The XIRTAM Language Manual

Annie Wang (aw3168) - manager
Andrew Peter Yevsey Gorovoy (apg2165) - manager/architect (environment selection)
Shida Jing (sj2670) - tester
Bailey Nozomu Hwa (bnh2128) - language
Lior Attias (lra2135) - architect (compiler design)

1. Acknowledgement

We would like to acknowledge that this document is heavily borrowed from the C language manual located here:

https://www.bell-labs.com/usr/dmr/www/cman.pdf

A large portion of this document is based off of snippets from the C language manual with minimal changes.

2. Introduction

The XIRTAM language is an object-oriented C-style language for manipulating matrices. Besides matrix declarations and operations, XIRTAM behaves analogously to C. The XIRTAM language is a statically typed and compiled language focused on the manipulation of matrix (XIRTAM) objects. The purpose of XIRTAM is to provide functionality akin to Python's numpy matrix manipulation module in addition to flexible matrix operations not possible in traditional languages.

3. Lexical Conventions

In XIRTAM, newlines, spaces, tabs, and comments as described below are ignored. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

XIRTAM contains the following tokens:

3.1. Comments

The characters `/*` introduce a comment, which terminates with the characters `*/`.

3.2. Identifiers

A letter is defined to be one of the 26 English letters, either uppercase or lowercase. A digit is defined to be one of 0 through 9. An identifier is a sequence of letters and digits, with which the first character must be a letter. The underscore “_” counts as a letter. Upper and lower case letters are considered different.

Identifiers are used as corollary labels for variables, functions, and Xirtam matrix objects. The above definition of identifiers provides restrictions on valid identifiers.

3.3. Punctuators

Punctuators in XIRTAM have syntactic and semantic meaning to the compiler but do not, of themselves, specify an operation that yields a value. Any of the following in it of themselves are considered as punctuators.

```
! % ^ & * ( ) - + = { } | ~ [ ] \ ; ' : " < > ? , . / #
```

3.4. Keywords

The following keywords are reserved and cannot be used otherwise.

```
numeric
string
bool
Xirtam
if
else
else if
for
void
return
true
false
new
del
NULL
```

4. Basic Types

4.1. Strings

A string is a sequence of characters surrounded by double quotation marks `"`, declared by the keyword `string`. Double quotation marks are not allowed to appear in a string.

4.2. Numeric

A `numeric` is a number, encompassing integers and floating point numbers. It could be negative, 0, or positive. An operation on two numerics always returns another numeric. An operation between a numeric and another type has different return types, depending on the situation. All non-zero numerics are interpreted as the keyword `true.` 0 is interpreted as the keyword `false`.

4.3. Boolean

A `bool` is either the keyword `true` or the keyword `false`.

4.4. Xirtam

A `Xirtam` is an array of arrays of type `numeric`. (see Section 7)

5. Xirtam Expressions

5.1 General Overview

1. For non-Xirtam object types (see section 7), common unary operators such as `=, ==, !=, +, -, *, /, ^, <, >, =<, >=, &&,||` work the same way as they do in C. More specifically:
    a. The assignment operator `=` must have an identifier on the left, and anything on the right. If the right side is a value, it assigns that value to the identifier. If the right side is an expression, it evaluates the expression and assigns the value to the identifier. The assignment is kept within scope.
    b. `==, !=` evaluates both sides and returns a bool, analogous to C. `==` only evaluates to 1 or `true` if both the left and right side evaluate to the same logical conclusion. `!=` only evaluates to 1 or `true` if left and right side evaluate to opposite values.
    c. `+, -, *, /` can be used on numerics to perform arithmetic operations.
    d. `^` can be used to perform string concatenation
    e. `<, >, =<, >=` can only be used on numerics, analogous to C.
    f. `&&, ||` are logical operators used on booleans. It cannot be used on any other type.`&&` only evaluates to 1 or `true` if both the left and right side evaluate to 1 or `true`. `||` only evaluates to 1 or `true` if either the left and right side evaluate to 1 or `true`.

5.1 Operator Precedence
Operator precedence will take place in the following order (with the exception of assignment, Xirtam objects which have their own methods for dealing with certain operations as outlined in Section 7). Within the same level of precedency, they are evaluated from left to right, and top to down. Parenthesis will override the precedence order, with expressions inside the parenthesis evaluated first:

1. Function calls
2. Multiplication, division
3. Addition, Subtraction
4. Equal not-equal, and other boolean operations
5. Assignment

6. Declarations

6.1 Single Variables

Declarations are used to give identifiers certain values. To declare a variable, you must specify the type, followed by the identifier, followed by `=`, followed by the value you wish to assign. The following is the required basic pattern for declaring single variables ({} aren't part of the actual code):

{type} {identifier} = {value}

If a value is of type `string` then the value must be defined in between two double quotes as stated previously.

{string} {identifier} = "{value}"

If a value is of type `numeric` then the value must be either a decimal or integer value

{numeric} {identifier} = {value}

If a value is of type `bool` then the value must be either `true` or `false`

{boolean} {identifier} = {true or false }

For example,

```
string s = "Hello";
numeric n = 14;
bool b = true;
```

declares a string type identifier s with the value “Hello”, a numeric type identifier n with value 14, and a bool type identifier b set to true, respectively.

6.2 Functions

XIRTAM supports function declarations in a standard C style. The function declaration must adhere to the following pattern:

```
<return_type> <function_name>(parameter list) {
      body of the function
}
```

<return_type> - is the type returned by the function
<function_name> - is the identifier used to reference the function declaration
<parameter list> - is the list of arguments passed to the function. Arguments must take the following form {type} {identifier}

For example, to write a function that adds two integers in XIRTAM it looks like this:

```
numeric foo (numeric a, numeric b){
      return (a+b);
}
```

6.3 Function Calls

In order to call a function, the function name must be called followed by parentheses. If the function declaration has arguments, then the necessary arguments identifiers must be passed into the function reference. The following pattern must be followed:

```
function_name() /*if no arguments*/
```

```
function_name(parameter list) /*if function declaration takes arguments*/
```

For example,

```
foo(10, 11);
```

A function that returns nothing has a declared type `void`.

7. Xirtam Matrix Objects

7.1 Overview

To declare a matrix object, you must use the `Xirtam` keyword, followed in order by the identifier, assignment operator `=`, by the `new matrix` expression, and finally by the matrix. Matrices can be expressed using an array of arrays format. The following is the required pattern for a matric array:

```
{ {<num>,<num>,<num>}, {<num>,<num>,<num>}, {<num>,<num>,<num>}... }
```

The Xirtam matrix object can only take numeric type values. There is no multi-type matrix object.The Xirtam matrix object is not dynamic. Once an object has been defined with a given number of rows and columns, it can no longer be changed.
The following is then the required pattern for declaring a Xirtam matrix object:

```
Xirtam <identifier> = new matrix({ {<num>,<num>,<num>},
{<num>,<num>,<num>}, {<num>,<num>,<num>}... });
```

There are also multiple other ways to declare a Xirtam matrix object

```
Xirtam matrix = new matrix({ {1, 2, 3}, {4, 5, 6}, {7, 8, 9} });
Xirtam matrix = new matrix(2, 3, 0); /* a 2 row, 3 column matrix containing
the int value 0. */
Xirtam matrix = new matrix(2, 3); /* a 2 row, 3 column matrix containing
the int value 0 by default */
Xirtam matrix = new matrix("identity", size); /* the identity matrix, size
is an integer, specifying the number of rows and columns. */
```

7.2 Xirtam Matrix Operations

XIRTAM matrices have built-in operations for the basic operations such as elementwise, addition, multiplication. For equality, we return true for two matrices if (they are of the same size and they are equal elementwise). XIRTAM also has functions specific to matrix-operations beyond what is mentioned above:

7.2.1 Basic operations

```
Xirtam matrix_addition = XIRTAM.add(matrix1, matrix2);
Xirtam matrix_multipication = XIRTAM.multiply(matrix1, matrix2);
Xirtam matrix_multipication = XIRTAM.scalar_multiply(matrix1, <numeric>);
```

7.2.2 Equality

```
bool matrix_equal = XIRTAM.equal(matrix1, matrix2);
```

7.2.3 Advanced Matrix Functions

```
Xirtam matrix_tensor = XIRTAM.tensor(matrix1, matrix2);
Xirtam matrix_transpose = XIRTAM.transpose(matrix1);
XIRTAM.determinant(matrix1);
```

7.2.4 Matrix Manipulations Functions

```
Xirtam new_matrix = XIRTAM.remove(0, matrix); /* replaces all 0 values in
the matrix with NULL, for example for JPEG compression */
Xirtam new_matrix = XIRTAM.remove(0:10, matrix); /* replaces all Xirtam
new_matrix = values 0 - 10 in the matrix with NULL */
XIRTAM.remove([0, 10, 11, 12], matrix); /* replaces specific values the
matrix with NULL */
Xirtam new_matrix = XIRTAM.replace(0, 10, matrix); /* replaces all 0 values
with 10 */
Xirtam new_matrix = XIRTAM.replace(0:10, 100, matrix); /* replace all 0
through 10 (inclusive) values in the matrix with the value 100 */
```

```
Example matrix m
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

```
numeric rows = XIRTAM.rows(m);  /* returns 3 */
numeric cols = XIRTAM.cols(m);  /* returns 3 */
Xirtam col_1 = XIRTAM.cols(m, 1);    /* returns [1, 4, 7] */
Xirtam cols_1_2 = XIRTAM.cols(m, 1:2); /* returns columns 1 through 2
inclusive */
Xirtam rows_1 = XIRTAM.rows(m, 1);   /* returns [1 2 3] */
Xirtam rows_2:3 = XIRTAM.rows(m, 2:3); /* returns a 2D array rows 2 through
3 inclusive */
Xirtam sub_matrix = XIRTAM.dims(m, 1:2, 2:3); /* returns a 2D array
representing {{2, 3}, {5, 6}} (the intersection of rows 1 - 2 and cols 2 -
3) */
```

Caveats

1. Xirtam does not support function overloading

8. Statements

8.1 Print Statement

```
print (string or variable);
```

8.2 Conditional Statements

```
if (condition1) {
   /* statement(s) */
}
else if(condition2) {
   /* statement(s) */
}
else if (condition3) {
   /* statement(s) */
}
else {
   /* statement(s) */
}
```

Statements in the `if` block is evaluated only if the condition evaluates to true. Otherwise, the program checks to see if an `else ` or `else if` block exists. If not, proceed to the next statement. If yes, check if conditions in the `else if` block(s) are ture. If so, the corresponding statement will be evaluated. If neither the `if` and `else if` blocks evaluates to true, then the statements in the `else` block will be evaluated.

8.3 For Statements

```
for ( init; condition; increment ) {
   statement(s);
}
```

Init is executed first and only once. It allows you to declare and initialize any loop control variables. Then the condition is evaluated. If it is true, the statements will be evaluated and afterwards, the increment will happen. This loop is repeated until the condition evaluates to `false`.

9. Sample Code

9.1 Fibonacci Sequence

Generating the nth number of a general Fibonacci Sequence. Parameters: a_1, a_0 are the starting values of the sequence.

```
numeric nthFibo(numeric  num n, numeric num a_1, numeric
num a_0){
        Xirtam m = new matrix({ {1, 1}, {1, 0} });
        Xirtam exp_m = new matrix("identity", 2);
        for (numeric i = 0; i< n; i++){
                exp_m = XIRTAM.multiply(new_m, m);
        }
        Xirtam init = new matrix({ {a_1}, {a_0} });
        Xirtam result = XIRTAM.multiply(exp_m, init);
        return result[1];
}
```

9.2 Markov Chain Stable State

Given a transition matrix of size n by n, and an initial state vector of size n by 1, we will find the stable state, if there is one, of the transition process. We do this by keeping applying state transition to the state vector, until there's minimal changes in the states.

```
numeric find_max_markov(Xirtam m) {
        max = -1;
        for (numeric i = 0; i < XIRTAM.rows(m); i = i + 1) {
                Xirtam row = XIRTAM.rows(m, i);
                for (numeric j = 0; j < XIRTAM.cols(row); j = j +
1) {
                        if (XIRTAM.cols(row, j) > max) {
                                max = XIRTAM.cols(row, j);
                        }
                }
        }
        return max
}
Xirtam find_stable_markov(Xirtam t, Xirtam v, numeric
thresh){
        Xirtam cur = v;
        numeric dif = find_max_markov(Xirtam.minus(cur, v))
        while dif > thresh {
                cur = XIRTAM.multiply(t, cur);
        }
        return cur;
}
```

9.3 Total Positivity

Check for total positivity. A minor of a matrix A is a square matrix which is formed by selecting a subset of the rows of A, then selecting a subset of the columns of A, then taking the entries from the intersection. A matrix is totally positive if all of its minors have positive determinant. If we have a 2 by n matrix A and we want to check for total positivity, there are (n choose 2) many non-trivial minors to check. However, a theorem states that we only need to check 2n-3 many minors to conclude total positivity.

```
bool 2bynTotalPositivity(Xirtam m){
      bool isTotallyPositive = true;

      for (numeric i = 0; i < 2 * XIRTAM.cols(m) - 3; i++){
            Xirtam minor = new Xirtam({ XIRTAM.cols(m, 0),
XIRTAM.cols(m, i) });
            isTotallyPositive = isTotallyPositive &&
(XIRTAM.determinant(minor) > 0);
      }
      return isTotallyPositive;
}
```

Total positivity is important in **algebraic combinatorics** but historically it stems from real analysis, because totally positive matrices have real distinct positive eigenvalues, and therefore they are diagonalizable.