Twister: Language Reference Manual

Manager: Anand Sundaram (as5209)
Language Guru: Arushi Gupta (ag3309)
System Architect: Annalise Mariottini (aim2120)
Tester: Chuan Tian (ct2698)

February 23, 2017

Contents

1 Introduction

Twister is a language designed for image manipulation with standard functions that handle conversion between commonly-used image file formats and matrix representations of the images. It is an imperative programming language with first-class functions and a sparse set of built-in types, including a matrix type.

Twister is intended to be used for image manipulation with efficient implementations of linear algebra operations such as convolution. Twister programs may be used to read in images, sharpen, de-noise, and write out images. Twister can be used by anyone from selfie lovers to image designers. It implements a simple to use pipe operation that allows users to chain together alterations made to images. It also provides built-in support for element-wise matrix operations.

2 Parts of the Language

2.1 Comments

- Single-line — any characters following the string // are ignored until a newline is encountered
- Multi-line — any characters after the string /* are ignored until the string */ is encountered

2.2 Identifiers

Variable names must match the following regex:

```
['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
```

2.3 Keywords

The following words are keywords and cannot be used as variable names: int, float, bool, char, fun, List, String, Tup, Struct, Matrix, if, else, return, and, or, not, for, map, reduce, fread, fwrite, print, range, toint, tofloat, True, False.

2.4 Initializing and Declarations

Initialization and declaration must occur in the same statement. Variables are declared by specifying the variable type and the identifier separated by a space and initialized by an assignment operator that takes the right hand side of the assignment operator and stores it in the specified operator.

```
<type> <id> = <expr>
```

Where `<type>` is either a primitive or complex type, `<id>` is a valid identifier, and `<expr>` is an expression that returns type `<type>`.

2.5 Primitive Types

2.5.1 Scalar primitive types

Type	Description	Initialization Example
int	an integer	`int a = 5`
float	a floating point number	`float a = 5.2`

Scalar constants: Literal strings of digits are interpreted as integers; strings of digits containing one interior "." character are interpreted as floats. Both of these are scalar constants and can be used to assign specific values to scalar variables when declaring them, as shown in the "Initialization Example" column of the table above.

2.5.2 Other primitive types

Type	Description	Initialization Example
bool	a boolean	`bool x = True`
char	a character	`char x = 'a'`

Constants: Any single character enclosed in single quotes is interpreted as a character constant. The keywords True and False are interpreted as Boolean constants. These constants can be used to assign specific values when declaring the corresponding type of variable; examples are shown in the table above.

2.6 Complex Types

2.6.1 Summary

Type	Description	Attributes	Initialization Example
List	a mutable array	$[i]$, length	`List l = {'a', 'b','c'}`
String	a List of chars	$[i]$, length	`String s = 'hello'`
Tup	an n-tuple of value	$[i]$, size	`Tup t = (0, 'a', 5.2)`
Struct	a collection of specified types	—	`Struct pt = {int x; int y;}`
Matrix	a series of scalar arrays	$[i][j]$, dims, rows, cols	`Matrix m = [0,0,0;0,0,0]`
fun	a function (input to output)	—	`fun x = (i:  int) -> int {return i;}`

2.6.2 Lists

Initialization: Lists are initialized using List literals, which consist of two curly braces with elements separated by commas. An empty list may be initialized with a pair of braces with no elements.

```
List empty = {};
List stuff = {1,2,3};
```

Element Access/Assignment:

- $[i]$
  - As expression: accesses position i of a List
  - As assignment: assigns to position i
- $[i : j]$
  - As expression: returns a List from position i (inclusive) to position j (exclusive) of the List
  - As assignment: assigns a List of length from position i to j to the positions $i - j$ (inclusive and exclusive respectfully)
- $[i :]$
  - As expression: returns a List from the position i (inclusive) to the end of the List
  - As assignment: assigns a List of length from i to the end of the assignment List to the positions i (inclusive) to the end of the assignment List
- $[: i]$
  - As expression: returns a List from position 0 to position i (exclusive)
  - As assignment: assigns a List of length from 0 to i to the positions 0 through position i (exclusive)

```
List l = {'a', 'b', 'c', 'd'};
List m = l[1:3]; // m = {'b', 'c'}
l[2:] = m; // L = {'a', 'b', 'b', 'c'}
```

Other Attributes:

- Length - the length of a List may be accessed with the ".length" selector

2.6.3 Strings

Description: Strings are special type of List that contain only characters.
Initialization: Strings may be initialized as Lists, but may also be initialized with a String literal. A String literal consists of a pair of doubles quotes surrounding zero or more chars.

```
String s = "abc";
char x = s[0];
```

2.6.4 Tuples

Description: An n-Tuple (aka Tuple) is a collection of n-many identically typed values. The value n of a Tuple cannot change after initialization.
Initialization:: A Tuple is initialized with a pair of parentheses that surround n-many expressions that return the same type.

```
Tup t = (1, 2, 2 + 4);
```

Attributes:

- $[i]$
  - As expression: returns the element at position i of the Tuple
  - As assignment: assigns an element of the same type to position i of the Tuple
- Size - the selector ".size" returns the value n of an n-Tuple

2.6.5 Structs

Description: Structs may contain variables of different types.
Initialization: Structs are initialized with a pair of brackets that contain a series of expressions separated by semicolons.

```
int x = 3;
Struct numbers = { x ; float y = 5.0 };
```

Element Access/Assignment: Elements of a struct may be accessed and assigned using the dot operator and the id of the element.

```
int z = numbers.x;
numbers.x = 3;
```

2.6.6 Matrices

Description: Matrices are two dimensional collections of scalars or tuples of scalar types.
Literal Initialization: You may use a Matrix literal to initialize a Matrix object. A Matrix literal consists of a pair of square brackets with commas separating the elements of of a row, and semicolons designating the beginning of a new row.

```
Matrix m = [1,2,3;4,5,6;7,8,9];
```

Functional Initialization: One may also initialize with a tuple, consisting of a tuple to represent dimensions and an iterative function to initialize individual values based on position. The last two arguments of this initializer function will represent the (x, y) position of the Matrix currently being initialized. This function must return either a scalar or a tuple of scalars. If the user does not have a initializer function specified, the matrix will initialize to zeros.

```
fun fill = (apply: int, x: int, y: int) -> int {
    if (x %% 2 == 0 and y %% 2 == 0) {
        return apply;
    } else {
        return 0;
    }
};
Matrix m = Matrix((2,2), fill(2)); // [0,0;0,2]
```

Attributes: Each of the following selectors can be used by using the identifier of a Matrix object followed by the selector name.

- Element access
  - [i][j] — returns the element in row i and column j.
  - [i][:] — returns a single-row Matrix with all the values in row i
  - [:][j] — returns a single-column Matrix with all the values in column j
- Element assignment
  - [i][j] — assigns to the element in row i and column j.
  - [i][:] — assigns a single-row Matrix of identical width to the single-row Matrix in row i
  - [:][j] — assigns a single-column Matrix of identical height to the single-column Matrix in column j
- Row access — the ".rows" selector returns a List of of the row vectors (single-row Matrices)
- Column access — the ".columns" selector returns a List of the column vectors (single-column Matrices)
- Dimensions — the ".dims" selector returns a tuple containing the row and column lengths of the Matrix
- Row/Column Lengths — the ".num_rows" and ".num_cols" selectors are syntactic sugar for "dims[0]" and "dims[1]", respectively

Demonstration of Attribute Usage:

```
Matrix a = Matrix((2,3)); // a is a 2 by 3 matrix of all zeros
Tup d = a.dims; // (2,3)
int r = a.num_rows; // 2
int c = a.num_cols; // 3
List rows = a.rows; // {[0, 0, 0], [0, 0, 0]}
List cols = a.cols; // {[0; 0], [0; 0], [0; 0]}
a[0][0] = 3
a[0][:] = [1,2,3];
a[:][0] = [1;2];
```

2.6.7 Functions

Description: All functions are first-class objects and may be treated in the same manner as other objects.
Initialization: A function is initialized in the following manner:

```
fun x = (<id>: <type-1) -> <type-r> {
<BODY>
    return <expr>;
};
```

Where `<id>` is a valid identifier, `<type-*>` is a primitive or object type, `<BODY>` is a series of statements, and `<expr>` is an expression that returns type `<type-r>`. To pass in addition arguments, you may add in other `<id>: <type>` pairs, sequenced by commas.
Function Calls: A function is called by its id, followed by a pair of parentheses with 0 or more expressions (to be passed as arguments) separated by commas. If fewer arguments are supplied than the function accepts, then the function returns a function bound to the supplied arguments that accepts the remaining arguments. E.g., if a function takes 3 arguments and is supplied with 2, then it returns a function that takes in the 1 remaining argument.

```
\en

\begin{verbatim}
fun sum = (x: int, y: int) -> int {
    return x + y;
};

fun sum_matrix = (m: Matrix<int>) -> Matrix<int> {
row_sums = {};
    for (row in m.rows) {
     row_sums = row_sums + reduce(sum, row);
}
    return reduce(sum, row_sums);
};

Matrix a = [1,2;3,4];

sum_matrix(a) // returns 10
```

2.6.8 Object Typing

The List and Matrix objects have associated value types that are defined either upon initialization or upon being declared with a specific type. To manually specify this type, immediately after the object type use triangle brackets to contain the desired type.

```
Matrix M = [1,1;1,1]; // automatically typed to int
List<float> N; // manually typed to float
```

2.7 Operators

2.7.1 Arithmetic Operators

Operator	Description
$-$	the subtraction operator
$+$	the addition operator
$*$	the multiplication operator
/	the division operator
%%	the modulo operator
.operator	element-wise matrix application

The arithmetic operators may be used between two scalar values or one scalar and one matrix to perform scalar multiplication as expected. The $*$ operator may be used between two matrices to perform matrix multiplication. All arithmetic operators may occur between two matrices if preceded with a '.' in order to perform element-wise operations of one matrix upon another matrix of identical dimensions. A matrix and a scalar may also be multiplied using the $*$ operator.

```
Matrix a = [2,2;2,2];
Matrix b = [1,0;0,1];
int x = 3;

b .+ x; // returns [4,3;3,4]
a .* b; // returns [2,0;0,2]
a * b; // returns [2,2;2,2]
```

2.7.2 Functional Operators

Operator	Description	Example
|	function composition	`transpose(A) | convolution(this, kernel, 1)`

The functional operator pipes the return value of the left-hand-side function into the 'this' keyword, to be used in the arguments of the right-hand-side function.

2.7.3 Logical Operators

Operator	Description	Examples
==	equality comparison operator	`1 == 1 // True`
>	greater than operator	`1 > 2 // False`
<	less than operator	`1 < 2 // True`
and	takes two bools and returns their AND	`(1 > 2) and (2 == 2) // False`
or	takes two bools and returns their OR	`(1 > 2) or (2 == 1) // True`
not	returns the negation of a boolean	`not (1 == 1) // False`

2.7.4 Bitwise Operators

Operator	Description	Example
&&	bitwise and	`1 && 2 // 0`
||	bitwise or	`1 || 3 // 3`
*|	bitwise xor	`1 *| 3 // 2`
<<	left shift	`1 « 1 // 2`
>>	right shift	`2 » 1 // 1`
!	bitwise not	`!0 // -1`

2.8 Flow Control

2.8.1 For Loops

A 'for' loop is defined as an iteration over a List or Matrix with a named variable to represent the current element. If looped over a List, the elements iterate in indexed order. If looped over a Matrix, the elements iterate over rows first, then columns, in indexed order.

```
for(<var> in <values>) {
    <BODY>
}
```

Where `<var>` is any type, `<values>` is a List of values of that type, and `<BODY>` contains any sequence of statements that are valid Twister code.

```
for(elem in range(0,3)) {
    print(elem);
} // output:
// 0
// 1
// 2
```

2.8.2 If-Else Statements

Users can check booleans and conditionals with if statements, with an optional else block to run if the condition is not satisfied.

```
int a = 5;
if (a == 5) {
    print('a was 5');
} else {
    print('a was not 5');
} // output: a was 5
```

The exact syntax of the if block is

```
if (<CONDITION>) {
<BODY-1>
} else {
<BODY-2>
}
```

Where `<CONDITION>` is any expression that evaluates to a Boolean value, and `<BODY-1>` and `<BODY-2>` contain any sequence of statements that are valid Twister code.

2.8.3 Iterating over matrices

There is a simplified syntax available to iterate over matrix rows and columns. The following code will loop over the rows in M, and for each row, print the values in that row.

```
for(i in M.rows)
{
    for(row_elem in i)
    {
        print(row_elem)
    }
}
```

A similar syntax is available for columns:

```
for(i in M.cols)
{
    for(col_elem in i)
    {
        print(col_elem)
    }
}
```

2.9 Built-in functions

Function	Description
map	applies an element-wise function to a Matrix
reduce	folds a function across a List
fread	takes a file name as a string and outputs a Matrix
fwrite	takes a Matrix and a file name and write the Matrix to the file
print	takes a String and a file and writes the string to the file
range	takes two arguments a and b, returns a List of ints = $\{a, a+1, ..., b-1\}$
tofloat	converts ints to floats
toint	converts floats to ints

3 Useful Library Functions

scale(Matrix a, float fraction)	zooms in to the matrix a by the fraction specified
rotate(Matrix a, float degrees)	takes a matrix and rotates it by the number of degrees
stretch(Matrix a, float factor)	takes a matrix and stretches it by the factor

4 Example Program

```
fun square = (x: int) -> int {
    return x * x;
};

fun square_matrix = (m: Matrix<int>) -> Matrix<int> {
    return map(square, m);
};

fun transpose2D = (image: Matrix<int>) -> Matrix<int> {
    Tup newDims = (image.dims[1], image.dims[0]);
    fun swapVals = (x: int, y: int) -> int {
        return image[y][x];
    };
    return Matrix(newDims, swapVals);
};

fun fill = (apply: int, x: int, y: int) -> int {
    if (x %% 2 == 0 and y %% 2 == 0) {
        return apply;
    } else {
        return 0;
    }
};

Matrix a = fread('\Home\image.bmp');
Matrix b = Matrix((2,2), fill(2)); // [0,0;0,2]

a = transpose2D(a) | square_matrix(this) | this * b;

fwrite('\Home\image2.bmp', b);
```

5 Scope

Any variable defined between braces, {}, has a local scope within those braces and goes out of scope when the code between those braces finishes executing. Otherwise, a variable's scope extends from the time it is declared to the end of the program run. Variable declared in loops do not have scope outside of the loop that they are declared in. Variable names used for function arguments also have scope limited to the function they are defined in.

6 Compilation

Context Free Grammar Representation

program	→	decls EOF
decls	→	decls ’;’ decl | ϵ
decl	→	vardecl | fundefn | conditional | comment
conditional	→	if_condition "else" decls | if_condition
if_condition	→	"if" expr decls
comment	→	"/*" <anything but "*/"> "*/" | // <anything but a <newline>
vardecl	→	TYPE IDENT ’=’ expr
fundefn	→	"fun" IDENT ’=’ (args) -> TYPE fundecls
fundecls	→	fundecls fundecl | ϵ
fundecl	→	decl | "return" expr
args	→	args ’,’ arg | ϵ
arg	→	IDENT ’:’ TYPE
expr	→	(expr) | expr operator expr-token | expr-token
expr-token	→	brace-list | val | tup | matrix
val	→	IDENT | BOOL | CHAR | STRING | INT | FLOAT
operator	→	<all the operators in our table above>
matrix	→	matrix-rows
matrix-rows	→	matrix-rows ’;’ matrix-row
matrix-row	→	matrix-row ’,’ matrix-row
brace-list	→	list
list	→	val ’,’ list | ϵ
tup-list	→	val ’,’ tup-list | val
tup	→	val ’,’ list | ϵ

Twister compiles down to LLVM.

7 Garbage Collection

Twister offers garbage collection, so that when a variable goes out of scope, its memory is freed. This means that the user does not have to worry about allocating memory or freeing it, and prevents memory leaks.