Frogger Language Definition.

**VARIABLES:**

The only variable data types are double, string, and object. Variables are defined at first use and are typed based on usage. Variables of object type default to the values defined within the Object Data File (ODF). Variable identifiers are alpha with underscores starting with an alpha character. So, valid variable names (identifiers) follow the regex [a-zA-Z][a-zA-Z\_]\*, so numerical digits are not allowed.

**OBJECTS:**

All object data are private and functions are public access, accessible through the scope resolution operator (:). A common example is the use of double functions (e.g. 5.0:toString() ). Object types are defined using Object Folders (OFs). Inheritance is facilitated through the Structure Configuration Files (SCFs). Only single inheritance is allowed; that is, only one inheritance record is allowed within any given SCF. If a User-Defined Function (UDF) in the child object match signatures with a UDF in the parent object, the child’s UDF overrides the parent’s. A parent’s overridden UDF can be called through the parent scope resolution operator (::). The parent scope resolution operator can also be used within a UDF as a primary for parent [e.g. MyDouble is a child of double and MyDouble:toString() exists overriding double:toString(); inside MyDouble:toString(), the line “end(::toString());” is valid and calls double:toString()].

**TEMPLATES:**

Objects can be templatized through the Structure Configuration Files (SCFs). If a template is used, the template identifier, defined in the SCF, acts as a placeholder object(s). To instantiate from a template simply replace the placeholder object name(s) with the desired fully-defined object(s). For example, if a Collection object is templatized as %template% T in the SCF, the templatized name is Map{K,T}. Then if a map from a string key to a double type, the instantiated object name would be Map{string,double}.

**BUILT-IN TEMPLATES, OBJECTS, & VARIABLES:**

double - a numerical representation including decimals, default value is 0.

:toString()~string - returns a string representation of the double value.  
(e.g. 65.9:toString() -> ’65.9’)

:toAscii()~string - returns a single character string matching the truncated double value in ascii  
or ‘’ if double value is out of ascii range.  
(e.g. 65.9:toAscii() -> ‘A’)

string - a textual representation, default value is ‘’.

:asciiAt(<double>)~double - returns the ascii value of the char at the given index in the string.   
Zero indexed. [e.g. ‘ABC’:asciiAt(0) -> 65]   
returns 0 if the index DNE

:length()~double - returns the number of characters in the string.

:parseDouble()~double - returns the double value parsed from the string value or 0 if string cannot be parsed.  
(e.g. ‘65’:parseDouble() -> 65)

stringList - a list of strings, default value is empty.

:elementAt(<double>)~string – returns the string at the given index in the list. Zero indexed.  
returns default value if the index is out of range.

:size()~double - returns the number of strings in the list.

:add(<string>)~null - appends the given string to the end of the list.

:remove(<double>)~null - deletes the string at index double from the list or does nothing if index out of bounds.

args=stringList - a list of the command line arguments.

**BUILT-IN COMMANDS:** (Commands have null return type)

end() - terminates the function’s execution, returns null.

end(<string>) - terminates the function’s execution, returns <string>.

end(<double>) - terminates the function’s execution, returns <double>.

display (<string>) - prints the argument to the user.

display (<double>) - prints the argument to the user.

openI(<string>) - opens the input file.

openO(<string>) - opens the output file.

write(<string>) - writes the argument to the output file.

closeI() - closes the input file.

closeO() - closes the output file.

**BUILT-IN FUNCTIONS:**

retrieveDouble()~double - obtains a double value from the user.

random()~double - generates a new pseudo random number between 0 and 1.

retrieveString()~string - obtains a string value from the user.

read() -reads a single ascii character from the input file as a string of len 1.   
(returns -1 ascii character for EOF)

**OPERATORS:**

**Arithmetic operators:** (listed in order of operations) [Left Operand – LO, Right Operand – RO]

exponentiation (^^) - LO is raised to the power of RO (5^^2 = 25)

rootation (##) - RO is rooted to the power of LO (3##8 = 2)

multiplication (\*\*) - Standard multiplication

division (//) - Standard division

integer division (\\) -

modulus division (%%) -

addition (++) - Standard addition

subtraction (--) - Standard subtraction

assignment (=) - RO is evaluated and assigned to LO as a double

**String operators:** (left associative)

string concatenation (++) - Standard concatenation between two strings

assignment (=) - RO is evaluated and assigned to LO as a string

**Boolean operators:** (All standard Boolean operators)

less than (<)

greater than (>)

equal (==)

less than or equal (<=)

greater than or equal (>=)

not (!)

**STRINGS:**

Strings include only printable characters and the listed escape characters enclosed within single quotes. Escape characters: &t (tab), &n (new line), &’ (single quote), and && (ampersand).

No control characters are allowed.

**FILES:**

Frogger supports only simplex file operations; that is, one input and one output file at any given time. Writing to the output file is done through the write(<string>) command. If a number or ascii value is to be written to a file, the programmer must utilize double, string, and ascii conversion functions.

**KEYWORDS:**

if, then, and else.

**COMMENTS:**

Comments are enclosed within tildes (~) and are completely ignored.

**PROCESSING ORDER:**

Each FLOWSTMT is associated with a number (starting at 0 and incremented by 1 until the end of file linearly). Frogger is not a linear language; that is, code is not processed top to bottom. Instead, at the end of each JMPSTMT, control is passed to the FLOWSTMT corresponding to the number obtained by the following process:  
Add up all the printable characters’ ascii codes for the current JMPSTMT (excepting extraneous parens and subsequent spaces within string literals), mod this number by the total number of FLOWSTMTs in the source program. (Note ascii values for comment characters are ignored because comments do not carry over into the CFG. Note also that conditional structures themselves are not included in ascii summation because the JMPSTMTs are structures contained within the conditional structure.)

**PROGRAM STRUCTURE:**

A program is, at minimum, a single program entry function file; but can be as complex as a Project Folder (PF), a program-level Structure Configuration File (SCF), a Program Entry Function File (PEFF), multiple User-Defined Function Files (UDFFs), and multiple Object Folders (OFs).

**Project Folder (PF):**

The PF is named <projectName> and is the root directory for the project. It contains all files related to the project. The PF is optional if the project is a stand-alone PEFF.

**Structure Configuration File (SCF):**

SCFs allow for the use of UDFFs. The project-level SCF is named <projectName>.struct (where <projectName> matches the name of the PF), **must** be stored in the PF root and is optional if the project’s only functional code is the PEFF. Object-level SCFs are named <objectName>.struct (where <objectName> matches the name of the OF) and **must** be stored in the referenced OF.

The SCF ties all elements of the project together. The SCF contains function declaration records, object declaration records, an inheritance record, and a templatization record: one per line.

A templatization record is formatted as follows without spaces:

%template%<templateIdentifier>[,<templateIdentifier>]\*

The parent OF **must** be in the same folder as the current OF. An inheritance record is formatted as follows without spaces:

%parent%<*parentObjectName*>.struct  
A function declaration record is formatted as follows without spaces: *<functionName>*([*argTypeList*])~<*returnType*>.fgr  
An object declaration record is formatted as follows without spaces:  
<*objectName*>.struct

**<functionName>** - the name of the function.

**<objectName>** - the name of the object.

**[argTypeList]** - a comma-separated list of the arguments and their types: *<argName>*=*<type>*

**<argName>** - the name of the argument to be used as the local variable name within the function.

**<returnType>** - the data type to be returned by the function (<*type*> or “null”).

**<type>** - the data type of the argument (“double”, “string”, or <*objectName*>).

**Program Entry Function (PEF) File (PEFF):**

The PEFF is named <projectName>()~null.fgr, **must** be placed in the PF (if a PF is used), and acts as the entry point to the project. The PEF can optionally have a corresponding record in the SCF, but is not required to where its existence, location, and name is implied from the PF (if a PF is used). Command line arguments are only accessible within the PEF and are not treated as arguments in the function record.

**User-Defined Function (UDF) File (UDFF):**

Each UDF is defined in a separate file as <functionName>([argTypeList])~<returnType>.fgr and must have a corresponding record in the SCF. PEFFs and UDFFs follow the Frogger syntax described throughout this document as they contain functional Frogger code.

**Object Folder (OF):**

Each OF represents an object in OOP, is named <objectName>, and must be placed either in the PF or in a parent OF. It contains all files related to the representative object. It must contain an object-level SCF and may contain an ODF, multiple UDFFs, and other OFs.

**Object Data File (ODF):**

ODFs allow for objects to contain data. ODFs are named <objectName>.data and **must** be placed in the OF. The ODF contains data declaration records: one per line.  
A data declaration record is formatted as follows (with or without spaces):  
<*memberName*> = <*type*> [#<*defaultValue*>#];

**<memberName>** - the local name of the object variable.

**<type>** - the data type of the argument (“double”, “string”, or <*objectName*>).

**<defaultValue>** - (optional) the data member’s value upon initialization.  
object-type data members’ <defaultValue> must be empty and is required   
 (e.g. myDate = Date ##;)  
for string or double data members, <defaultValue> must be a valid string or double value respectively.  
 (e.g. score = double #100#; , name = string #‘Joe’#;)  
if no default value is provided for a string or double data member, the data member will be initialized to ‘’ or 0 respectively.

**CFG**:

1. PROG -> FLOWSTMT FLOWSTMTS
2. FLOWSTMTS -> FLOWSTMT FLOWSTMTS
3. |
4. FLOWSTMT -> IFSTMT
5. | JMPSTMT
6. NESTEDFLOWSTMT -> IFSTMT
7. |JMPSTMT

**Control:**

1. IFSTMT -> if ( BOOLEXP ) then NESTEDFLOWSTMT else NESTEDFLOWSTMT
2. BOOLEXP -> EXPR BOOLOP EXPR
3. | EXPR not BOOLOP EXPR
4. BOOLOP -> lt
5. | gt
6. | eq
7. | lte
8. | gte

**Action Statements:**

1. JMPSTMT -> [::] id ( [ARGLIST] );
2. | TYPEDTERM : id ( [ARGLIST] );
3. | TYPEDTERM :: id ( [ARGLIST] );
4. | id assign EXPR ;
5. ARGLIST -> EXPR , ARGLIST
6. | EXPR

**Expressions:**

1. EXPR -> EXPR ADDOP ADDTERM
2. | ADDTERM
3. ADDTERM -> ADDTERM MULOP MULTERM
4. | MULTERM
5. MULTERM -> MULTERM EXPOP TYPEDTERM
6. | TYPEDTERM
7. TYPEDTERM -> TYPEDTERM : id ( [ARGLIST] )
8. | TYPEDTERM :: id ( [ARGLIST] )
9. | PRIMARY
10. PRIMARY -> dbl
11. | id [( [ARGLIST] )]
12. | :: id ( [ARGLIST] )
13. | string
14. | ( EXPR )

**Operators:**

1. ADDOP -> add
2. | sub
3. MULOP -> mul
4. | div
5. | mod
6. | idiv
7. EXPOP -> rt
8. | exp

**OBFUSCATION:**

Frogger offers in-line obfuscation if the programmer should choose to further confuse herself/himself. Including the first line of source code as strictly an even number of tildes(~) followed by a carriage return will trigger the de-obfuscator. Note: 0 is considered an even number so if the first character in the source code is a carriage return, the de-obfuscator will run.

Obfuscator Examples (<\n> denotes the new line character within the source code):   
<\n> ~~<\n> ~~~~<\n> ~~~~~~<\n>  
Non-Obfuscated Examples:  
~<\n> (odd number of tildes) ~a~<\n> (‘a’ is not a tilde character) ~~ <\n> (space is not a tilde)

Obfuscation is as follows:  
Each character within variable names should be ascii incremented based on the number of identifiers occurring prior in the source code. Similarly for each keyword and routine name (functions and commands are both routines) but based on the number previously occurring (each category is treated separately). The de-obfuscator will decrement by these counters. Valid symbols are restricted to alphanumeric and the underscore and are incremented in order of ascii value, so order is 0-9A-Z\_a-z. Then if a variable named x\_Dbl is to be used and 6 variables have been used between the start of file and the current location, x\_Dbl should be represented (+7) as 4gKis.