Language Reference Manual

1 Preface

This language reference manual describes the Civ language, developed by Mikhail Klimentov, Michael Nguyen, Prateek Sinha, Yuchen Zeng, and Eli Bogom-Shanon for Stephen Edwards's Programming Languages and Translators class (W4115).

For the most part, this document follows an organizational precedent set by Brian Kernighan and Dennis Ritchie in their "The C Programming Language."

2 Civ Overview

Civ's purpose is to provide a simplified version of C that enables a user to quickly grasp fundamental programming concepts, such as control structures, data types, functions, and so on and so forth.

However, there is a distinct difference between Civ and C, and that is that Civ has no explicit usage of pointers in its syntax. This means that the symbol *** is almost never used in Civ outside of exponentiation and multiplication functions. And because there are no explicit pointers, there are also no explicit references using the & symbol.

Consider the following code for swapping two numbers written in C:

```
void swap(int *a,int *b);

int main(){
        int num1=5,num2=10;
        swap(&num1,&num2);

//Passing in two addresses as arguments
        return 0;
}
void swap(int *a,int *b){  //Two pointers at two addresses
        int temp;

//temp holds num1
        temp=*a;

//&num1 holds num2
        *a=*b;

//&num2 holds num1
        *b=temp;
}
```

Now consider the following code written in Civ:

```
int main(){
        float num1=5,num2=10;
        num1,num2= num2,num1;
        return 0;
}
```

While it is true we have NOT swapped addresses (as that is not possible in Civ), we have effectively achieved what we wanted: a swap of values in variables. But rather than actually swapping addresses, we are actually going to manipulate the symbol table such that the values of num1 and num2 change while staying in the same memory allocation.

3 Lexical Conventions

3.1 Comments

Comments are styled after the C multiline comments. Open with /* and close with */. Civ does NOT have single line comments, nor nested comments.

```
/* This is a comment in Civ */

/*
This is a multiline comment in Civ
*/

// ILLEGAL: This is NOT a comment in Civ

/*/*This is also ILLEGAL*/*/
```

3.2 Identifiers

In Civ, an identifier is an alphanumeric string used for any variables, functions, data definitions, etc. Identifiers cannot begin with symbols (ex: _toast would not be a valid identifier). Upper case and lower case letters are distinct in Civ. It is advised that identifiers be written in mixedCase notation.

3.3 Keywords

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

```
bool      print
break     return
do        static
else      string
float     struct
for       while
goto      void
if
```

3.4 Constants

There are no constants in Civ — everything is mutable.

3.5 Literals

A literal in Civ is a literal like anywhere else: any raw data that is presented symbolically is a literal. For example, 2 represents the float 2.0. Literally are useful as they provide an alternative to references, and can be used in pass by argument parameters for functions.

3.6 Punctuation

Civ is meant to look as close to C as possible, with the exception of explicit pointer notation.

[ ] - Brackets are used as indices for lists and list declarations.

( ) - Parantheses are used in function calls to surround the function arguments.

{ } - Curly braces are used to indicate the beginning end of a body or block statement.

, - Commas are used in lists generation and as a separator between inline statements.

: - Colons indicate the beginning of a declaration, and is usually followed by an opening curly brace.

; - Semicolons tell the compiler that the statement, expression, or body/block is complete.

3.7 Operators

Operator	Use	Associativity
+	Addition	Left
-	Subtraction	Left
*	Multiplication	Left
/	Division	Left
%	Modulus	Left
=	Assignment	Right
==	Equal to	Non-associative
!=	Not equal to	Non-associative
<	Less than	Non-associative
>	Less than	Non-associative
<=	Less than or equal to	Non-associative
>=	Less than or equal to	Non-associative
!	Not	Right
&&	And	Non-associative
||	Or	Non-associative

Table 1: Operators

The precedence of operations is shown as below, from greatest to least precedence:

```
  * / %
   + -
 !  && ||
<  >  <=   >=
  ==   !=
     =
```

4 Types

4.1 Primitive Data Types

Boolean In Civ, a boolean is defined by the values `true` and `false`.

```
if(true){print("Test")}
Test

if(false){fork();};
/*No response*/
```

Float All numbers in Civ are floats by default. In the event that the float has no significant digits past the decimal point, it is displayed as an integer, e.g.,

```
print(5 + 5);
10

print(5.0 + 5.1);
10.1

print(5 + 5.1);
10.1

print(4.5 + 4.5);
9
```

Char In Civ, chars are a primitive data type. Chars are used in arrays to make strings.

```
char string[50];
/* Creates an array of length 50 */

char greeting[] = "Hi";
/*
greeting is an array with values: H, i, and \0
*/
```

4.2 Type Conversions and Type Inference

Note that the following conversions happen based off of the grammar. The only time values are implicitly type casted is when the compiler is expecting a certain data type it gets something else.

String to Boolean Any string that is NOT an empty string is treated as a positive boolean, i.e. True. A string that is empty is automatically regarded as false.

```
        if("test"){print("This works.");};
        if(""){print("Unreachable code.");};
```

Float to String Printing a float automatically converts the float to a string. If the float has no numbers past the decimal, it is presented as an integer in the print statement. This does NOT hold for string concatenation, or any other string operation.

All Other Conversions The following type pairs do not convert automatically:

- String to Float
- Boolean to String
- Boolean to Float
- Float to String.

In the event the compiler catches such a situation, an error will be thrown.

4.3 Data Structures

There is only one fundamental data structure that Civ provides, and that is a mutable array. It follows standard C declaration procedures, e.g.,

```
        String test[10];
        String hw[2] = \{"Hello", "world!"\};
        print(hw[0]) /*Returns "Hello"*/
```

There are some fundamental differences in Civ due to lack of pointers. For example, consider the following C code:

```
        char *word;
        char **sentence;
        char ***chapter;
        char ****book;
        char *****library;
```

This is obviously strange C code, but it shows how powerful pointers can be. In Civ, a multidimensional-array would be used, and accessing a the fifth book's fourth chapter's third sentence's second word would look like this:

```
        library[5][4][3][2] = "new word";
```

Because of these differences, every array access is ALWAYS a reference to the original array, and consequently, just like variables, arrays are always pass by reference. The fact that they are mutable in structure and size also aids array manipulation.

5 Workflow

5.1 Overview and Main Differences from C

Pass by Reference - Civ ALWAYS passes by reference if a primitive isn't provided. This is fundamentally different in C, as pass by argument and pass by reference are both allowed, as passing a pointer serves as passing by reference. One might argue that because they are passing a pointer address, it is actually pass by reference value, but that is outside the scope of Civ.

Declarations - Civ does NOT require declarations of variables at the head of every body.

5.2 Variable Scope

- Civ variables strictly have lexical scope, thus making every body closure have its own symbol table.

5.3 Explicit Typecasting

- Civ's type declarations are also functions that return a type casted variable or literal where valid. For example,

```
str s = "Hello";
float x = 2;
float y = 3.3;
bool t = true;

print(str(x));  /*2*/
print(str(y));  /*3.3*/
print(str(t));  /*true*/

float s = float(s); /*Throws error*/
float x = float(x); /*No change in memory*/
float y = float(y); /*No change in memory*/
float t = float(t); /*t = 1*/

bool(s); /*Returns true*/
bool(x); /*Returns true*/
bool(y); /*Returns true*/
bool(t); /*Returns true*/
```

6 Code Samples

6.1 FizzBuzz

6.2 BubbleSort

6.3 Primality Testing

6.4 DP Fibonacci