COMS W4115 : Programming Languages and Translators

TMIL : Text Manipulation Imaging Language

Language Reference Manual

Eli Hamburger (eh2315@columbia.edu)
Michele Merler (mmerler@cs.columbia.edu)
Jimmy Wei (jw2553@columbia.edu)
Lin Yang (ly2179@columbia.edu)

October 18, 2007

1 Introduction

TMIL (pronounced TEE-mil), short for Text Manipulation Imaging Language, is a revolutionary high level programming language that allows users to manipulate text programatically on an image. There are image processing libraries available that can manipulate text, but they are very difficult and cumbersome to use. TMIL was designed from the ground up to have a clean and simple syntax so that users can do repetitive and complicated imaging tasks quickly and efficiently.
Since the interpreter for TMIL is being developed in parallel with this reference manual, the final language may vary slightly from this version of the manual.

2 Lexical Conventions

2.1 Identifiers

An identifier consists of a sequence of upper or lowercase alphabetical characters, numerical digits, and the underscore character. The first character must be an alphabetical character. Identifiers are case sensitive.

2.2 Comments

TMIL employs both C and C++ style comments. Multiline comments start with the characters `"/*"` and terminate with the characters `"*/"`. Single line comments can start and end with the above character sequence or start with the characters `"//"` and end at the end of the line.

2.3 Reserved Keywords

The following identifiers are reserved keywords in the TMIL language.

```
bool          else      main
break         elseif    open
char          exit      position
color         false     return
continue      float     save
coordinate    for       string
create        if        text
do            image     true
drawline      int       while
```

2.4 Types

The following basic and derived types are supported by the TMIL language.

Type	Description
bool	A basic type that can only have two values, true or false.
char	A basic type of items chosen from the ASCII set
float	A basic type that contains an IEEE 32 bit single precision floating point number, with a range of 1.1 x 10e-38 to 3.4 x 10e38.
int	A basic type that contains a 32 bit signed integer, with a range of -2147483648 to 2147483647.
string	A basic type that contains an arbitrary sequence of characters, surrounded by quotation marks.
color	A built-in type that contains three integer values, each value representing a red, green, and blue value.
coordinate	A built-in type that contains two integer values, to indicate the x and y position of a point in an image
image	A built-in type that allocates enough memory to store an image, with certain properties
text	A built-in type that contains a string of text with certain properties

2.4.1 Built-in Types

Color

Color is a type consisting of three integers, one for each chromatic component of a pixel: red, green and blue. When a color object is created, its properties are all initialized to the value zero.

```
color {

   int R;
   int G;
   int B;

}
```

Coordinate

Coordinate describes the position of a point in the image by its x and y coordinates. When a coordinate object is created, its properties are all initialized to the value zero.

```
coordinate {

   int x;
   int y;

}
```

Image

Image is a type built-in to store an image file. It has two properties: integers h and w representing the height and width of the image itself. When an image object is created, its properties are initialized to the values corresponding to the characteristics of the image loaded or created.

```
image {

   int h;
   int w;

}
```

Text

Text is the most complex built-in type, comprehensive of many properties. It contains a string of characters, and allows the user to set its following properties: size, font, position, color, rotation and . When a text object is created, its integer properties and subproperties (colour, position,

rotation, size) are all initialized to the value zero, the font property is initialized to the default font of the system, while name is initialized as an empty string.

```
text {

   string name;
   string font;
   color colour;
   coordinate position;
   int rotation;
   int size;
}
```

2.5 Constants

Three types of constants are allowed in the TMIL language. Integer and floating point constants are represented in decimal format (base 10).

2.5.1 Integer Constants

An integer constant consists of a sequence of 1 or more numerical digits.

2.6 Floating Point Constants

A floating point constant follows closely with the C convention. It has a fractional or an exponential part and can be expressed in decimal or signed exponent notation. A decimal point without a preceding digit is not allowed.

2.7 String Constants

A string constant consists of a sequence of zero or more characters that are surrounded by quotation marks. It cannot include newlines in it. A character escape sequence (\") is required to enclose quotation marks within strings.

2.8 Special Characters

Some characters have special significance in the TMIL language:

Special Character	Use	Example
[ ]	Array delimiter	char arr[20];
{}	Initializer list, function body, or compound statement delimiter	x = 5; x++ ;
( )	Function parameter list delimiter; also used in expression grouping	func();
,	Argument list separator	func2(x, y);
=	Declaration initialize	x = 5;
;	Statement end	x++;
" "	String literal	char str[] = "Hello World";

3 Conversions

The following conversion are valid between types.

3.1 Float to integer

When a floating value is converted to an integral value, the rounded value is preserved as long as it does not overflow.

3.2 Integer and float

When an integral value is converted to a floating value, the value is preserved.

4 Expressions and Operators

Expressions consist of identifiers and operators. The table below lists the precedence and associativity of all operators in TMIL. Described from highest precedence to lowest.

Token	Operators	Associativity
Identifiers, constants, string literal, parenthesized expression	Primary expression	
`() [] .`	Function calls, subscripting, property	L/R
`++ --`	Increment, decrement	L/R
`!`	Logical NOT	L/R
`+ -`	Sign operator	L/R
`* / %`	Multiply, divide, modulus after division	L/R
`+ -`	Plus, minus	L/R
`== !=`	Equality comparisons	L/R
`> >= < <=`	Relational comparisons	L/R
`&&`	Logical AND	L/R
||	Logical OR	L/R
`=`	Assignment	L/R
`,`	Comma	L/R
`<~`	stamp	L/R

4.1 Primary expressions

4.1.1 Basic primary expressions

Identifiers :
An identifier is a reference to an object(ival) or function(id). A description of identifiers can be found in Chapter 2.

Constants :
A constant's type is determined by its form and value. Constant expression's type is identical after the operations are performed. A description of constant can be found in Chapter 2.

String literals :
A string literal is a characters array.

```
string literals:          '"' (.)* '"' ;
```

Parenthesized expression:
A parenthesized expression's type is the same as what is parenthesized.

```
parenthesized expression:    '('expression')' ;
```

4.1.2 Subscripts

The element of an array can be accessed by having an index number within square brackets after the array's object identifier. The return value is the same as the type of the array. For example, array[i] returns the ith element of the array array.

```
array element:          ival '['index number']' ;
index number:           digit(digit)* ;
```

4.1.3 Property Operator

An object primary expression followed by a period and the name of a type property can access this property. The return type is the same as the member accessed.

```
type number:            ival'.'(id|ival) ;
```

4.1.4 Function calls

A declared function followed by a pair of parenthesizes with possible variables in between is a function call. The return value is the declared function return type.

```
function call:      id'('parameters')';
parameters:         expression(',' expression)*
                    |  E ;
```

4.2 Unary operators

4.2.1 Increment and decrement

An object primitive expression followed by double plus signs or double minus signs is increment or decrement. The expression's type can only be int. The return type is int.

```
increment:      ival '++' ;
```

`decrement:` *ival* `'--' ;`

4.2.2 Logical NOT

The logical NOT operator followed by a expression returns the opposite of the expression. The return type is boolean. If the expression's type is int or float, it will return true if the expression's value is zero and return false if the value is not zero. The expression cannot any type other than int, float or boolean.

`logical not:` `'!'` *expression* `;`

4.2.3 Sign operator

A plus or minus sign followed by a primitive expression returns 0 plus or minus the expression. The expression's type can only be int or float. The return type is the same as the expression's type.

`sign operator:` `'+ | -'` *expression* `;`

4.3 Arithmetic operators

4.3.1 Multiply and divide

A primitive expression followed by a multiple sign or a divide sign, followed by a primitive expression returns the product of the two expressions. The expressions' type can only be int or float. The return type is int if both expressions' type are int, otherwise, the return type is float.

`multiply/divide:` *expression* `'* | /'` *expression* `;`

4.3.2 Modulus after division

A primitive expression followed by a modulus sign, followed by a primitive expression returns the modulus after the division. The return type is int. If the expression's type is float, it will be truncated before the operation is applied.

`modulus after division:` *expression* `'%'` *expression* `;`

4.3.3 Plus and minus

A primitive expression followed by a plus sign or a minus sign, followed by a primitive expression returns the sum or difference of the two expressions. The expressions' type can be int, float or string. The return type is int if both expressions' type is int. The return type is double if the

expressions' type are int and float respectively or both float. The return type is string when both expressions' type are string and the operator is plus. In such situation, the returned value is the concatenated string of the first string and the second string. The expressions' type can only be one of the situations listed above.

`add minus`: *expression* `'+ | -'` *expression* `;`

4.4 Relational operators

4.4.1 Equal

A primitive expression followed by double equal signs, followed by a primitive expression returns true when the two expressions are the same. Otherwise, it returns false. The return type is boolean. If the type of the expressions are int and/or float, the operator will compare the two expressions' value. If both expressions are string, it will return true when two strings are the same, otherwise it will return false. The expressions' type can also both be boolean. The expressions' type can only be one of the situations listed above.

`equal comparison`: *expression* `'=='` *expression* `;`

4.4.2 Not equal

A primitive expression followed by a not equal sign, followed by a primitive expression returns true when the two expressions are different. Otherwise, it returns false. The return type is boolean. If the type of the expressions are int and/or float, the operator will compare the two expressions' value. If both expressions are string, it will return true when two strings are different, otherwise it will return false. The expressions' type can also both be boolean. The expressions' type can only be one of the situations listed above.

`Not equal comparison`: *expression* `'!='` *expression* `;`

4.4.3 Relational comparisons

Greater than
A primitive expression followed by a greater than sign, followed by a primitive expression returns true when first expression is greater than the second expression. Otherwise, it returns false. The return type is boolean. The expression's type can only be int or float.

`greater than`: *expression* `'>'` *expression* `;`

Not greater than
A primitive expression followed by a not greater than sign, followed by a primitive expression returns true when first expression is not greater than the second expression. Otherwise, it returns

false. The return type is boolean. The expression's type can only be int or float.

`not greater than:` *expression* `'<='` *expression* `;`

less than
A primitive expression followed by a less than sign, followed by a primitive expression returns true when first expression is less than the second expression. Otherwise, it returns false. The return type is boolean. The expression's type can only be int or float.

`less than:` *expression* `'<'` *expression* `;`

Not less than
A primitive expression followed by a not less than sign, followed by a primitive expression returns true when first expression is not less than the second expression. Otherwise, it returns false. The return type is boolean. The expression's type can only be int or float.

`not less than:` *expression* `'>='` *expression* `;`

4.5 Logical operators

4.5.1 Logical AND

A primitive expression followed by a logical AND sign, followed by a primitive expression returns true when both expressions are true. Otherwise, it returns false. The return type is boolean. The expressions' type can only be boolean, int or float. If the expression's type is not boolean, the expression is true when the value is not zero. Otherwise it is false.

`logical AND:` *expression* `'&&'` *expression* `;`

4.5.2 Logical OR

A primitive expression followed by a logical OR sign, followed by a primitive expression returns true when at least one of the expressions is true. Otherwise, it returns false. The return type is boolean. The expressions' type can only be boolean, int or float. If the expression's type is not boolean, the expression is true when the value is not zero. Otherwise it is false.

`logical OR:` *expression* `'||'` *expression* `;`

4.6 Assignment Operator

An object primitive expression followed by an assignment sign, followed by a primitive expression will assign the second expression's value to the first expression. If the ival's type is boolean and

the expression's type are int and/or float, ival is false when expression's value is zero, otherwise it's true. If the ival's type is int and expression's type is float, ival will be assigned the value of the expression after truncated. If the ival's type is int or float and the expression's type is boolean, ival will be 1 if the expression is true, otherwise it's false. If the ival is string then the expression must be string, too. The ival and expression's type can only be one of the situations listed above.

`assignment`: *ival* `'='` *expression* `;`

4.7 Comma

Comma is used to separate expressions.

`comma`: *expression* `(',' expression )+ ;`

4.8 Stamp Operator

An image type expression followed by a stamp symbol, followed by a text type expression will print the string of the text expression on the image file of the image expression, with all the characteristics specified by the text object attributes.

`stamp`: *imageexpression* `<~` *textexpression*

5 Declarations

A declaration specifies the interpretation of identifier(s) or function(s). Variables, arrays and functions must be declared before being referred or called.

5.1 Variable declaration

One of more variables can be declared in each declaration. Only the same type of variables can be declared in each declaration. The value of the identifier can be assigned to the identifier when being declared. The type of the ival and the expression must be the same.

`variable declaration`:
type ival`('='`*expression*`)?  (',' ival('='`*expression*`)?)*;`

5.2 Array declaration

One of more arrays can be declared in each declaration. Only the same type of arrays can be declared in each declaration. A arraylist can be assigned to the identifier when being declared. The type of the arraylist and the ival must be the same. If the size of the arraylist is smaller than the

size of the array, the elements with smaller index number will be assigned first. If the arraylist's size is bigger than the size of the array, the exceeded elements in the arraylist will be ignored.

```
variable declaration:
type ival '[' size ']' ( '=' arraylist)?
(',' ival '[' size ']' ( '=' arraylist)?)*;

arraylist:        '{' ival(','ival)* '}' ;
```

5.3 Function declaration

```
Function declaration:     type id '(' parameters ')' ;
parameters:               type (','type)*
                          |  E ;
```

6 Statements

In TMIL, statements are usually executed in sequence. There are a few exceptions, specified in the following paragraph.

6.1 Expression Statement

Expression statements are the most common statements in the TMIL language. Usually they are assignments or function calls, and take the form of

expression ;

6.2 Compound Statements

Compound Statements consist of several statements enclosed in braces, which are considered as a single statement:

{(*statement*;)∗}

6.3 Conditional Statement

There are two forms of the conditional statement:

`if` (*expression*) *statement1*
`if` (*expression*) *statement1* `else` *statement2*

In both cases *expression* is evaluated. If it is non-zero, *statement1* is executed; in the second case, if *expression* is zero, then *statement2* is executed. `Else` ambiguity is resolved by connecting the `else` with the nearest elseless `if`.

6.4 While Statement

The `while` statement takes the form of

`while` (*expression*) *statement*

Expression is evaluated before the execution of *statement*. If it is non-zero, *statement* is executed. The process is iterated until *expression* evaluates to zero.

6.5 Do Statement

The `do` statement takes the form

`do` *statement* `while` (*expression*) ;

Statement is executed iteratively until *expression* evaluates to zero. At each iteration, *Expression* takes place after *statement* is executed.

6.6 For Statement

The for statement takes the form

`for` (*expression1* ; *expression2* ; *expression3*) *statement*

This statement is equivalent to

expression1 ;
`while` (*expression2*) {
    statement
    expression3 ;
}

6.7 Return statement

Functions return to their caller via the `return` statement, which takes the form

`return` (*expression*)? ;

Either no value (null) or the value of *expression* is returned to the caller of the function, assuming

the function is declared to return a value of matching type. If a function is not declared to return a matching type of *expression*, and *expression* is returned, an error occurs. Similarly, it is an error for a function declared to return null to include *expression* in the return statement.

7 Scope Rules

The scope rules in TMIL are very similar to those in C or C++. A variable or function is unavailable (out of scope) until it declared. Functions cannot be nested and therefore cannot be overridden after they are declared. They are declared in the global scope and available until the program completes execution. A variable is available until the end of the block (defined by), in which it was declared, is reached. In the case of variable declared in the global scope, the variable only goes out of scope on program termination. Nested blocks can access variables defined in parent blocks. If a new variable is declared in a nested block with the same name as a variable from a parent block, said variable was overridden. The nested block will then only have access to the new variable from point of declaration. When the block is closed, the original variable will return to scope.

8 Built-in Functions

The TMIL language includes some built-in functions which are available to the user.

8.1 Open

This function allows the user to load an image file.

image `Im = open(`*string* `filename) ;`

8.2 Create

Create allows the user to create a new image, without having to rely on preexisting files, by providing the size and the background color. The syntax is the following.

image `Im = create(`*int* `sizex,` *int* `sizey,` *color* `backgroundcolor) ;`

8.3 Save

This function allows the user to save an image after the processing has been performed.

`save(`*image* `Im,` *string* `filename) ;`

8.4 Drawline

Drawline draws a line from point a point p1 to a point p2, with the properties specified by the user.

```
drawline(image Im, coordinate p1, coordinate p2, color col, int width) ;
```

9 Example

Here is a sample program:

```
void doSomthing(string s1, string s2) {
        image im = open(s1);
        color col;
        col.r = col.g = col.b = 150;
        text t;
        t.name = s2;
        t.colour = col;
        t.colour.g = 100;
        t.position.x = 50;
        im <~ t;
}

void main(){
        doSomthing("myImage.jpg","haha!");
}
```

This program would print *haha!* on the image `myImage.jpg` at y=0, x=150 in the color specified in the function.