Contents

Chapter 1

Lexical Conventions

1.1 Comments

In Jtrix, `/*` opens a multi-line comment and `*/` closes the comment. `//` opens a single line comment that is ended by a newline. Comments are ignored and they do not nest. Comments cannot be contained in literals.

```
// This is a single line comment.

/* This is a multi-line comment,
   so I can end it down here. */

int x = 13; //Comments can follow lines of code
```

1.2 White Space

```
//white space is defined as follows:

' '      //space character
'\t'    //horizontal tab character
'\n' //newline character
```

White space is ignored, but can be utilized in cases of tokenization.

```
//white space is used to differentiate between the single token >=
    and the two tokens > and =

a >= b;  //this is a valid statement which evaluates whether a is
    greater than or equal to b, whereas
a > = b; //is not a valid statement
```

1.3 Tokens

Jtrix uses five different tokens:

Identifiers
Keywords
Literals
Separators
Operator

1.3.1 Identifiers

Jtrix Identifiers are any n-length set of characters beginning with an English letter that is either uppercase (`A - Z`) or lowercase (`a - z`) that do not form a Jtrix Keyword. Identifiers may not start with integers (`0 - 9`) or with underscores (_) but can include them in any following position.

```
//valid identifiers
thing
thing1
thingTwo
ThingTwo
Thing_Two

/* Note: identifiers are case sensitive, so thingTwo and ThingTwo
    are treated distinctly different */
```

1.3.2 Keywords

ADD CLASS ? The following ASCII character sequences are reserved words that may only be used as Jtrix Keywords:

break	boolean	char	col	continue	else	float	for	foreach
if	int	new	matrix	return	row	string	void	while

1.3.3 Operators (LIST ALL THE OPS BY PRECEDENCE)

Jtrix has both binary and unary operators. Operators are used in combination with variables or literals to create expressions. Binary operators are represented and evaluated in infix notation and unary operators are evaluated in prefix notation. Jtrix uses the following types of operators:

Arithmetic Operators
Equivalence Operators
Logical Operators

Arithmetic Operators

```
//addition applies to int, float, matrix
   +
//subtraction applies to int, float, matrix
   -
//multiplication applies to int, float, matrix
   *
//division applies to int, float
   /
//modulo applies to int
   %
```

Equivalence Operators

```
//equality applies to int, boolean, float, matrix
   ==
//inequality applies to int, boolean, float, matrix
   !=
//greater than applies to int, float
   >
```

```
//greater than or equal to applies to int, float
   >=
//less than applies to int, float
   <
//less than or equal to applies to int, float
   <=
```

Logical Operators

```
//all logic operators apply only to boolean

//and
   &&
//or
   ||
//not
   !
```

Unary Operators

The operator not (`!`) is used strictly as a unary operator. The minus sign (`-`) can be used as both a unary and binary operator.

```
boolean a = true;
return !a; // using not as a unary operator

int pos1 = 1;
int neg1 = -1; // using the minus sign as a unary operator
return (neg1 - pos1); // using the minus as a binary operator
```

1.3.4 Literals

Jtrix uses the following literals:

Integer
Float
Boolean
Character

String

Integer Literals

Integer literals represent whole number decimal values using characters `0 - 9` in a sequence

```
//examples of integer literals:
0
1
354234
```

Float Literals

Float literals represent whole or non-whole number values. Floats must have a decimal point followed by at least one number `1 - 9`. Floats may have `0 - 9` preceding the decimal. Floats may be exponents in which case the sequence is concluded with `+` or `-` and `e` or `E` and `0 - 9`.

```
//examples of float literals:
.3
0.3
3.6
3.6e+3
3.6E-3
```

Boolean Literals

Boolean literals are `true` or `false`. The type `boolean` has two values that are represented by the boolean literals.

Character Literals

```
//examples of character literals:
```

String Literals

String literals consist of zero or more character sequences enclosed within double quotes, ". A backslash is used as an escape to represent special characters.

```
//examples of String literals:
"hello world"      // an 11 character string
""                 // an empty string

//special characters:
"\n"             // a string containing the newline character
"\""             // a string containing the double quote character
"\'"             // a string containing the single quote character
```

1.3.5 Separators

Jtrix has the following separators:

```
( )   { }   [ ]   ;   ,   .
```

Chapter 2

Data Types

2.1 Primitive Data Types

Jtrix has 4 primitive types that represent a fixed length series of bytes:
bool (one byte)
char (one byte)
int (four bytes)
float (eight bytes)

2.2 Standard Library Types

Jtrix has arrays, strings and matrices as built-in types that can be referred to as *objects*. These built-in types are immutable and have methods that offer the user a large selection of operations. Arrays are not dynamic, meaning that they may only contain elements of the same type.

```
String a = "foo";       // this is a string

char[] charArr = ['a', 'b', 'c'] ;    // this is an array of chars
int[] numArr = [1, 2, 3] ;              // this is an array of ints

matrix mat = [ 1, 2 ; 4, 5]; // this is a 2x2 matrix
```

2.3 Mutability

All data types in Jtrix are immutable. The user can assign identifiers to each type but cannot overwrite the object that the identifier is assigned to. To effectively change a variable, the user reassigns the identifier to reference a different piece of data.

```
/* this assigns variable x to integer literal 3 and then reassigns
   it to integer literal 4 */
x = 3;
x = 4;
/* x is now referencing a different value, but the integer literal
   3 did not change */

// string example:
strFoo = "Foo ";
strBar = "Bar";
strFoo = strFoo.concat(strBar);
/* strFoo was reassigned to the concatenation of the literal
   values "Foo " and "Bar" to result in strFoo to reference "Foo
   Bar" */
```

2.4 MEMORY ALLOCATION PROCESSES?

Chapter 3

Meaning of Identifiers NOT SURE WHAT THIS MEANS

Chapter 4

Declarations SHOULD THIS GO UNDER STATEMENTS?

Jtrix uses *declarations* to introduce an item into the code. In Jtrix, all declarations are explicit with the aim of increasing the readability of the code to help solve runtime errors. Jtrix does not implicitly type declarations which reduces compiling errors that many implicitly typed languages experience.

4.1 Functions

4.2 Variables

4.3 Arrays

4.4 Matrices

Chapter 5

Statements SHOULD THIS BE CHANGED TO CONTROL FLOW?

5.1 Control Flow

Jtrix offers a number of `control flow` statements that are useful for iterating over data structures, executing expressions multiple times, and executing specific operations based on user defined conditions.

5.1.1 While Statements HOW ARE WE GOING TO DEFINE PRINTING(PRINTING USED IN WHILE LOOP)

`while` statements are used to iterate through a set of code based on a user defined boolean expression. The statement evaluates the expression and, if the expression is true, the code within the brackets will be executed. After executing the code within the brackets, the loop will restart by reevaluating the expression and repeating this process as long as the statement is true.

```
/* this code prints y and then concatenates y with foo until y is
   equal to x */
string x = "foo foo foo ";
string y = "";
string foo = "foo ";
```

```
while (x != y){
   print(y);
   y = y.concat(foo);
}
// first loop prints blank
// second loop prints "foo "
// third loop prints "foo foo "
// fourth loop prints "foo foo foo " and terminates the loop
```

5.1.2 For Statements WHAT IS OUR MEMORY/SCOPE?

for statements are similar to while statements, however they require the variable used in the boolean statement to be initialized within the loop. Therefore the scope of the variable is only within the for loop which can be useful for writing multiple loops. for loops define 3 arguments. The first initializes a variable, the second is the conditional statement, and the third is an expression to change the value of the variable.

```
for( int i = 0; i < 5 ; i = i + 1 ){
   //some code
   }
```

5.1.3 If-Else Statements