Kazm Language Reference Manual

Aapeli Vuorinen (oav2108) - Systems Architect
Katie Kim (jk4534) - Manager
Molly McNutt (mrm2234) - Tester
Zhonglin Yang (zy2496) - Language Guru

October 2021

Contents

1 Introduction **2**

2 Lexical Conventions **2**
2.1 Tokens . . . 2
2.1.1 Identifiers . . . 2
2.1.2 Keywords . . . 2
2.1.3 Operators . . . 2
2.1.4 Literals . . . 3
2.1.5 Delimiters . . . 3

3 Types **3**
3.1 Default Types . . . 3
3.2 Data Structures . . . 4
3.2.1 Tuple . . . 4
3.2.2 Array . . . 4
3.3 A Comment on Pointers and References . . . 4

4 Operators **4**
4.1 Arithmetic Operators . . . 4
4.2 Relational Operators . . . 5
4.3 Logical Operators . . . 5
4.4 Expression Operators . . . 5
4.5 Assignment Operators . . . 5

5 Declarations **6**

6 Statements **6**
6.1 If, Else If, Else . . . 6
6.2 While . . . 8
6.3 For . . . 8
6.4 Break . . . 9

7 Functions **9**

8 Scope of Variables **9**

9 Classes **10**
9.1 String Class – Part of the Standard Library 11

10 Example Programs **12**

1 Introduction

The Kazm programming language is a statically and strongly typed programming language which extends the C programming language with lightweight classes implemented as C-style structs that support methods and instances. General purpose programming functionality such as input/output will be provided through built-ins and the inclusion of existing C libraries; however, we will not provide class inheritance or private methods.

Our motivation is to bridge the gap between `C` and `C++`.

2 Lexical Conventions

2.1 Tokens

There are 5 types of tokens: identifiers, keywords, operators, literals, and delimiters. All white space, except for white space separating tokens or within strings, will be ignored. Comments will also be ignored.

2.1.1 Identifiers

A Kazm identifier is a sequence of one or more case sensitive ASCII letters, digits, or underscore '_' characters. Identifiers must not begin with a digit. Identifiers are used for class or variable types.

2.1.2 Keywords

The following are keywords in the language and cannot be used in other contexts.

```
int, double, void, bool, char, String, tuple, if, else, for, while, me, break,
return, true, false, class
```

2.1.3 Operators

Kazm supports the following operators:

- Arithmetic Operators: `+`, `-`, `*`, `/`, `+=`, `-=`, `*=`, `/=`
- Relational Operators: `==`, `!=`, `>`, `<`, `>=`, `<=`

- Logical Operators: `&&`, `||`, unary `!`
- Expression Operators: `[]`, `.`

2.1.4 Literals

Kazm literals represent a value of a primitive type. Kazm supports the following literals: int literal, double literal, bool literal, char literal, and string literal.

- int literals (64-bit): Sequence of digit characters 0 through 9 with an optional "`-`" in front to denote negative numbers. Ex: `int x = 100;` or `int y = -100;`
- double literals (64-bit): Sequence of digit characters 0 through 9 which contain a decimal point and digits following the decimal point. Ex: `double x = .9;` or `double y = 15.73;`
- bool literals (8-bit): denoted by the case sensitive constants: `true` and `false`
- char literals (8-bit): a character within single quotation marks ('). Ex: `char a = 'a'`

2.1.5 Delimiters

Comments: Single line comments will begin with // and multi-line comments begin with /* and end with */

White Space: All white space, except for white space separating tokens or within strings, will be ignored.

Delimiter Tokens: ( ) , ; { }

- Parentheses () will be used for mathematical expressions and delimit function arguments
- Curly Braces { } help with defining scope
- Semicolons (;) end statements as well as follow the right curly brace } after a class definition
- Commas (,) will separate function arguments and array elements

3 Types

3.1 Default Types

`bool, char, double, int`

`bool` : A primitive data type that takes 1 byte of memory that takes either a `true` or `false` value. Can exist in the program as a boolean literal or as an identifier typed as `bool`. Examples: `true`, `bool a = false;`

`char` : A primitive data type that takes 1 byte of memory. Can exist in the program

as a character literal or as an identifier typed as char. Examples: `'a'` or `char a = 'a'`

`double`: A primitive data type that takes 8 bytes of memory and follows standard double precision conventions. Can exist in the program as a double literal or as an identifier typed as double. Examples: `3.14`, `double x = 3.14`

`int`: A primitive data type that takes 8 bytes of memory. Can exist in the program as an integer literal or as an identifier typed as int. Examples: `5`, `int a = 5`

3.2 Data Structures

The following subsection walks through `tuples` and `arrays`. Classes will be explored later in the manual.

3.2.1 Tuple

A tuple in Kazm is of fixed length but can have mixed type. A tuple declaration for a tuple with `n` members is formatted as follows: `tuple (type_1, type_2, ..., type_n) variable_name;`
`tuple` indicates that the following variable is of type tuple. `(type_1, type_2, ..., type_n)` indicates the types of the tuple elements, and `n` indicates the total length of the tuple.

3.2.2 Array

An array in Kazm is of variable size. All elements in the array must be of the same type. An array declaration is formatted as follows:
`type[] variable_name[length];`
`type[]` indicates the type of the elements in the array. `variable_name` indicates the name of the array in the declaration. `[length]` with "length" being a required parameter indicates once again that `variable_name` denotes an array of length `length`.

The user is required to include the length of the array in the declaration unless an array is passed as a function parameter. A global function `len` which returns an int is used as follows to access the current length of any array:

```
int[] foo[5];
int a = len(foo); // a == 5
```

3.3 A Comment on Pointers and References

We do not have pointers or references.

4 Operators

4.1 Arithmetic Operators

Arithmetic Operators: `+`, `-`, `*`, `/`

Arithmetic operators are binary operators and are left to right associative. Addition and subtraction have lower precedence than multiplication and division.

4.2 Relational Operators

Relational Operators: `==`, `!=`, `>`, `<`, `>=`, `<=`

Relational operators are left to right associative but have lower precedence than arithmetic operators. `==` and `!=` have a lower precedence than the other relational operators (who have same precedence).

4.3 Logical Operators

Logical Operators: `&&`, `||`, unary `!`

Logical operators have lower precedence than relational operators and can be binary (`&&` and `||`) operators or unary `!`.

`expr && expr` will return true if and only if both expressions operands are true, else false.

`expr || expr` will return true if one operand is true, else false.

`!expr` will return true if the resulting value of the `expr` is false itself, else false.

4.4 Expression Operators

Expression Operators: `[]`, `.` These expression operators have highest precedence and are left associative.

The dot operator is used to access class member functions and variables.

The square bracket operator will behave differently on different types. For example, `[]` can be used to index on an array or on a string.

4.5 Assignment Operators

Assignment Operators: `=` , `+=` , `-=`, `*=`, `/=`
`=` simply assigns an expression on the right to a variable on the left.

`+=`, `-=`, `*=`, `/=` manipulate a variable on the left to the result of an expression determined by the assignment operator. += will add the expression on the right to the variable on the left. This could also be written as `var = var + expr`, or generally as `var = var operator expr` A similar form will follow for the other assignment operators, with the arithmetic operator on the right hand side substituted for the operator present in the assignment operator. These operators are right associative.

5 Declarations

All variables in Kazm must be declared before use. All declarations in Kazm associate a given identifier with a type. The type can be one of the default types or can be a user-defined type (i.e. class). While the type declaration is absolutely required, it is left optional for the user to initialize the identifier in the declaration. The value must be of the specified type. Each variable declaration must be preceded by the variable type. It is not possible to declare multiple variables in the same declaration.

Example declaration:

```
int a;
int b;
int c; // this is legal
char a, b, c; // this is not
```

Example declarations with initialization:

```
int a = 0;
char alphabet[26];
double pi = 3.14;
bool a = false;
```

6 Statements

An expression is one or more operands and zero or more operators.

Examples: `5`, `int x = 5`, `int y = 10 + 20/4`

Expressions can also be grouped with parentheses:

```
int x = (3+7) - (2*5) // x == 0
```

An expression becomes a statement when it is followed by a semicolon. In Kazm, the semicolon is a statement terminator, rather than a separator. This means that the following is a valid – however strongly discouraged – statement:

```
int a
= 0;
```

Curly braces { and } are used to group declarations and statements together into a compound statement, or block, such that they are syntactically equivalent to a single statement. The braces that surround the statements of a function are one obvious example; braces around multiple statements after an if, else if, else, while, or for are another. There is no semicolon after a right curly brace } that ends a block. On the other hand, there is a semicolon after a right curly brace that ends a class definition.

6.1 If, Else If, Else

The if, else if, else statement is used to express decisions. else if and else are optional. There may be multiple chained else if blocks contained between if and else. It is required to enclose the statements following if or else within curly braces. The syntax is as follows.

```
if (expression1)
{
    statement1
}
else if (expression2)
{
    statement2
}
else
{
    statement3
}
```

Other valid examples include the following:

Example 1.

```
if (expression1)
{
    statement1
}
else
{
    statement2 // there is no else if as it is optional
}
```

Example 2.

```
if (expression1)
{
    statement1
}
else if (expression2)
{
    statement2
}
else if (expression3)
{
    statement3
}
else if (expression4)
{
    statement4
}
else
{
    statement5 // multiple else if are chained together
}
```

Example 3.

```
if (expression1)
{
    statement1 // the else statement is missing as it is optional
}
```

6.2 While

A while expression allows a statement block to be executed multiple times while a condition holds true. The statement block must be enclosed within curly braces. The syntax is as follows:

```
while (expression)
{
    statement
}
```

Upon entering the while loop, the expression is evaluated. If it is non-zero, statement is executed and expression is reevaluated. This cycle continues until expression becomes zero, at which point execution resumes after statement.

6.3 For

The for statement allows a statement block to be executed a fixed number of times. The statement block must be enclosed within curly braces. The syntax is as follows:

```
for (expr1; expr2; expr3)
{
    statement
}
```

This is equivalent to the following:

```
expr1;
while (expr2)
{
    statement
    expr3;
}
```

None of expr1, expr2, expr3 may be omitted.

Example for loop:

```
int n = 5;
for (int i=0; i != n; i += 1)
{
    println(i);
}
```

6.4 Break

It is sometimes convenient or necessary to exit from a for loop before the initially designated number of executions. The break statement provides an early exit from the for and while loops. A break causes the innermost enclosing loop to be exited immediately.

Examples:

```
int n = 5;
for (int i=0; i != n; i += 1)
{
    if (i == 3)
    {
        break;
        // for loop iteration corresponding to i == 4 is not executed
    }
}
```

Kazm does not support continue.

7 Functions

The execution of a program in Kazm requires one main function which takes no arguments and returns nothing:

```
void main()
{
    statements
}
```

Example functions within an example program are as follows:

```
// add.kazm
int sum(int[] list, int list_size) {
  int result = 0;
  for (i = 0, i < list_size, i++) {
      result += list[i];
  }
  return result;
}

void main() {
    int[] list[5] = [1, 6, 12, 17, 25];
    int sum = sum(list, 5);
}
```

Arguments are always passed by value.

8 Scope of Variables

The scope of a variable is the part of the program in which the name can be used. All variable names and functions in Kazm are assumed to be accessible globally.

9 Classes

A class is a user-defined structured object consisting of zero or more member variables and zero or more *methods* (functions).

```
class-specifier:
    class identifier { class-body }
```

Class identifiers customarily start with an uppercase character. The class body consists of a list of member variables and methods.

```
class-body:
    variable-declaration-list;
    function-declaration-list;
```

The following is an example of a valid class declaration and usage

```
class Book {
    int isbn;
    String title;
    String author;

    Book(int isbn, String title, String author) {
        me.isbn = isbn;
        me.title = title;
        me.author = author;
    }

    String as_text() {
        return me.title.concat(" by ").concat(me.author);
    }
}

Book my_book = Book(1234, "The Kazm LRM", "The Kazm team");
println("ISBN: ", my_book.isbn);
println(my_book.as_text());
```

Class members are referred to using the dot syntax, and they are accessible from anywhere (there are no private or protected members). In the above example, `String` is a class and `me.title.concat(...)` invokes the `concat` method of `String`. Similarly `my_book.isbn` refers to the `isbn` member variable of the `Book` class `my_book`.

Member variables are initialized in the same way as any other variable.

Methods are similar to other functions, except that they have a symbol `me` available that makes it possible to refer to member variables or methods. Incomplete types and self-referential types are not supported.

9.1 String Class – Part of the Standard Library

```
class String {
    char[] str;

    /* Constructor */
    String(char[] str) {
        me.str = str;
    }

    /* member function */
    String concat(String you) {
        str.resize(me.str.len + you.str.len);
        // resize takes as argument the new length of the array

        //"hello" " world" 5, 6 ==> 12

        int start = me.str.len;
        for (int i = 0; i< you.str.len; i+=1){
            str[start] = you[i];
            start += 1;
        }
        //[h, e, l, l, o, , w, o, r, l, d]
    }
};
```

10 Example Programs

Example 1.

```
{
    int gcd(int a, int b) {
        while (a != b) {
            if (a > b)
            {
                a -= b;
            }
            else
            {
                b -= a;
            }
        }
        return a;
    } // notice how while, if, and else all require curly braces
    // even if there is only one statement within the braces
}
```

Example 2.

```
{
    int fibonacci(int n) {
    // returns the nth Fibonacci number
    // with the first and second Fibonacci numbers defined as 1 and 1
        if (n == 1)
        {
            return 1;
        }
        else if (n == 2)
        {
            return 1;
        }
        else
        {
            int a = 1;
            int b = 1;
            for (int i=3; i <= n; i += 1)
            {
                int store = b;
                b = a + b;
                a = store;
            }
            return b;
        }
    }

}
```