

COMS W4115
Programming Languages and
Translators

Prof. Stephen Edwards

LANGUAGE REFERENCE
MANUAL

MASC

Fall 2007

Christos Angelopoulos
Kristina Chodorow
Michael Masullo
Vaibhav Saharan
{ca2269,kc2424,msm2148,vs2220}@columbia.edu

1. Introduction

MASC is a language for creating fast information extraction tools. The MASC shell is embedded in the web-browser and allows users to write code snippets in MASC while browsing the web. It offers the user a number of interesting features to process the data on the web-page, like crawling the web-page, extracting various types of information, saving different types of media etc. Furthermore, it also allows the user to author general purpose stand-alone programs. This is as important component of the language as it gives the user, the ability to process the data available on web page in a faster and more efficient manner. It is more stable, concise, and occupies less memory than existing GUI tools.

MASC helps to work with the information available on the web in a smarter way. Consider some examples to this end. Search results often return web-pages riddled with advertisements and irrelevant data. MASC can help derive the relevant information and meaningful links from such results. Alternatively, it can help crawl a web page to find links that go to relevant results and even save such results for future reference. It can help save the various media elements like images, sound snippets, videos etc., embedded in a web page. Such uses of MASC unbound.

This document, as the name suggests, serves as a reference to the many features and constructs of this language and also offers a few examples in an attempt to exhibit the use of some of these features.

2. General Lexical Conventions

Any given MASC token, falls into one of these broad categories: identifiers, keywords, constants, strings, operators and general separators. With regards to these general separators, they include the likes of spaces, tabs, newline characters and comments, which are generally ignored by the compiler, with the exception that they serve to separate tokens. As a general rule, one of these characters are required to separate any two adjacent identifiers and constants.

2.1 Comments

A comment in MASC begins with a (: and ends with a corresponding :)

2.2 Identifiers

Identifiers consist of a sequence of letters and digits with the condition that the first character must be an alphabet from the English language or an underscore. All identifiers are case sensitive.

2.3 Keywords

MASC reserves the following identifiers as keywords to represent specific pre-defined functionalities:

for if int float string else return void

2.4 Constants

MASC has two categories of constants described as follows:

2.4.1 Integer Constants

Integer constants are simply, a sequence of digits.

2.4.2 Floating Constants

A floating constant consists of an integer part, followed by a decimal point and a fraction part. Both the integer part and the fraction part are essentially, a sequence of digits.

2.5 Strings

Strings in MASC, consist of a sequence of characters surrounded by a pair of double quotes, for example, "jack".

3. Data Types

Identifiers in MASC are interpreted based on their assigned data type, which precede the identifier in its declaration. MASC has the following built-in data types:

3.1 Basic Types

string: This data type represents a sequence of any number of characters. It is represented by the keyword *string*.

int: This data type represents all 32-bit whole numbers. It is represented by the keyword *int*.

float: This data type represents floating-point numbers. It is represented by the keyword *float*.

webpage: This data type represents the parsed data that is returned by the HTML parser after processing an HTML page. It is represented by the keyword *webpage*.

3.2 Derived Types

list: This data type is essentially, an array of any of the basic types. It is represented by the keyword *list*.

In addition to this, functions, which return objects of a given basic type, are also admissible constructs.

4. Conversions

Since there are different data types, using combinations of them with binary operators will cause them to be cast. In general, given a precedence of float, int, string, a higher type is cast down to a lower type. The specifics are given in this section. These conversions only apply to float, int, and string. There are no conversions to or from webpage or list.

4.1 Float and int

In all cases of binary operators with a float on one side and an int on the other, the float will be cast as an integer by truncating the decimal portion, no matter which side of the operator it is on. Therefore, $42.5 - 42 = 0$ and $26.5 == 26$ is true.

4.2 Float and string

Since the + operator is overloaded as the concatenation operator for string, it is the only valid arithmetic operator involving float and string. As stated above, the float shall be cast into a string. Thus, $67.5 + \text{"The end"} = \text{"67.5The end"}$ and this holds no matter which side of the + operator the float is on. All comparison operators still hold, meaning $54.5 > \text{"26.4"}$ is true.

4.3 Int and string

As with floats, integers are converted to strings when used with a string and a binary operator. Again, since the only arithmetic operator that can be used with two string is + (as concatenation), $\text{"Home Runs"} + 12 = \text{"Home Runs12"}$ is valid, but $12 - \text{"Home Runs"}$ is a syntax error. As before, comparison operations are still valid, so $59 <= \text{"blueberries"}$ is true.

5. Expressions

The following is a list of expression operators, in order of precedence, from greatest to least. Within each subsection, operators have the same precedence and associative is listed with each operator.

5.1 Primary Expressions

Primary expressions normally group from left to right.

5.1.1 *identifier*

An identifier is a primary expression, as long as it has been correctly declared.

5.1.2 *constant*

A decimal number (that is, a number in the decimal (Base 10) system) such as 96 or 3.14 or a single character or multiple characters contained in double quotes like “x” or “supercalifragilisticexpialidocious” is considered a primary expression.

5.2 Unary Operators

These operators are right associative.

5.2.1 *!expression*

The logical NOT operator changes the value of a non-zero expression to 0 and a 0 expression to 1. These results are integers. This operator can only be used with ints and floats.

5.3 Multiplicative operators

These operators group from left to right.

5.3.1 *expression * expression*

The binary * operator multiplies the expression on the left of the * by the expression on the right. As mentioned in Section 4, the result will be a float if and only if both expressions are floats; otherwise, the result is an integer. A string in either expression results in a syntax error.

5.3.2 *expression / expression*

The binary / operator divides the expression on the left by the expression on the right. Once again, a string in either expression results in a syntax error, while both expressions must be floats for the result to be a float. Otherwise, the result is an integer.

5.4 Additive Operators

These operators also group from left to right.

5.4.1 *expression + expression*

The result is the sum of the two expressions if neither expression is a string. The sum will be returned as a float if both expressions are floats and as an integer if one or both of the expressions is an int. If either expression is a string, the binary + operator becomes a concatenation operator and returns a string that is the concatenation of both expressions. If the other expression is not a string, it will be converted to a string as per Section 6.

5.4.2 *expression – expression*

The binary – operator returns the difference of the two expressions. The result will be an integer unless both expressions are floats, in which case a float will be returned. A string in either expression results in a syntax error.

5.5 Relational Operators

These group from left to right, but it is not advised to put multiple relational operators together such as $a > b > c$.

5.5.1 *expression < expression*

5.5.2 *expression > expression*

5.5.3 *expression <= expression*

5.5.4 *expression >= expression*

Less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=) all return 0 if false and 1 if true. Once again, these results are integers and type conversion is done according to Section 4. Note that if either expression is a string, the other expression will be cast as a string. This comparison can be useful for alphabetizing as the result will be based on ASCII values of the first character of each string.

5.6 Equality Operators

5.6.1 *expression == expression*

5.6.2 *expression != expression*

These operators are just like the relational operators above, but they have lower precedence. == is the equality operator and != is the inequality operator.

5.7 *expression & expression*

This is the logical AND operator. It returns 1 if both expressions are non-zero and 0 otherwise. The second expression is not evaluated if the first expression evaluates to 0. The & operator operates left to right. Note that this is NOT a bitwise comparison.

5.8 *expression | expression*

This is the logical OR operator, which returns 0 if both expressions are zero and 1 otherwise. The second expression is not evaluated if the first expression evaluates to something other than 0. Like & above, the | operator operates left to right, and is NOT a bitwise comparison.

5.9 *identifier = expression*

The assignment operator places the result of the expression on the right hand side of the = operator into the identifier on the left hand side of it. The expression will always be cast as the type of the identifier, so if the expression evaluates to 47.42 but the identifier is an integer, the truncated value of 47 will be the value of the identifier.

6. Declarations

6.1 Type Specifiers

The type specifiers are float, int, string, webpage, and list. It is required to use a type specifier before declaring a variable.

6.2 Function Declarations

All functions must be declared and defined at the same time. Therefore, the form that functions take is:

type-specifier identifier (argument [, argument]*) { function-body }

It should be noted that MASC also supports void functions, which can be declared by using the keyword void in place of a type specifier. Also, any function can contain 0 or more arguments, but all arguments must be of the form:

type-specifier identifier

Multiple arguments must be separated by commas.

7. Statements

Statements are executed in sequence unless otherwise specified.

7.1 Expression statements

This is the most common type of statement and have the form
expression;

where the semicolon serves as a statement terminator. This type of statement is used most often for assignments and function calls.

7.2 Compound statements

This statement is used for when multiple actions are required. A compound statement consists of a statement list inside of curly braces like this:

```
compound-statement:  
    {statement-list}  
statement-list:  
    statement  
    statement statement-list;
```

7.3 Conditional statement

There are two forms of the conditional statement:

```
if (expression) compound-statement
```

OR

```
if (expression) compound-statement else compound-statement
```

First, the expression in parentheses is evaluated. If the result is non-zero, the first compound statement is executed. Otherwise, the compound statement after the else (if there is one) is executed. Each else is matched up with the first elseless if preceding it.

7.4 While statement

While statements have the form:

```
while (expression) compound-statement
```

The compound statement is executed as long as the expression in parentheses is non-zero. The expression is evaluated each time before the compound statement is executed.

7.5 Return statement

Return statements can take two forms:

```
return;
```

OR

```
return (expression);
```

In the first form, nothing is returned to the function that called it. An error will result if the calling function is expecting something. In the second form, the expression in parentheses is returned to the calling function. If the expression is not of the same type as the calling function was expecting, it will cast it to the same type, if possible.

7.6 Null statement

A null statement is nothing but a semicolon like so:

;

It is often used for while statements that have no body.

8. Scope Rules

MASC programs can call precompiled functions from libraries. However, variables may only be declared within the main program. A variable exists after it is declared within the current block. If there are nested blocks, it exists in any sub-blocks. Global variables can be declared at the beginning of the program, before the function prototype section.

9. Examples

9.1 Example 1

```
void
list_print(List<string> l)
{
    while(hasElements(l)) {
        print(nexElement(l));
    }
}

void main()
{
    int f = FileOpen("Data.txt");
    string line;
    string str;
    string l[];

    (: read every line of the file until the end of file is found :)
    while((line getline(f)) != EOF) {
        (: match a word followed by a space and a number or
        for a number followed by a space and a word :)
        if((str = match(line, "((\w)+/s(/d)+) | ((\d)+\s(\w)+)")) !
= -1) {
            (: if such a match is found then add it to the list :)
            add(l, str);
        }
    }
    (: print the list :)
    list_print(l)
}
```

9.2 Example 2

```
void main()
{
    int n;
    print("Give a positive integer number and I will print its
factorial");
    read(n);
    while(n < 0) {
        print("Only positive integers please");
        read(n);
    }
    print("Factorial of ", n, " is ", n!);
}
```

9.3 Example 3

```
void main() {
    string st = getCode();
    string str = parseSite();
    string s;
    string a[];
    if(contains(getHeader(st), "Addresses")) {
        (: match(regex) is a built-in library function
        returning one match at a time :)
        while(s = match(st, "(\\w|\\W|\\d)+@(\\w|\\W|\\d)+.(\\w|\\W)+")) {
            print(s);
        }
    }
    else {
        (: matchtag(tag_type[, regex]) is another library
        function returning all matches as an array :)
        a = matchtag("a href", "addresses");
        print(a);
    }
}
```

9.4 Example 4

```
void
sort_list(List<int> l)
{
    int i = 0;
    int j = 0;
    int ls[];
    int size = sizeof(l);
    int temp;
    (: sort the list using bubblesort :)
    for(i = 0; i < size; ++i) {
        for(j = 0; j < size; ++j) {
            if(get(l, i) > get(l, j)) {
                temp = get(l, i);
                set(l, i, get(l, j));
                set(l, j, temp);
            }
        }
    }
}

void
main()
{
    int f = FileOpen("Data.txt");
    string line;
```

```

string str, s;
int l[];
int i;
float d;
(: read a line from file till the end of file is found :)
while((line getline(f)) != EOF) {
    (: match a number followed by a space and a number :)
    if((str = match(line, "(/d+) (\s) (\d+)")) != -1) {
        (: if such match is found tokenize the string :)
        i = 0;
        d = 0;
        (: add the numbers and calculate the average :)
        while((s = tokenizer(str, " ", i) != -1) {
            d += toInt(s);
            ++i;
        }
        d /= i;
        (: add it to the list :)
        add(l, d);
    }
}

(: sort the list :)
sort_list(l);
}

```

9.5 Example 5

```

void
list_print(string l[])
{
    while(hasElements(l)) {
        print(nexElement(l));
    }
}

void
main()
{
    webpage WP;
    string refList[];
    (: get the links to other sites from the current :)
    getReferences(WP, refList);
    (: save all the pictures displayed on the webpage
    to the desktop folder, unless specified otherwise :)
    savePictures(WP);
    (: print the list :)
    list_print(l)
}

```

9.6 Example 6

```
void
main()
{
    webpage WP;
    string text;
    (: print the title of the current webpage :)
    print(getTitle(WP));
    (: get the parsed html text from the website :)
    text = getParsedText(WP);
    (: get every number of the text and print it :)
    while((str = match(text, (\d+)) != -1) {
        print(str);
    }
}
```