Sheets Language Reference Manual

- Amelia Brunner - `arb2196`
- Gabriel Blanco - `gab2135`
- Ruchir Khaitan - `rk2660`
- Benjamin Barg - `bbb2123`

1. Introduction

The *Graphics Processing Unit* (GPU) was invented in 1999 as a single-chip processor that allowed the CPU to offload graphics-intensive tasks to a separate processor. Unlike the CPU, which is built to contain only a handful of cores but a lot of cache memory, a GPU has limited memory but hundreds of cores, which, thanks to very efficient context switching, allows it to handle thousands of threads concurrently without significantly degrading the performance of the CPU. In the past, GPUs were seen as a luxury reserved for video processing and computer gaming. However, because of the advent of larger screen displays and a greater demand for video, image and signal processing applications, GPUs are quickly becoming more mainstream.

Although we are seeing more and more applications take advantage of the computational capabilities of the GPU, it is still very difficult to program for the GPU because of the many different types of GPU architectures and chip specific proprietary software. *Sheets* empowers engineers to take a high-level approach to programming on the GPU. With syntax, semantics and program structure to help programmers run parallelizable algorithms on the GPU, *Sheets* is a hardware-portable language that allows users to leverage the GPU's ability to handle large-vector data operations without needing to worry about hardware specifications. It does so by compiling down into *OpenCL*, an open-source programming language that can run on a wide variety of GPUs.

2. Types

2.1 Primitive Types

Integer Types

These are simply signed (two's complement) integral numbers (fixed precision) whose literals are sequences of digits. The difference between these types is their size in bytes, which correspondingly limits the range of numbers they can represent.

- `int`

Signed 32 bit integer type. An integer literal is a sequence of digits that fit within a 32 bit range.

- `long`

Signed 64 bit integer type. An integer literal is a sequence of digits that fit within a 64 bit range.

Floating Point Types

The two types in this category correspond to IEEE single-precision and double precision floating point numbers, as defined in IEEE 754. s Floating point constants consist of an integer part, a decimal point, and a fraction part.

- `float`

Single precision floating point type, with a size of 32 bits.

- `double`

Double precision floating point type, with a size of 64 bits.

- `char`

A single character is 1 byte wide, which can be alphanumeric or a form of punctuation, or any other valid character that is escaped with a backslash '\' (see section 3.3).

Vector/Array Types

Vector/Array types represent multiple instances of primitive types allocated in contiguous ranges of memory, either on regular machine stack, heap, or in GPU global/local memory. Array allocation and data transfer between CPU/GPU is handled automatically.

Arrays are zero indexed and can be accessed with the square-bracket notation, so (for example) `array[7]` returns the element at index 7 of the variable `array`.

Arrays can be multi-dimensional, and can be indexed by separating the dimensional index numbers by commas so 'array[2, 5]' accesses row index 3, column index 5 of the two dimensional array. Arrays are limited to having at most 3 dimensions in the Sheets language.

Single dimension arrays can be defined as follows:

```
<type T> arrayName [size]
```

This will allocate an empty array of type T with size elements. The size parameter is optional, and arrays can be initialized as follows:

```
<type T> arrayName[] = [element, element, element, ...  ]
```

If you do give a size parameter, the number of elements within the right value [] must be less than or equal to the size parameter. If it is less, then the remaining spaces in the array are initialized to zero.

Multi-dimensional arrays

Multi-dimensional arrays are very similar to single-dimensional arrays/vectors:

```
<type T> arrayName [num_rows, num_cols]
```

This allocates an empty 2D array of type T with num_rows rows and num_cols columns for a total of num_rows * num_cols elements.

Again, the size parameters are optional, and arrays can be initialized as follows:

```
<type T> arrayName[size1,size2] = [[element, element],  [element, element,
element],  ...  ]
```

Here, all of the subarrays don't have to be the same length, and the array is initialized to be of dimensions `(maximum_subarray_size * number of subarrays)` with blank elements again initialized as zero. Note that this example is of a 2-dimensional aray, but 3-dimensional arrays are also supported with the same syntax.

Note that Sheets does not support arrays of arbitrary types. The total list of supported array types are as follows:

- `int[]` - Contiguous array of integers as defined above
- `long[]` - Contiguous array of longs
- `double[]` - Contiguous arrays of doubles
- `float[]` - Contiguous arrays of floats
- `char[]` - Contiguous arrays of chars. Used in the underlying representation of Strings as well.

2.2 Non-Primitive Types

- `String` - Defined as a wrapper over a character array, with an integer specifying length. Can be ASCII or Unicode encoded. String literals are defined as a sequence of characters enclosed by double quotes.
- `struct` - A programmer defined data type consisting of variables of primitive types, and

other structure data types. The size of a struct is large enough to hold all members of the struct.

A structure is defined with the 'struct' keyword, followed by a name, followed by declarations of all the variables inside the struct, enclosed within an opening curly-brace '{' and a closing curly brace '}'.

You can declare an instance of a struct as:

```
struct <struct_name> <var_name>
```

This instantiates a struct <struct_name> named <var_name>. You can access elements of a struct with a dot '.' and then the variable name. For example:

```
<var_name>.<element_name>
```

This accesses a variable named <element_name> inside of a struct named <var_name>.

- `Block` - Defined as a struct used for metadata holding input/output information for functions executed on the GPU. See section 5.1

2.3 Casting

Casting is allowed between:

- Any two numbers
- doubles/floats to longs/ints lose fractional precision
- longs/doubles to ints/floats get truncated
- No casting between primitives and non-primitives
- chars can be casted to ints/longs/floats/doubles
- longs/ints can be casted to chars (preserves least significant 8 bits)

3. Lexical Convections

3.1 Identifiers

Identifiers refer to a variable, function, or function argument. They must begin with alphabetic character or underscore, but the rest of the identifier can be alphanumeric or underscores. Capital and lowercase letters are considered distinct. We reject dashes.

3.2 Keywords

- `if(` *boolean condition* `):`
  - execute the following block if the boolean condition is true
- `elif(` *boolean expression* `):`
  - following exactly one `if` statement and one or more `elif` statements, execute the following block if the *boolean condition* is true and all conditions in the preceding chain were false
- `else`
  - following exactly one `if` statement and zero or more `elif` statements, execute the following block if all conditions in said chain were false
- `while (` *boolean condition* `):`
  - execute the loop contents until the *boolean condition*, evaluated at the start of each iteration, is false
- `for` loop execution in two formats:
  - `for` *var* `in` *list* `:`
    - iterate through each item in a list
  - `for (` *assignment* `;` *boolean condition* `;` *iterative step* `):`
    - Assign a variable and loop on contents, performing the *iterative step* at the end of each loop, until the *boolean condition* (evaluated at the start of each iteration) is false
- `break`
  - Jump out of the current scope and continue execution in the parent scope (invalid when used outside of a loop). Must be the only expression on its line. Any symbols following it on its line will be ignored.
- `continue`
  - Skip forwards to the next iteration of the enclosing loop (invalid when used outside of a loop). Must be the only expresion on its line. Any symbols following it on its line will be ignored.
- `TRUE`
  - Constant "true" (i.e. a char with value '0') for use in boolean expressions.
- `FALSE`
  - Constant "false" (i.e. a char with a value of '1') for use in boolean expressions.
- `NULL`
  - Value of an unitialized object.
- `return` *expression*
  - In a `func` block, return *expression* by value to the caller.
- `const`
  - Variable identifier that indicates to the compiler that the variable cannot be modified.
- `func` *type identifier* `(` *type identifier* `,` *type identifier* `,` ... `):`
  - Define a standard CPU function taking 0 or more arguments and return a value of a given *type*. Tabs and spaces between all components of the expression are ignored. *type* may be `void`.

- `gfunc`
  - Define a function that will run on the GPU (see section 5, GPU Functions, for a detailed description of the syntax).
- `main`
  - Name of the entry-point function. Every Sheets program must have a function called `main`. `main` must return a result of type `int`.
- `void`
  - A type keyword used solely in `func` declaractions indicating that the function does not return a value.

3.3 Literals

- `int` literals
  - an unbounded string of numerals without a decimal point with an optional sign character
- `float` literals
  - an unbounded string of numerals before and after a decimal point with an option sign character

For both integer and float literals, maximum representable values are limited by the underlying system's OpenCL implementation.

- `char` literals
  - Some characters require escaping because they already have a syntactic meaning in the language, or because their representation is rejected by Sheets. Character literals for these characters are expressed by a pair of single quotes surrounding one of the following expressions:
    - `\'` - single quote
    - `\"` - double quote, also referred to as a quotation mark
    - `\n` - newline
    - `\t` - horizontal tab
    - `\\` - backslash
  - for all other ASCII characters, the literal is expresssed a pair of single quotes surrounding the character
- string literal
  - A sequence of ASCII characters (excepting those who have corresponding character literals) and character literals
- single dimensional array literal
  - An opening `[` followed by comma-delimited float and/or integer literals, followed by a `]`. If the array literal contains only float or only integer literals, it will be of the respective type, but a mix will always be interpreted as a float array (integer literals will be casted to floats). Whitespace between brackets, commas, and int/float

literals is ignored.
- multi-dimensional array literals
- An opening '[' followed by a comma delimited series of single or multi-dimensional array literals, all of the same dimensionality, but not necessarily the same size, followed by a closing ']'. The literal has dimension 1 + dimension of subarrays, and has size (number of subarrays * size of largest subarray)

3.4 Punctuation

- `,`
  - function parameters (see 3.2 Keywordss)
  - array literal separation (see 3.3 Literals)
- `[]`
  - array literal declaration (see 3.3 Literals)
  - array access
- `()`
  - expression precedence
  - conditional parameter
  - function arguments
  - type casting
- `:`
  - start of scoped block (`if`, `else`, `elif`, `while`, `for`, `func`, `gfunc`)
- `'`
  - character literal declaration
- `"`
  - string literal declaration

3.5 Comments

It's like threads coming out of a sheet!

```
# for inline comments

#~ for nested comments ~#

#~
 ~ for long nested
 ~#
```

In an individual line, all characters after a `#` are ignored by the compiler unless the `#` is a part of a string or character literal that is not itself part of a comment.

All text from `#~` to the next `~#` is ignored, excepting those occurences of `#~` and `~#` that appear in a string literal that is not itseslf part of a comment.

3.6 Operators

Sheets includes basic arithmetic operators for both scalar and array types. **Because Sheets' intended use case is with arrays that are too large to efficiently manipulate on the CPU, the vector operations will always run on the GPU.** If the user wants vectorized operations to be executed on the CPU, they must use a loop keyword and implement their own operation.

3.6.1 Operator List

```
.   Access

*   Multiplication              :*  Vector multiplication
/   Division                    :/  Vector division
%   Mod                         :%  Vector mod
+   Addition                    :+  Vector addition
-   Subtraction                 :-  Vector subtraction

^   XOR                         :^  Vector XOR
&   AND                         :&  Vector AND
|   OR                          :|  Vector OR
~   NOT                         :~  Vector NOT
>>  Right shift                 :>> Vector right shift
<<  Left shift                  :<< Vector left shift

=   Assignment                  :=  Vector assignment
!   Negation                    :!  Vector negation

==  Equivalence                 :== Vector equivalence
!=  Non-equivalence            :!= Vector non-equivalence
<   Less than                   :<  Vector less-than
>   Greater than                :>  Vector greater-than
<=  Less than or equal to       :<= Vector less-than-or-equal-to
>=  Greater than or equal to    :>= Vector greater-than-or-equal-to
```

3.6.2 Operator Limitations

An expression may include *either* scalar or vector operators *but not both*. This restriction is intended to force the user to package GPU operations into the largest chunks possible to minimize the number of times that operations must be sent to and retrieved from the GPU.

Vector operations may be applied in the following orders:

1. *array vector_operator array*
2. *array vector_operator scalar*

Any attempt to apply vector operations to arrays of unequal size results in a compiler error.

For case **1**, the result is an array for which the *i*th entry is equal to:

scalar (*i*th value of lhs) *scalar_operator scalar* (*i*th value of rhs)

For case **2**, the result is an array for which the *i*th entry is equal to:

scalar scalar_operator scalar (*i*th value of rhs)

3.6.3 Operator Precedence

Within the two groups, order of precedence will be the same as for C. The comparison operators will be treated the same as in C.

3.7 Whitespace

Symbols that are considered to be whitespace are blank, tab, and newline characters. Blank characters will be used for token delimination within lines of the program. Additionally, any blank characters directly following newline characters will be used to block out functions (which is discussed more in depth in section 4.4: Scope). Tab characters are ignored.

4. Syntax

4.1 Program Structure

A program consists of a main function which can call any other functions within the program namespace or declared inside the program.

- The `main` function is where the execution of the program begins. If there is no `main` function, the compiler will throw an error.
- You can declare other functions besides `main`, however if they are not directly or indirectly called by `main`, then the code within them will not be executed.
- There are two types of functions: `gfuncs` and `funcs`. Both can be called from main, although how they are executed will be different. The order in which you declare `gfuncs` and `funcs` in your program does not matter.
- Sheets is not object oriented, and therefore there are no such things as classes.

4.2 Expressions

Expressions in *Sheets* can be primary, unary or binary, with precedence being given in that order, from highest to lowest. The rules of precedence and associativity differ amongst

operators within each category, and so we will explicitly state the cases in which one operator will bind tighter than another.

4.2.1 Assignment

There are two assignment operators in *Sheets*, `=` and `:=`, and syntactically they behave in the same way. Both are binary operators that are right-binding. The value of the expression on the right is stored in the variable on the left hand side. The standard assignment operator (`=`) simply assigns the value to the left-hand variable, whereas the GPU assignment operator (`:=`) copies values from one array to another using parallel processing.

```
# Simple Assignment
a = 2

# Parallel Assignment
int[] A = [0,1,2,3,4,5]
int[] B := A
```

4.2.2 Arithmetic

For arithmetic expressions, you have an operator representing a simple mathematical operation and then the equivalent vector operation which performs the same arithmetic operation but on each element of the array. All arithmetic operations are left-associative by default, but that associativity can be superseded by standard mathematical order of operation: multiplication and division bind tighter than addition and subtraction, and mod (`%`) binds tightest of all.

```
%   Mod                 :%  Vector mod
*   Multiplication      :*  Vector multiplication
/   Division            :/  Vector division
+   Addition            :+  Vector addition
-   Subtraction         :-  Vector subtraction
```

For vector arithmetic, you get a different type of operation depending on the types of the operands:

For a vector operation on an **array and a scalar** (both of the same type), the scalar is applied to each element in the array via the operator (i.e. A :+ 3 would yeild an array in which each element was the same as it was in A but with the addition of 3). Note that in this case, the array must be the left-hand operand and the scalar must be the right-hand operand.

Vector operations on **two arrays**, which must be of the same type and size, return an array where the vector operation was performed on each corresponding element in both arrays.

For example:

```
int[] A = [0,1,2,3]
int[] B = [3,2,1,0]

int[] C = A :* B
# C has value [0,2,2,0]
```

If an arithmetic vector operation is called on **two scalar values**, e.g. `3 :+ 2`, the compiler will throw a warning but still calculate the operation on the GPU. This is a bad practice, and much less efficient than simply using a standard operation.

4.2.3 Comparison Operators

Comparisons are binary operators, returning a boolean value based on how the right-hand-side of the expression compares to the left-hand-side.

```
==  equivalence                 :== Vector equivalence
!=  non-equivalence             :!= Vector non-equivalence
<   less-than                   :<  Vector less-than
>   greater-than                :>  Vector greater-than
<=  less-than-or-equal-to       :<= Vector less-than-or-equal-to
>=  greater-than-or-equal-to    :>= Vector greater-than-or-equal-to
```

Vector less-than (`:<`), greater-than (`:>`), less-than-or-equal-to (`:<=`) and greater-than-or-equal-to (`:>=`) all behave similar to the arithmetic vector operations, where the standard version of the operation is applied element by element and returns an array of booleans. If the two types are both arrays, then it will compare each element one by one, returning the boolean.

```
int[] A = [1,2,3,4,5]
int[] B = A :< 3

# B has value of [TRUE,TRUE,FALSE,FALSE]
```

However, equivalence (`:==`) and non-equivalence (`:!=`) behave slightly differently from the rest of the comparison operators. Array-to-scalar operations behave as you would expect, returning an array of booleans for the result of the operation, element by element.

However, for array-to-array vector operations, instead of returning an array of booleans, these only return one char value, which represents the total outcome of the operation (0 for true, 1 for false). The expression `A :== B` asks if `int[] A` contains all the same values as `int[] B`,

and returns `TRUE` if it is the case, `FALSE` otherwise. This allows a clean and simple way to do array content comparisons.

4.2.4 Logical

Logical operators test the logical truth of expression.

```
&&   AND                        :&& Vector Boolean AND
||   OR                         :|| Vector Boolean OR
```

AND (`&&`) and OR (`||`) compares the boolean values of its two operands. In the case of shifting, the right-hand operand must be a fixed point number. The vector version simply performs a given logical operation on every element of the array.

4.2.5 Bitwise

Bitwise operators apply bit operations to the operands on either side of the operators.

```
&   AND                         :& Vector AND
|   OR                          :| Vector OR
^   XOR                        :^  Vector XOR

>>  Right shift                :>> Vector right shift
<<  Left shift                 :<< Vector left shift

~   NOT                        :~  Vector NOT
```

AND (`&`), OR (`|`) and XOR (`^`) are binary operators that apply corresponding bit operations from the right operand to the left one (i.e. following left associativity). Left shift (`<<`) and right shift (`>>`) are also binary operators, and they apply the a bit-level shift to the contents of left-hand-side operand.

Not (`~`) is a unary operator which, as previously stated, takes precedence over binary operators. It returns a negated version of the operand, where all `FALSE` values are now `TRUE`, and `TRUE` values are `FALSE`.

For all of the corresponding Vecotr operations, the same bit logic is applied to the left-hand vector operand in the same fashion as the other vector operators.

4.3 Statements

A statement is a complete instruction that can be interpreted by the computer. Unless otherwise specified, statements are executed sequentially within a function.

4.3.1 Expression statements

Expression statements are the most common type of statement. Because whitespace has syntactic meaning in *Sheets*, a statement is ended by a newline (`\n`). You can ignore a newline and continue an expression statement on the next line by using the continuation operator (`...`) at the end of a line.

Expression statements can include any of the expressions previously covered. The only exception is that *Sheets* explicitly does not allow the mixed use of vector and non-vector operations in the same statement. This is an enforced standard which forces users to write better parallelizable functions.

4.3.2 Conditional statements

Conditional statements check the truth condition of an expression, and then choose a set of statements to execute depending on the result. Here is a common implementation of an `if`/`else` conditional statement

```
if (expression):
    statement
elif (expression):
    statement
else:
    statement
```

Only the `if` statement is required, you can choose not to account for other conditions or to account for any number of additional conditions using `elif`. The `else` statements execute only if none of the preceding conditions returned true.

4.3.3 Loop Statements

while/for

Statements such as `while` or `for` allow you to iterate over blocks of code. In the case of `while` loops, the controlling expression is checked every time before the execution of the body of the `while` loop. `for` loops have two syntaxes:

The first one has more functionality but is more verbose, that is:

```
for (expression1; expression2; expression3):
    ...
```

Where expression 1 initializes the loop counter (an integer), expression 2 gives the controlling

expression that is checked before every execution of the `for` loop body, and the third expression gives the action performed on the counting element after every execution of the loop -- usually incrementation.

Because of how often we iterate over vectors in *Sheets*, there is one additional syntax for a `for` loop that is much cleaner and easier to write:

```
for expression1 in vector:
    ...
```

Expression 1, similar to in the first syntax, is the declaration of a loop counter. This creates a loop that iterates from 0 to n-1, where n is the length of the array. This allows you to easily access the index of every element in the array without having to worry about fencepost errors.

4.3.4 Loop Interruption Statements

There are two statements that allow you to interrupt the execution of statements in the body of either a `for` or a `while` loop. These are `break` and `continue`.

The `break` statement will end the execution of the body of a loop, then exit the loop completely as if the controlling expression returned `FALSE`.

The `continue` statement also ends the execution of the body of a loop, but then returns to the top of the body to evaluate the controlling expression to see if the loop should continue.

Anything on the line following `break` or `continue` will be ignored.

4.3.5 Return Statements

Ends the execution of a function, including the `main` function. If a function does not have a `return` statement at the end, it is assumed to be a void function without a return type. GPU functions (`gfunc`) do not return a value, instead they write to a an output block in memory, and therefore a `return` statement within the function does not cause the function to return a value; instead, it just ends the execution of that GPU function at a given statement.

4.3.6 Function Statements

Function statements call a function, returning a value if the function itself has a return type. Statements that invoke GPU functions will always block until every thread started by the `gfunc` has returned.

4.4 Scope

One of the difficulties with working simultaneously in the CPU and GPU domain is that each has its own private memory which the other cannot access, but can be sent back and forth via a

memory bus. For those reasons, we limit the amount of memory that is shared/transferred between both domains.

4.4.1 Scope within the GPU

GPU memory hierarchy

Because of the memory limitations we mentioned, `gfuncs` do not have direct access variables declared in CPU memory space; this includes `global` variables. In order to pass data from the CPU into a `gfunc`, it has to be passed through the arguments of the function. The only caveat to this rule is that *Sheets* will implicitly pass a small set of parameters to the GPU and store them within the `block` environmental variable. These include things like `block.size`, `block.index` and `block.input_length`.

4.4.2 Scope within the CPU

Variables that are denoted as `global` can be declared outside of a function and accessed within any CPU function. "Global" in this case only refers to being global in the CPU memory space, not within the GPU.

Sheets will use block scoping, such that any variable defined within a given indentation level is accessible within that level and any level of indentation greater than that level.

5 GPU Functions (`gfunc`)

`gfunc` defines a function that will be run on the GPU. The contents of a `gfunc` are compiled into an OpenCL kernel and linked into a Sheets executable via the sheets runtime library.

Our goal for the `gfunc` keyword is to shield the programmer from writing blatantly unparallelizable code but to leave enough freedom that the resulting code is at least reasonably optimal. As such, we enforce the following rules:

- Any call to a `gfunc` is blocking.
- `gfunc` function arguments are immutable.
- `gfunc` function arguments are limited to arrays of equivalent size and scalar types (i.e. int, long, double, or float), and a `gfunc` must take at least one array as an argument.
- There is no return value from a `gfunc`; it will write to a global output array that must be the same size as the first array of the arguments.
- The code within a `gfunc` performs accesses and writes to the output array through an environment variable called `block` (see 5.1 "The `block` Keyword")
- Vector operations as described in sections 3 and 4 cannot be called from within a `gfunc`, and doing so will result in a compile error. For example, the following code would not be accepted by the Sheets compiler:

```
gfunc gadd(int[] A, int[] B):
    A :+ B
```

A `gfunc` is thus declared:

`gfunc` *array_type identifier* `(` *array_type identifier* `,` *type identifier* `,` ... `):`

5.1 The `block` Keyword

In OpenCL, a kernel represents the smallest concurrently-executable chunk into which a problem may be divided. Through this structure, OpenCL forces programmers to conceive of their problem as a number (hopefully a large one) of concurrently-executable sub-problems that work with some subset of the entire input data.

Sheets encouragess the programmer to conceive of their parallelizable problem according to this "sub-problem" format. To do this, Sheet provides the `block` keyword as a reference to the aforementioned "smallest concurrently-executable chunk" of the problem.

`block` is a struct containing:

- `block.size`
  - The number of elements referenced by the subproblem. Stored as an `unsigned int`.
- `block.id`
  - The identifier of this instance of the subproblem, used for calculating the indices of the input array(s) to access.
- `block.out`
  - An array of size `block.size` that maps to the writable region of the output array in the block.

5.2 Calling a `gfunc`

A `gfunc` is called just like a standard `func` except for an optional argument at the end indicating the block size for this particular call.

```
my_gpu_function(arg1, arg2, arg3).[block_size]
```

If the `.[]` is omitted, the function is called with a `block` size of 1. The `.[]` argument has several restrictions:

- `block_size` must be a positive integer.
- `block_size` may not exceed the input array size.

- `block_size` must evenly divide the input array size.

6 Standard Library Functions

Sheets will provide some simple library functions to assist with array manipulation and standard input output operations.

6.1) Vector Operations

The number of vector library functions that we provide in Sheets is limited for a purpose; we want to encourage our users to take advantage of how easy it is to write operations related to vectors on their own usinfg `gfunc`s! That being said, we also want to help our users out with a few extremely common operations:

- `reverse()` - performs an in-place reversal of an array
- `length()` - returns the dimensionality of the array (number of elements)

Note that these functions will be executed on the GPU.

6.2) File I/O

Sheets will have a few relatively simple I/O functions that take string literals or other variables as arguments, and perform print/reads either to/from terminals or to a specified file descriptor. These functions mirror the behavior of comparable functions in standard C libraries.

- `int open(String)` - opens a file specified by the given path, and returns a file descriptor that can be read from or written to.
- `void print(int, String)` - write the contents of the String buffer specified in the second argument to the file descriptor specified in the first argument.
- `void read(int, String)` - read from stdin or another file descriptor, as specified in the first argument, into the String buffer specified in the second argument.
- `int write(int, array)` - write binary (non null terminated arrays), as specified in the second argument, to the file descriptor specified in the first argument, and return the number of bytes successfully written. Note that for brevity, we have written 'array' as the second argument in the function signature, which we are using to represent all possible array types.