>
<td>boolean</td>
<td>4</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>array</td>
<td>4</td>
</tr>
<tr>
<td>map</td>
<td>4</td>
</tr>
<tr>
<td>Operators</td>
<td>5</td>
</tr>
<tr>
<td>Unary Operators</td>
<td>5</td>
</tr>
<tr>
<td>Arithmetic Operators</td>
<td>5</td>
</tr>
<tr>
<td>Relational Operators</td>
<td>5</td>
</tr>
<tr>
<td>Equality Operators</td>
<td>5</td>
</tr>
<tr>
<td>Logical Operators</td>
<td>5</td>
</tr>
<tr>
<td>Expressions</td>
<td>5</td>
</tr>
<tr>
<td>Primary Expressions</td>
<td>5</td>
</tr>
<tr>
<td>Unary Expressions</td>
<td>5</td>
</tr>
<tr>
<td>Arithmetic Expressions</td>
<td>5</td>
</tr>
<tr>
<td>Relational Expressions</td>
<td>6</td>
</tr>
<tr>
<td>Equality Expressions</td>
<td>6</td>
</tr>
<tr>
<td>Logical Expressions</td>
<td>6</td>
</tr>
<tr>
<td>Statements</td>
<td>6</td>
</tr>
<tr>
<td>Expression Statement</td>
<td>6</td>
</tr>
<tr>
<td>Compound Statement</td>
<td>6</td>
</tr>
</tbody>
</table>
Conditional Statement................................................................. 7
For Loop Statement....................................................................... 7
While Loop Statement................................................................. 7
Return Statement.......................................................................... 7
Prompt Statement......................................................................... 7
Answer Statement......................................................................... 7
Choice Statement......................................................................... 7
Function Definitions..................................................................... 8
  Basic Functions......................................................................... 8
  Question Functions................................................................... 8
  Built-in Functions.................................................................... 9
Scope............................................................................................ 9
Example....................................................................................... 9
Introduction

The Examination Generation Grading Language (EGGL) seeks to provide a simple language with built-in grading and question sequencing mechanisms that can easily facilitate the creation and scoring of computerized exams and assignments. Such a language would be beneficial to the teaching community in that it would allow instructors to quickly write an interactive exam that can be scored instantly.

Lexical Convention

Comments

EGGL supports single line comments. The sequence of /* indicates the start of a comment and a subsequent */ indicates the end of the comment. Anything within the bounds of the comment declaration will not be processed by the compiler.

Example: /* This is a comment */

Identifiers

An identifier is the name of a variable, constant, or a function declared in EGGL. A variable name can only consist of alphanumeric characters and must begin with a letter. The use of symbols and special characters are not permitted.

Constants

Constants are immutable assignments in EGGL. They are declared with the keyword const before the variable assignment.

Example: const int CONSTANT = 10;

Reserved Keywords

The following keywords are reserved and could not be used for variable or function names:

- const
- print
- prompt
- choices
- answer
- double
- if
- else
- for
- while
- return
- func
**Separators**

There are two separators in the EGGL language, they are commas (,) and semi-colons (;). A comma is used for declaring a sequence of items such as in an array. Semi-colons are used to indicate the end of a statement.

**White-spaces**

White-spaces such as tabs, carriage returns, and new lines are ignored during compilation. Spaces are used to identify keywords and variable declarations.

**Types**

The following data types are supported by EGGL:

**boolean**

The boolean data type is used for declaring a boolean value which can either be true or false.

- Declaration Example: `boolean x = true;`

**int**

The int data type is used for declaring a 32-bit signed integer.

- Declaration Example: `int x = 9;`

**float**

The float data type is used for declaring a 32-bit floating point.

- Declaration Example: `int x = 5.5;`

**array**

The array data type is used for creating an array of n-elements whose size must be defined at declaration. It supports the insertion of elements at an n-index.

- Declaration Example: `array x = array(2); // empty array`
- Alternative Declaration Example: `array x = [ 1, 2, 3 ]`
- Insertion Example: `x[0] = 2;`
- Retrieval Example: `int y = x[0];`

**map**

The map data type is used for creating a key-value pair map. It supports the insertion of values with unique keys into a map.

- Declaration Example: `map x = map();`
- Insertion Example: `x("key1") = 10;`
- Retrieval Example: `int y = x("key2");`
Operators

**Unary Operators**
- ‘-’: negative
- ‘!’: not

**Arithmetic Operators**
- ‘+’: addition
- ‘-’: subtraction
- ‘*’: multiplication
- ‘/’: division

**Relational Operators**
- ‘<’: less than
- ‘<=’: less than or equal to
- ‘>’: greater than
- ‘=>’: greater than or equal to

**Equality Operators**
- ‘==’: equal to
- ‘!=’: not equal

**Logical Operators**
- ‘&&’: and
- ‘||’: or

**Expressions**
All expressions group left to right.

**Primary Expressions**
Primary expressions can be an identifier or constant.

**Unary Expressions**
unary_expression → unary_operator expression
The only valid unary_operators are ‘-’ and ‘!’. ‘-’ is valid when expression is a float or int. ‘!’ is only valid when the expression is a boolean.

**Arithmetic Expressions**
arithmetic_expression → expression + expression
An arithmetic_expression is only valid when expression evaluates to a float or int.

**Relational Expressions**

relational_expression → expression < expression
| expression <= expression
| expression > expression
| expression => expression

The evaluation of the operators '<', '<=', '>', and '=>' returns true or false. A relational_expression is only valid when expression evaluates to an int, float, or string and are of the same type on both sides of the operator.

**Equality Expressions**

equality_expression → expression == expression
| expression != expression

The evaluation of the relational operators '==' and '!=' returns true or false. An equality_expression is only valid when expression evaluates to an int, float, string, or boolean and are of the same type on both sides of the operator.

**Logical Expressions**

logical_term → relational_expression | equality_expression
logical_expression → logical_term && logical_term
| logical_term || logical_term

The evaluation of the logical operators '&&' and '||' returns true or false based on AND/OR truth table logic.

**Statements**

**Expression Statement**

expression;

The most common and basic statement is an expression statement.

**Compound Statement**

{
    statement
    statement
}

A compound statement is a list of statements to be evaluated.
**Conditional Statement**

if(expression) statement else statement
if(expression) statement elseif (expression) statement
A conditional statement is used to evaluate if-else logic control logic.

**For Loop Statement**

for(expression-1; expression-2; expression-3)
    statement
The for-loop statement is used to run a statement until a condition is no longer met.
expression-1 is the initial condition, expression-2 is the condition in which to continue,
expression-3 is the mutation of the initial condition to a new value.

**While Loop Statement**

while(expression)
    statement
The while-loop statement is used to run a statement until the expression is no longer met.
The expression should only evaluate to true or false.

**Return Statement**

return(expression);
The return statement is the value that is returned from a function. expression must match the
data type that the function is declared to return.

**Prompt Statement**

prompt(expression);
A prompt statement is required for question-functions and can only be declared once inside a
function. The expression must be a string data type in the form of a text question.

**Answer Statement**

answer(expression);
An answer statement is required for question-functions and can only be declared once inside a
function.

**Choice Statement**

choice(expression);
A choice statement is optional for question-functions. It provides a discrete set of choices that
can be selected to match the answer of a question-function. The expression must be an array of
the same data type.
Function Definitions

**Basic Functions**

\[
\text{return-data-type} \rightarrow \text{int} \\
| \text{float} \\
| \text{boolean} \\
| \text{string} \\
| \text{map} \\
| \text{array} \\
| \text{void}
\]

```plaintext
func \text{return-data-type} \ \text{function-name}( \ \text{param-1}, \ \ldots ) \\
\{
\begin{align*}
\text{statement}* \\
\end{align*}
\}
```

A basic function definition is composed of a return-data-type (the data type to be returned), the function-name (name of the function), and the parameters to be passed into the function. Inside the function body can be any number of statements. The last statement that must be called is a return-statement if the return-data-type is any value other than void.

**Question Functions**

\[
\text{weight-expression} \rightarrow \text{integer} \\
\text{difficulty-expression} \rightarrow [1-10] \\
\text{appearance} \rightarrow \text{asc} | \text{desc} | \text{rand}
\]

```plaintext
func \text{question} \ @\text{weight}=\text{weight-expression}? \ [\@\text{difficulty}=\text{difficulty-expression}|\@\text{appearance}=\text{appearance-expression}]? \ \text{function-name}( \ \ldots ) \\
\{
\begin{align*}
\text{statement}* \\
\text{prompt-statement} \\
\text{statement}* \\
\text{choice-statement}? \\
\text{statement}* \\
\text{answer-statement} \\
\end{align*}
\}
```

A question function definition has the keyword question immediately after the keyword func. The weight and difficulty attributes following the function are used for grade and question sequencing respectively.

@weight=weight-expression is an optional field that is used for weighting questions when EGGL grades the questions at the end of a completed exam during run-time.
expression can only be an integer value between 1-10. If weight-expression is constant across all questions, all of the questions will be equally weighted. Additionally, if @weight is not present, the question will be assigned a weight of 1 by default. The grade calculation that occurs at the end of the exam's run-time is the sum of the weights of the questions that are correct divided by the total weight of all the questions.

@difficulty=difficulty-expression is an optional field that is used for easily creating adaptive exams. difficulty-expression must be an integer value between 1-10. If at least one question function in the program has this attribute then all question functions in the program must have this attribute.

@appearance=appearance-expression is an optional field that is used for determining the order in which questions will appear to the tester. asc displays question declarations from the bottom of the file to the top, desc displays the questions declaration in the file from the top to the bottom, and rand displays questions in a random order. If @appearance is not defined, the default sort order is desc.

A question function must have a prompt-statement and an answer-statement. It can optionally have a choice-statement and any number of statements in between.

**Built-in Functions**

There are several built-in functions in EGGL library as defined below:

- size(expression): used to find the size of an array, expression should of type array.
- print(expression): prints text to standard out, expression should be of type string.

**Scope**

EGGL uses static scoping and has its scopes separated by blocks which are encapsulated by curly braces { ... }.

**Example**

File: BasicExam.eggl
const string YES = “yes”;
const string NO = “no”; 

/* custom function declaration */
func array yesNoChoices(){
    return([ YES, NO ]); 
}

/* question definition */
func question question1(){
prompt( "Is the sky blue?" );
choices( @yesNoChoices );
answer( "yes" );
}

func question question2() {
    prompt( "Who is the largest car maker?" );
    answer( "Toyota" );
}

Sample output:
1. Is the sky blue?
a. yes
b. no
> a
2. Who is the largest car maker?
> Honda
50% of questions answered correctly.