The MaSH Programming Language At the Statements Level

Andrew Rock
School of Information and Communication Technology
Griffith University
Nathan, Queensland, 4111, Australia
a.rock@griffith.edu.au

June 14, 2010

1 Introduction

This document defines the MaSH programming language at its statements level. The statements level omits all control structures. It permits focus on declaring and using variables and constants, and using the APIs provided by MaSH environments.

2 Lexical syntax

The MaSH lexical syntax describes how to make up the tokens that make up the MaSH language out of individual characters.

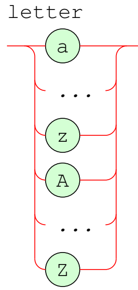
We can describe a syntax either *informally* in plain language, or *formally* with special languages like Extended Bachus-Naur Form (EBNF) or equivalent diagrammatic representations such as railroad diagrams. This document uses English and railroad diagrams.

2.1 Kinds of characters

Certain kinds of characters are appropriate in different places in a MaSH program.

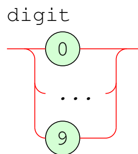
2.1.1 Letters

Letters are letters from the Roman alphabet, in either lower or upper case.

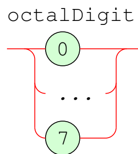


2.1.2 Digits

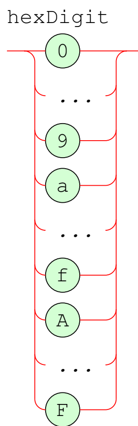
Ordinary digits are the decimal digits from `0` to `9`.



Octal digits are those we use when writing a number in octal (base 8): `0` to `7`.



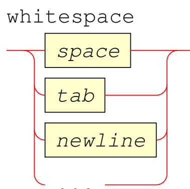
Hexadecimal digits are those we use when writing a number in hexadecimal (base 16): `0` to `9` and the letters `A` to `F` in upper or lower case.



2.1.3 Whitespace characters

Whitespace characters are those characters that exist in a text file, but don't normally show up as a visible glyph when the file is viewed, and cause the

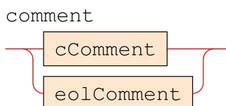
visible characters to be spaced out either horizontally or vertically. Common whitespace characters include the ordinary space (you put between words when you press the spacebar on your keyboard), the tab character used for aligning text, the newline character that marks the end of each line, and some much more rarely used others.



In a program, the whitespace is really only significant if it separates two tokens (e.g. identifiers) that could otherwise be taken as one token if there were not whitespace characters between them. However, the use of whitespace is important to improve the readability of a program for humans.

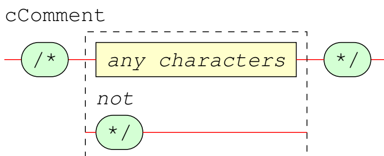
2.2 Comments

Comments are text for human readers only and are treated like whitespace by compilers. Java and MaSH have the same syntax as C++. One kind is the same as those used in the C language.



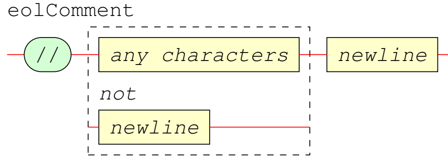
2.2.1 C-style comments

A C-style comment may extend over may lines of text. It starts with the characters `/*` and ends with `*/`. There can be any amount of any characters other than the sequence `*/` in between.



2.2.2 C++-style end-of-line comments

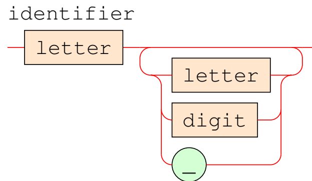
A C++-style end-of-line comment is for shorter comments, starts with `//` and ends at the end of the line.



2.3 Identifier and keywords

2.3.1 Identifiers

MaSH programmers create new names for their variables and constants. These names are called *identifiers*. Identifiers start with a letter, which may then be followed by additional letters, digits or underscores.



2.3.2 Keywords

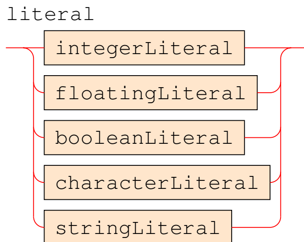
Some words, made up only of letters and therefore meeting the description of identifiers given above, are special keywords in the Java language (and some of them in MaSH too) and therefore can not be used as names for variables, constants and methods. The full list of these *reserved words* or *keywords* is given in table 1.

`abstract`	`else`	`interface`	`switch`
`assert`	`enum`	`long`	`synchronized`
`boolean`	`extends`	`native`	`this`
`break`	`false`	`new`	`throw`
`byte`	`final`	`null`	`throws`
`case`	`finally`	`package`	`transient`
`catch`	`float`	`private`	`true`
`char`	`for`	`protected`	`try`
`class`	`goto`	`public`	`void`
`const`	`if`	`return`	`volatile`
`continue`	`implements`	`short`	`while`
`default`	`import`	`static`	
`do`	`instanceof`	`strictfp`	
`double`	`int`	`super`	

Table 1: The Java (and therefore MaSH) keywords.

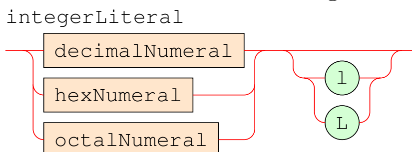
2.4 Literals

Literals represent a value explicitly. Every literal has a type.

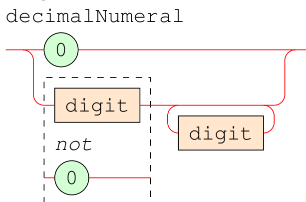


2.4.1 Integer literals

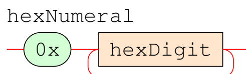
Integer literals may be written in decimal, hexadecimal or octal. An `l` or `L` at the end of the integer literal, means that the type on the integer is `long`. By default, the type is `int` unless the value is too big to fit in an `int`.



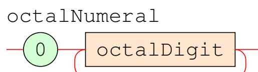
Unless the whole decimal numeral is just `0`, a decimal numeral must not start with a zero or it will be assumed to be an octal numeral. A non-zero decimal numeral must start with a digit other than `0`, and then more digits may follow.



A hexadecimal numeral starts with `0x`, which is followed by one or more hexadecimal digits.



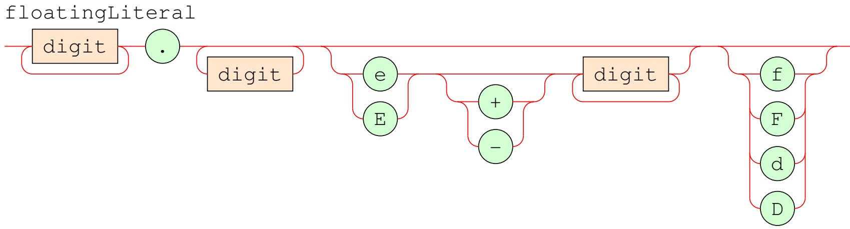
An octal numeral starts with `0`, which is followed by one or more octal digits.



2.4.2 Floating point literals

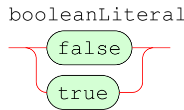
A floating point literal is used to create values of type `float` or `double`. These types are used for fractional values and for very large or very small numbers. A floating point literal starts with a whole number part, then a decimal point,

then optionally the fractional digits, then optionally an `e` or `E` that introduces an optionally signed exponent. An `f` or `F` at the end of the floating point literal indicated that the type is `float`. A `d` or `D` at the end of the floating point literal indicated that the type is `double`. By default, the type is `double`.



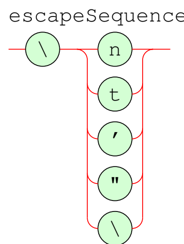
2.4.3 Boolean literals

Boolean values are represented by the keywords `false` and `true`.



2.4.4 Special character escape sequences

To put some special characters in character and string literals we use a special sequence that starts with the special *escape* character \ (a backslash). The next character indicates which character the whole escape sequence is representing. The escape sequences and what they mean are listed in table 2.

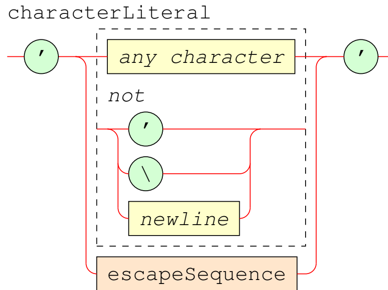


escape sequence	*represents a ...*
`\n`	newline character
`\t`	tab character
`\'`	single quote (apostrophe) character
`\"`	double quote character
`\\`	backslash character

Table 2: Special character escape sequences.

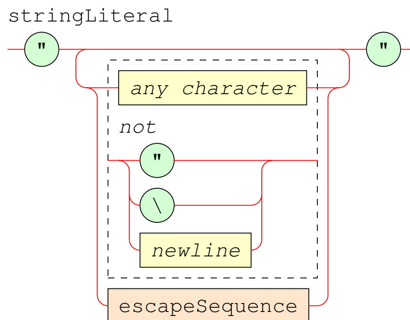
2.4.5 Character literals

A character literal, of type `char`, is usually just that character displayed between single quotes. The single character can't be a backslash, single quote or the end of a line. Instead of a single character, you can put one of the special character escape sequences between the quotes.



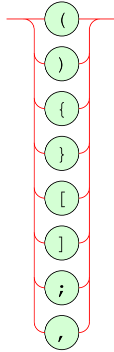
2.4.6 String literals

A string literal, of type `char`, is usually just zero or more characters displayed between double quotes. The characters can't be backslashes, double quotes or the ends of lines. Strings can also contain the special character escape sequences.



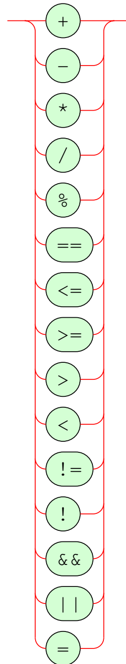
2.5 Separators

These characters are the punctuation symbols of the language.



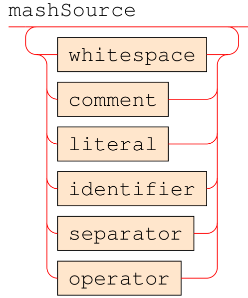
2.6 Operators

These symbols made up of one or more special characters are represent actions like addition and assignment.



2.7 A MaSH input source

A MaSH input source (a mash program) is made up of a sequence of these elements in any order (as far as the lexical syntax is concerned): whitespace; comments; literals; identifiers (including keywords); separators; and operators.

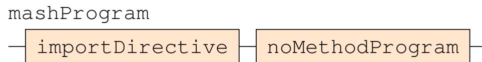


3 Context-free grammar

The MaSH context-free grammar describes how to order MaSH tokens, but does not define the context rules (rules like "Declare a variable before using it."). Because any amount of whitespace (including comments) is permitted before, after and between tokens, whitespace (including comments) is not mentioned again in this grammar.

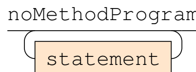
3.1 MaSH programs

A program begins with an import directive that selects the environment within which this program will run. Note that without selecting an environment, the program could not do anything that the user could observe. It could not display anything or control any hardware. The environments provide the methods by which these things can be done. After the import directive, comes the program in the no-method program structure.



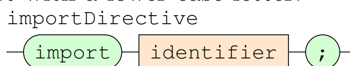
3.1.1 No-method programs

A no-method program consists of a sequence of statements.



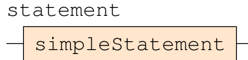
3.2 Import directives

An import directive starts with the keyword **`import`**, which is followed by an identifier (the name of the environment), and ends with a semicolon. Environment names should start with a lower case letter.



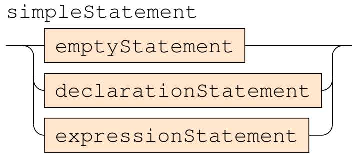
3.3 Statements

Statements are only simple statements at this level.



3.4 Simple statements

A simple statement is either: the empty statement (which does nothing); a declaration statement; or an expression statement.

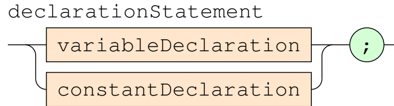


3.5 Empty statements

An empty statement does nothing and consists solely of its terminating semicolon.

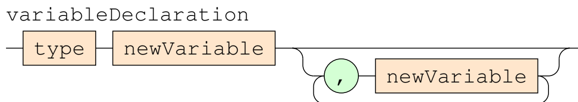
3.6 Declaration statements

Declaration statements declare either constants or variables and end with a semicolon.



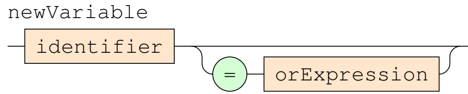
3.7 Variable declarations

A variable declaration begins with a type and then introduces at least one new variable. Multiple new variables are separated by commas. The variable declaration is terminated by a semicolon.



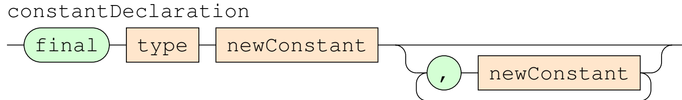
3.7.1 New variables

New variable introductions consist of the name of the new variable followed optionally by the assignment operator and an expression representing the initial value for this new variable. Variable names should start with a lower case letter.



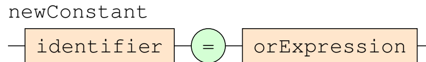
3.8 Constant declarations

A constant declaration is much like a new variable declaration, but starts with the keyword `final`. Then follows the type and then the introductions of at least one new constant. Multiple new constants are separated by commas. The constant declaration is terminated by a semicolon.



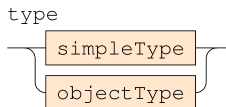
3.8.1 New constants

New constant introductions consist of the name of the new constant followed by the assignment operator and an expression representing the initial value for this new variable. Unlike new variables, constants must be inititalised immediately and can not be assigned new values later.



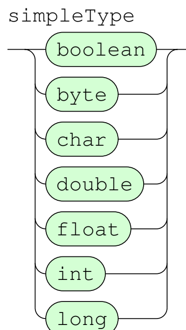
3.9 Types

A type is the name of either a simple type or a compound type.



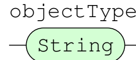
3.9.1 Simple types

The simple types contain one item of logical, numeric or character data.



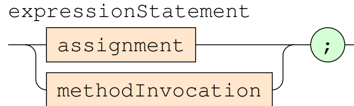
3.9.2 Object type

The only object type is the string.



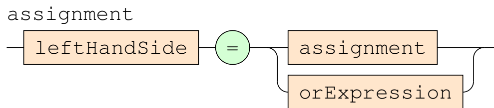
3.10 Expression statements

Expression statements consist of an assignment or a method invocation followed by a semicolon. A method invocation in this context should be a procedure call.



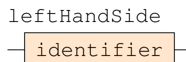
3.11 Assignments

An assignment is an expression that has at least one assignment operator (=) in it. Only certain things are permitted on the left hand side of an assignment operator. After the rightmost assignment operator comes an expression that contains any of the other operators.



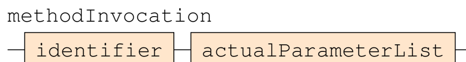
3.11.1 Left hand sides

The only thing allowed on the left hand side of an assignment operator is the name of a variable (not a constant).



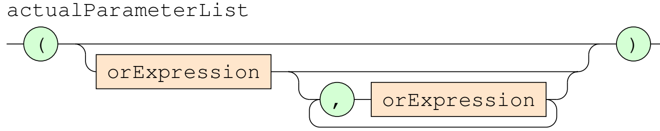
3.12 Method invocations

A method invocation consists of the name of the method followed by the actual parameter list.



3.12.1 Actual parameter lists

An actual parameter list is enclosed by parentheses and contains a sequence of zero or more expressions separated by commas.

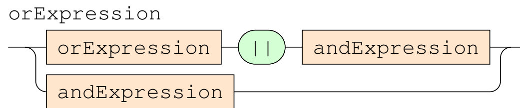


3.13 Expressions

The grammar for expressions defines the order of precedence of the various operators. The lowest precedence operator is the assignment operator (already dealt with above), the next lowest precedence operator is the "or" operator (||).

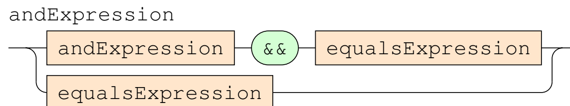
3.13.1 Or expressions

An "or" expression is a sequence of "and" expressions separated by the "or" operator (||).



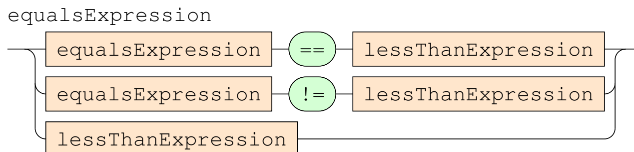
3.13.2 And expressions

An "and" expression is a sequence of "equals" expressions separated by the "and" operator (&&).



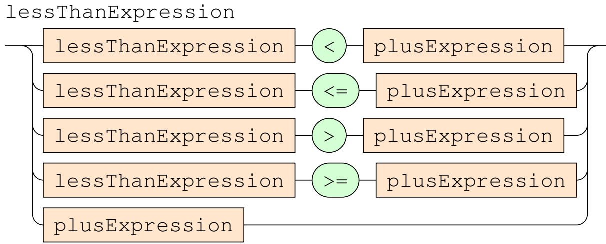
3.13.3 Equals expressions

An "equals" expression is a sequence of "less-than" expressions separated by the "equals" operator (==) or the "not-equals" operator (!=).



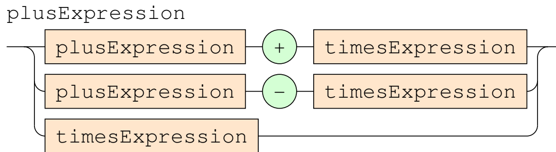
3.13.4 Less-than expressions

A "less-than" expression is a sequence of "plus" expressions separated by the "less-than" operator (<), the "less-than-or-equals" operator (<=), the "greater-than" operator (>), or the "greater-than-or-equals" operator (>=).



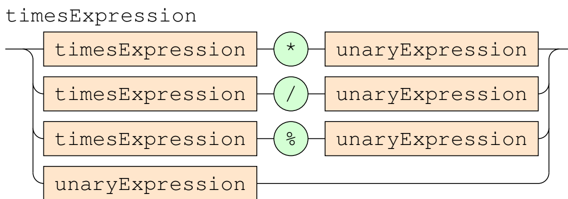
3.13.5 Plus expressions

A "plus" expression is a sequence of "times" expressions separated by the "plus" operator (+), or the "minus" operator (-).



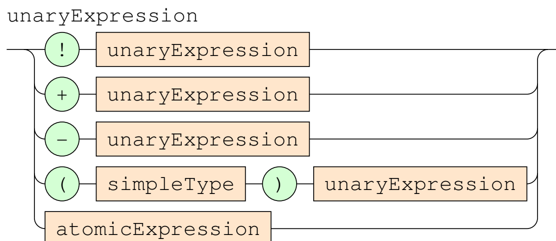
3.13.6 Times expressions

A "times" expression is a sequence of "unary" expressions separated by the "times" operator (*), the "divide-by" operator (/), or the "modulo" operator (%).



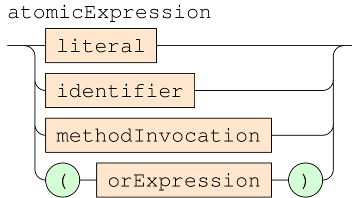
3.13.7 Unary expressions

A "unary" expression has zero or more *unary* operators followed by an atomic expression. The unary operators are "plus" (+), "minus" operators (-), "not" (!), or a *type cast*. A type cast is written as a simple type name between parenetheses.



3.13.8 Atomic expressions

An "atomic" expression is either: a literal; an identifier (a variable or constant name); a method invocation; or an "or" expression enclosed by parentheses. A method invocation in this context must be of a function (not a procedure).



4 Style conventions

These are rules all Java programmers adhere to, even if the Java compiler does not enforce them. The MaSH compiler will warn about these things.

- Variable names must start with a lower case letter.
- Any letters in a constant's name must be in upper case.
- All method names start with a lower case letter.

These are MaSH-specific rules, but are consistent with the equivalent in Java.

- Programs are saved in files that have names that start with an upper case letter.
- A program's file name must match the lexical definition of an identifier (with `.mash` added).
- All environment names are identifiers that start with a lower case letter.

5 Semantic rules

5.1 Variables

- A variable must be declared before it is used.
- Before a variable's value can be used it must have been assigned one.

5.2 Constants

- A constant must be declared before it is used.
- A constant must be assigned a value at the same time as it is declared.
- A constant may not be reassigned a second value.

5.3 Assignment compatibility

- Only a variable name may appear on the left hand side of the assignment operator (=).
- The value on the right of the assignment operator must be *assignment compatible* with the type of the variable on the left.
- If two types are the same, they are assignment compatible.
- If the expression on the right has a type that can automatically be promoted by a conversion that does not lose information to the type of the variable on the left, they are assignment compatible.

5.4 Automatic promotion and type casts

- Expressions that combine values of different types are usually only possible where the two types are the same or one type can be promoted by a conversion that does not lose information to the other type.
- Table 3 lists all of the automatic promotions that are possible because they do not lose information.
- Use a type cast to force a conversion between simple types where they are not normally permitted.

5.5 Method calls

- To call a method provided by the environment, it must be passed a list of actual parameter values that are the same or assignment compatible with the types of parameters that the method is declared to have.

from	*to*
byte	short
byte	int
byte	long
byte	float
byte	double
short	int
short	long
short	float
short	double
char	int
char	long
char	float
char	double
int	long
int	float
int	double
long	float
long	double
float	double

Table 3: The automatic promotions that are possible because they do not lose information.