**Documentation of Goldfish v8-2-2**

The following documentation is intended to be a succinct summary of the Goldfish syntax according to version 8-2-2 of the interpreter. Although much of the syntax carries over from previous versions, various modifications to the interpreter have changed key aspects, such as how objects are treated and what built-in system functions are available for use by the user.

It is important to know that in the Goldfish language, the entire file is run from start to finish, skipping function definitions unless called, as if it were a main function. However, the file is not treated as a function. This means that it cannot be passed variables nor are its variables locked as in other layers of the runtime stack.

**User data types**

The user can create three types of objects: variables, functions, and defined types. They do so simply by declaring a name. The default type is variable. However, this can be changed by either altering the type (see ...) or by adding parenthesis after the name, which automatically makes the object a function.

Upon declaration, an object is also used (except functions if followed by brackets). The object is saved and its current value returned. At this point, a variable returns a value and can be assigned one. Functions and defined types also return values, but they cannot be assigned a value. Furthermore, neither has to be used to return a value.

**Functions**

Functions are declared in the same way variables are: by being named. Although a variable can be made into a function by altering its type, normal functions are followed by parenthesis. Once a function is identified, its definition is searched for in file. The definition is given by the function name, followed by a pair of parenthesis and a pair of curly brackets containing operations to be performed.

Functions are followed by parenthesis that can contain the names of variables whose values are to be passed to the function. Only the values of these variables will be passed to the function. Once the function definition is found, the values will be assigned to the variable names contained within the parenthesis at the beginning of the definition. These values can then be used within the function by the names given in the parenthesis.

The function definition has curly brackets that contain within them the operations to be performed upon calling the function. A “return” statement can be used to save a value to the function’s value so that it can be used in future operations.

**Defined Types**

A defined type is only created upon the altering of an object’s type from variable to “defined” or via first giving the definition of that type.

When a defined object’s name is found, its type’s name is identified and is searched for in file. The definition is given by the name of the type followed by a pair of curly brackets containing the operations to be performed. Note that the default type for a defined object is the object’s own name, but this can be changed by altering the object’s type.

What is unique about definitions is that they are considered a different type of layer in the stack than functions. Variables outside of the defined type’s definition are not locked when a defined type’s definition is processed. However, the runtime location in file after the calling of the defined type’s name is accessible by certain system commands (see System Commands > System) within the defined type’s definition. This means operations can be performed on objects around the location of the original calling.

**Alittle on how objects are managed...**

User-created objects are stored in a vector stack. This stack is searched from bottom to top (or, in other words, the last added to the first) whenever a user object is found. The first object whose name matches the one from file is considered. This is true for variables when it is not “locked” (This occurs when an object is either exported by the user or when the object’s declaration is outside of a function definition that is currently being processed). The object’s type is next examined. Both functions and definitions cause a search for a definition of the object in file.

**Built-in Operators**

System characters, or built-in operators, are special characters that are read as individual tokens, despite being next to other characters and not separated from other tokens by spaces. These characters are:

! @ ` ~ # $ % ^ & \* ( ) \_ + - = [ ] { } | \ / ? < > , . “ ‘ ; :

The Goldfish language is unique in that many of the system characters can be assigned a definition. Those that cannot be are listed below along with the action they perform:

**Space**

The space ( “ ” ) performs only one function, the separation of tokens. Tokens are individual items recognized by the interpreter’s parser and are extracted and acted upon according to the actions attributed to them (based on their type if they are a user object).

**;**

The semicolon is the statement-deliminating character. All operations in the interpreter treat it as a statement deliminating character and some even expect it. It does not return a value, nor does it allow for any final value of the statement it ends be available for use in the next statement.

**( )**

Parentheses group operations. All of the operations in a set of parenthesis are performed and the final value returned. This also applies for “if” and “while” statements, although the parentheses for those operations are handled by the system functions handling their associated operations.

**“**

The string of characters between quotation marks will be treated as a value with no name and not as an object’s name.

**\n**

Backslash n ( “\n” ) is the line-deliminating character in c++, the language the Goldfish interpreter is written in. Thus, it also carries over to the Goldfish syntax. Furthermore, it can be saved in values given in quotation marks via \n and not some special character.

**=**

The equals sign is used as an assignment symbol. It assigns a value to a variable if the variable is either universal (declared outside of all functions) or is unlocked. The value to be saved to the variable is the final value from all of the operations between the assignment symbol and the first unused semicolon following it. (Used semicolons are those that have already been used to terminate semi-colon dependent operations, such as other assignment statements contained within this one.)

**,**

The comma has no particular function but is reserved for separation of items and user use. However, like the other system characters given here, it cannot have a definition assigned to it.

**{ }**

Curly brackets act like parenthesis. The only difference is that their operations are terminated upon the action of a return statement or a break statement.

**\**

Backslash returns the next character found in file without executing any operations associated with it or the token to which it may be attached.

**Built-in Functions – System commands**

The following commands are available for use in Goldfish. Note that items contained within square brackets ( [ ] ) are numbers or values dependent on user input. Everything else is exactly how it should be typed.

**system**

system previous name

= Returns the name of the previous object found in file.

system previous value

= Returns the value of the previous object found in file.

system previous save\_as [new name] [new value]

= Overwrites the information concerning the previous object found in file.

system next

= Returns information concerning the next character found in file.

Realize that a reading position is saved whenever a defined type is called. The definition of the type is then searched for and processed. However, within the processing, it may be necessary to access objects around the location of the original calling of the defined object. They can be accessed via “system next”. Note, however, that a defined type can be called within a defined type. When it is still necessary to access the objects around the location of the first calling of a defined type and not the recent calling, the “sub” command can be used. Preceding “next” and after “system” may be ( sub [level] ) where “level” is the number of old reading positions from the most current to scan for objects. Note that the “sub” command (with its parameter) can be inserted regardless of the commands that follow the “next” command.

system next name

= Returns the name of the next token found in file after the location the defined type object was called.

system next value

= Returns the value associated with the next object found in file after the location the defined type object was called.

system next total

= Returns the final value of the operations between the location the defined type was called and the next semicolon (like the assignment symbol, but it does not assign a value to an object).

Between “next” and the given sub command, “name”, “value”, or “total”, can be a value indicating the number of tokens to skip.

system next goto [token]

= Skips all tokens (around the location the defined type was called) up to the given token.

**end\_main**

This indicates the end of operations in the file. It does not have to be included anywhere in the file, but it can be used to end all operations prematurely.

**if**

This command is to be followed by parentheses and curly brackets. If the final value of the parentheses is the value “false”, the operations contained within the curly brackets are not executed. Otherwise, the operations within the curly brackets are executed. Optionally, the if-statement maybe be followed by an “else” command. Following the “else” are curly brackets that contain operations that are executed only if the operations in the brackets following the “if” command are not executed. The system commands “break” and “return” both end operations for both the “if” and “else” commands.

**while**

The while command is simply the “if” command that is repeated until the condition is shown false. However, no “else” command can follow it. Furthermore, if the command “while” is directly followed by the token “do”, the operations within the curly brackets will be executed once before the result of the condition within the parentheses is checked. Note that the system command “break” will break the loop.

**memory**

Allows creation and management of variables whose names are determined during runtime.

**print**

The values of the objects following this command are displayed on the console. The line-deliminating character \n, backslash, apostrophe, and quotation mark held in the value are special characters. Backslash n ( “\n” ) creates a newline in the console window. Backslash, apostrophe, and quotation mark are displayed as they are; however, they can only be stored as or in a value string if preceded by a single backslash (see Built-in Operators > \ ).

**delete**

This command, followed by a name of an object, deletes the object whose name is given from the vector list of user objects.

**input**

A cursor is displayed in the console window for the user to type. This command returns as a string the line from the user that is typed into the console.

**return**

The next value from the operations to the right of this command is assigned as a value to the function currently being processed. Note that when this is used outside of a function, it applies to the main file, in which case it can be used to return a final value from running a file (see file run).

**file**

File handling. Incomplete functionality.

**break**

Same as in C++

**define**

Like functions you can’t pass parameters to. However, certain system commands (system next and system previous) allow access of data in the same line as the function call.

**math**

Sub-functions “addition”, “subtraction”, “multiplication”, and “division” which act on either numbers or strings depending on whether the specific keyword “str” is given.

**in**

Allows searching for a definition (given by “define”) in another file.

**manip**

String manipulation.