# GAQ Language Reference Manual

## 1. Introduction

The GAQ language was invented to allow for easy, flexible survey creation and is intended to have a broad range of applications. The Reference Manual is intended to describe the GAQ Standard and is based off of the C Reference Manual. Since there are many elements of GAQ that coincide with C, there will be parts of the C Reference Manual quoted verbatim.

## 2. Lexical Conventions

The GAQ language is designed for readability and ease-of-use in mind and meant to be quickly taught to non-programmers. Thus, each program will be stored in one file. There is no concept of including separate translation units in separate files and linking them together at this time, although that would be a useful feature to add in a future edition.

### 2.1 Tokens

There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. White space, as described below, is ignored except as it separates tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants. If the input stream has been separated into tokens up to a given character, the next token is the longest string of characters that could constitute a token.

### 2.2 Comments

The characters /* introduce a comment, which terminates with the characters */ Comments do not nest, and they do not occur within string literals.

### 2.3 Identifiers

An identifier is a sequence of letters, digits, and the underscore, beginning with a letter. Identifier names are case-sensitive and may have any length.

### 2.4 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

*default*
*repeat*
*response*
*results_to_file*
*results_to_stdout*

*print*
*int*
*float*
*bool*
*string*
*yes*
*no*

## 2.5 Constants
There are several kinds of constants.  Each has a datatype.

- *Integer-constants*
- *String Literals*
- *Floating-constants*
- *Boolean-constants*

### *2.5.1 Integer Constants*
An integer constant consists of a series of digits, taken to be decimal.  It may be preceded by a – character to represent negative integers.  Its type is an **int.**

### *2.5.2 String Literals*
A string literal, also called as string constant, is a sequence of characters surrounded by double quotes.  It is of type **string** and initialized with the given characters.  The literals are immutable and two identical string literals are not necessarily equal.

### *2.5.3 Floating constants*
A floating constant consists of an integer part, a decimal point, a fraction part, and e or E, and an optionally signed integer exponent.  The integer and fraction parts both consist of a sequence of digits.  Either the integer part or the fraction part (not both) may be missing; either the  decimal point or the e and the exponent (but not both) may be missing.  The type is **float.**

### 2.5.4 Boolean constants
There are 2 boolean constants that are reference via keywords – true and false.  Their type is **bool**.

## 2.6 Whitespace
Whitespace is defined as the ASCII space, horizontal tab and form feed characters, as well as line terminators and comments.

# 3 Syntax Notation
The syntax for specifying a full GAQ program is as follows:

*Program*:

$declarations_{opt}\ question_{opt}\ footer_{opt}$

Thus, an empty program is an acceptable, though admittedly useless, program.

## 3.1 Meaning of Identifiers

Identifiers are used to refer to objects, also called variables. A variable is a location in storage, and its interpretation depends on the type of the variable. The valid types for variables are ***int***, ***float***, ***bool***, and ***string***. Variables must be assigned to when they are declared.

## 3.2 Variable Declarations

Variable declarations are only allowed at the beginning of the program. Once the next section, the question specification, begins, no further declarations are allowed.
Variable declarations are of the form

*variable-declarations*:

*type identifier = constant;*

where the constant needs to be of the same type as the variable type. The valid types for variable declarations are ***int***, ***float***, and ***bool***.

## 3.3 Question Specifications

### 3.3.1 Question Syntax

The questions in a GAQ program are specified in a nested fashion. Only one root question is currently allowed.
Each question that will be asked to the user is represented as a string followed by the type that it will be cast to. The response to each question will be stored in the implicitly-declared response variable of type *opt-type*. Each *answer* will be able to access that variable. When a new nested question is asked, a new response variable will be in the scope, shadowing the one available for the previous question. The scope of the *response* variable will begin after the response *opt-type* declaration and will be available for the rest of the *question*, except when a nested question is in scope. For discussions about how casting is done from the string response into the *response* variable, see the section on casting below.

*question:*

*string-literal* $type_{\text{opt}}$ *answer*

#### *3.3.1.1 Strings*

A string is a sequence of characters surrounded by double quotes. It is of type ***string*** and initialized with the given characters. The string can have references to variables in it which will be evaluated at runtime. To reference a variable from within a string, the variable identifier will be preceded by ${ and followed by }. To print the actual characters of ${ and not have them interpreted as enclosing an identifier, they can be preceded by a backslash. Thus, the question of "`${hello}`" would print the contents of the variable identified by hello. Whereas the question of "`\${hello}`" would print the string of characters: `${hello}`. In order to print non-string variables inside questions, they will implicitly be cast into strings. See the casting section below for the exact definitions.

## 3.3.2 Answer Syntax

Each question is followed by a series of expressions enclosed in square braces.

> a*nswer*:
>
> > [*expression*] *answer*
> >
> > *{answer-block}*

### *3.3.2.1 Expressions*

An *expression* can be either a constant, which is compared to the *answer*, or it can be an expression based on other variables, in which case the response is saved as a string-literal in the reserved identifier response.

> *expression:*
>
> > *constant*
> >
> > *variable*
> >
> > *(type) variable*
> >
> > *expression operator expression*

### *3.3.2.2 Operators*

The following operators are available for use in expressions.  Their use is the same as in expressions in the C language.

*Relational operators*: < = , < , >= , > , == , !=

*Logical operators*: & (and), | (or) - apply only to float or int

*Mathematical binary operators*: + , -, *, / - apply only to float or int

### *3.3.2.3 Casting*

Casting is allowed inside expressions.  Explicit casts must be performed between each type – ***bool**, **int**, **float***, and ***string**.*

*Casting to string*

- ***bool*** will be cast to the strings "yes" and "no".
- ***int*** will be cast to a string representation of an int, with an optional – followed by a series of digits.
- ***float*** will be cast in the notation of optional negative sign followed by the integer part, followed by a decimal, followed by the fractional part to the 4th decimal place.

*Casting to an **int***

- ***bool*** will cast yes to 1 and no to 0.
- ***float*** will cast to the integer part of the float.
- ***string*** will cast to an integer in the same format as the C function atoi.  Invalid strings will cast to 0.

*Casting to a **bool***

- ***int*** will cast to yes if they are non-zero, no otherwise.
- Casting of ***float*** to ***bool*** is not allowed.
- Casting a ***string*** to a ***bool*** will cast to yes if the ***string*** is "yes", to no otherwise.

After each group of expressions in square braces, there will be a block of code called the *answer-block*, discussed next. This block of code will be entered into only if the user's response to the questions matches the constant in the square braces, if it is a constant, of if the expression evaluates to true, if it is an expression.

### 3.3.3 Answer Block

Each block of code in the *answer-block* will consist of answer expressions, separated by ';' characters and further nested questions. An answer block can be empty, in the case of the final answer block in the tree.

```
answer-block :
        answer-expression;$_{\text{opt}}$ answer-block
        question
```

### 3.3.4 Statement

Expressions can take two forms. They can either be a variable identifier assigned an expression, such as hello = 5+2; or a print output statement, described below.

```
statement:
        identifier = expression;
        print (string);
```

### 3.3.5 Answer Expression

Expressions that are allowed inside the *answer-block* can take two forms. They can either be a variable identifier assigned an expression, such as hello = 5+2; or a print output statement, described below, or they can be the keyword repeat. Repeat must be the last expression in the Answer Expression.

```
answer-expression:
        statement
        repeat;
```

## 3.4 Program Footer

The program footer will be run after all questions are evaluated. This will be the place where further variable assignments can be done and output operations will be performed. It consists of a list of statements. The program footer is optional.

```
footer:
        statement$_{\text{opt}}$ footer
```

# 4 Output

There is one way to print output in a GAQ program.

```
print ( string );
```

This syntax will print the string inside of the parenthesis to stdout. The format of the string is the same as in the *question* syntax and will do variable substitution before printing the results. This form of printing can be done anywhere in the program – following variable declarations, inside an *answer-block*, or following any *question*.