MASL Language Reference Manual

COMSW4115 Programming Languages & Translators

Jiatian Li	Wei Wang	Chong Zhang	Dale Zhao
jl3930	ww2315	cz2276	dz2242

Contents

1 Overview

This document serves as a formal description of the Multi-Agent Simulation Language, or MASL. The lexicon, grammar and semantics of the core language are elaborated in this reference. However, this document will not provide much information on the runtime infrastructure and standard libraries for MASL. These topics will appear in other related documents.

The chapters of this document come as follows.

Chapter 2 discusses the lexical conventions of MASL for identifiers, keywords and comments. Chapter 3 introduces the data types of MASL. Chapter 3.1 is about the expressions and operators. Chapter 4 and Chapter 5 focus on functions and objects in MASL respectively, both of which in fact share a lot of features in syntax and behavior as basic data types. Chapter 7 discusses the control flow facilities of MASL, and classifies different types of statements and discusses what constitutes a MASL program. Chapter 8 provides a formal definition for the syntax of MASL using context-free grammar.

1.1 Conventions

In this text, we will use fixed-width font for MASL code, such as:

```
int year = 2012;
```

And a serif font type different from the text for production rules:

control-flow-statement:

if (expression **)** statement
if (expression **)** statement **else** statement
for (expression$_{opt}$ **;** expression$_{opt}$ **;** expression$_{opt}$ **)** statement
for (identifier **:** expression **)** statement
while (expression **)** statement
do statement **while (** expression **)**

With terminal symbols in bold type and non-terminal symbols in regular type.

2 Lexical Conventions

This chapter gives some basic knowledge of MASL lexicon. More lexical issues will be discussed in appropriate contexts later.

Currently a MASL source program is written with ASCII only, so the characters mentioned in the following text all refer to those in ASCII.

2.1 Tokens and Whitespaces

A token is a sequence of characters that specify an entity or mark a language construct in MASL. Tokens include identifiers, literals, keywords, operators and separators, each of which will be discussed later.

Whitespaces, including spaces, tabs and newlines, can be used to separate two adjacent tokens. Sometimes such separation is optional, but in other cases whitespaces are mandatory.

2.2 Identifiers

An identifier is used to uniquely name an entity in MASL, such as a variable of some basic type, a function or an object. A legal identifier is a character sequence of one or more letters, digits or underscores, the first of which cannot be a digit. So the following 3 identifiers are legal:

```
month Year  Matrix3x3  _message
```

But the following ones are not:

```
someone@somewhere     9lives
```

MASL is a case-sensitive language. So the following 3 identifiers are mutually different:

```
masl MASL Masl
```

2.3 Keywords

Keywords are tokens with special meanings that should be reserved. A user-defined identifier should not be one of the MASL keywords, otherwise the compiling may end up with errors.

All the keywords in MASL are listed below:

```
boolean break char continue do double else for if int object return
state this while
```

2.4 Comments

Comments are simply treated as whitespaces by the MASL compiler, but may contain information that helps explains the code nearby. MASL supports two kinds of comments: single-line comments and multi-line comments.

A single line comment starts with two slashes (`//`). The two slashes may or may not be the first of the line, but everything that follows until the end of the line will be part of the comment.

A multi-line comment starts with a slash and an asterisk, i.e. `/*`, and the ends at the first `*/` combination. The pair of `/*` and `*/` may or may not be on the same line, and everything in between is part of the comment.

Comments may not be surrounded by quotes (`"`), or they will become part of the string instead of comments.

3 Types and Values

This chapter discusses the data types supported by MASL and the representation of their literals, as well as how to define variables.

3.1 Data Types and Literals

3.1.1 Basic Data Types

In MASL, 5 basic data types are supported, namely integers, characters, doubles, booleans and voids, identified using keywords `int`, `char`, `double`, `boolean` and `void`, respectively.

3.1.1.1 Integers

An integer in MASL is signed and 32-bit long, and its literal is a decimal number consisting of one or more digits, such as:

```
142857
```

3.1.1.2 Characters

A character in MASL is an 8-bit ASCII character. It is written as a single character surrounded with single quotes, e.g. `'s'`, `'0'`, `' '`, etc.

MASL provides a few escape sequences for characters that are not easy to read on the screen or hard to type with a keyboard, including:

`'\n'` New line character

`'\t'` Horizontal table character

`'\0'` Null character

3.1.1.3 Floating Numbers

As for floating point numbers, MASL supports the double precision floating number defined by IEEE 754. A double literal consists of an integer part and a fraction part separated with a decimal point, followed by an optional exponent part, which has a letter `e` or `E` followed by a signed or unsigned integer. The fraction part may be omitted with the presence of the exponent, and the integer part may be omitted with the presence of the fraction part. For instance, the following double literals are valid:

```
3.14
3.14e-10
0.314
.314
3e14
```

3.1.1.4 Booleans

Booleans are used to represent the value of logical truths. There are only 2 literals for the boolean type, i.e. `true` and `false`.

3.1.1.5 Void

The data type void is used to represent "nothing". Sometimes a MASL function does not return a value, and in this case, we say the return type of that function is void, equivalent to saying the function returns nothing.

There is only one literal for void: `void` itself.

3.1.2 Lists

A list is essentially an array of elements of the same type. The literal of a list is written as zero or more elements surrounded with a pair of curly braces, each adjacent two separated with a comma:

```
{1, 2, 3, 4, 5}
```

At runtime, we can read, write or remove any elements of a list, and also add new elements to a list at specified positions. These will be discussed in Section 4.9.

3.1.2.1 Strings

A string in MASL is essentially a list of characters. MASL provides a more convenient way to write a string literal. That is to write a sequence of characters and surround them with a pair of double quotes. For example:

```
"Goodbye, cruel world."
```

3.1.3 Functions and Objects

Functions and objects are two other important data types of MASL. Since their features are much more complex than these basic data types, they will be elaborated in Chapter 5 and 6 respectively.

3.2 Variables

Generally, a variable is a named memory block containing a value of some data type. For instance, the following two statements define an integer variable and a double variable respectively:

```
int x = 2;
double y = 3.14e11;
```

And the following statement defines an integer list which contains 4 integers:

```
int[] list = {1, 2, 3, 4};
```

As a string is just a list of characters, the following code defines a string:

```
char[] str = "This is a string.";
```

If a variable is declared but not initialized, however, that variable will hold the default value of its type. The default values for `int`, `char`, `double` and `boolean` are `0`, `'\0'`, `0.0` and `false`, respectively. The default value for a list of type `T` is `[]`.

3.3 Type System

MASL will enforce strong and static typing rules on basic data types, functions and lists. That is, the type check is done at compile time, and there are a lot of restrictions on intermixing operations between different types of data to prevent runtime errors.

However, MASL deals with objects in a duck typing manner, which will be discussed in detail in Chapter 6.

4 Expressions

This chapter classifies all forms of expressions in MASL, and gives a formal description to each of them.

4.1 Primary Expressions

Primary expressions can be identifiers, literals or expressions in parentheses.

primary-expression:
identifier
literal
(expression **)**

4.1.1 Identifier

An identifier is a primary expression provided it has been properly declared. The type of an identifier is determined by its declaration.

4.1.2 Literal

The type of a literal primary expression specifies the constant of some data type.

4.1.3 Parenthesized Expression

A parenthesized expression is a primary expression whose type and value are identical to the enclosed expression.

4.2 Postfix Expressions

Postfix expressions contain operators grouping from left to right.

postfix-expression:
    primary-expression
    postfix-expression **[** expression **]**
    postfix-expression **(** argument-expression-list$_{opt}$ **)**
    postfix-expression **. identifier**
    postfix-expression **++**
    postfix-expression **--**

argument-expression-list:
    expression
    expression**,** argument-expression-list

4.2.1 List Reference

A postfix expression followed by an expression in square brackets is a postfix expression denoting a subscripted list reference, which will be introduced in Section 4.9.1.1.

4.2.2 Function Calls

A function call is a postfix expression, consisting of a function name followed by parentheses containing a possibly empty, comma-separated list of argument expressions, which constitute the arguments passed to the function.

4.2.3 Member References

A postfix expression followed by a dot followed by an identifier is an expression that accesses the member of an object (See Section 6.2).

4.3 Unary Operators

Expressions with unary operators group from right to left.

unary-expression:
    postfix-expression
    ++ unary-expression
    -- unary-expression

unary-operator cast-expression

unary-operator: one of
+ - !

4.3.1 Prefix Incrementation Operators

A unary expression proceeded by a ++ or −− operator is a unary expression. The operand is incremented or decremented by 1 respectively. The value of the expression is the value after the increment or decrement.

4.3.2 Unary Plus Operator

The operand of the unary + operator must have a numeric type, and the result is the value of the operand.

4.3.3 Unary Minus Operator

The operand of the unary − operator must have a numeric type, and the result is the value of the operand.

4.3.4 Logical Negation Operator

The operand of the `!` operator must have `boolean` type and the result is the negation of operand.

4.4 Algorithmic Operators

The algorithmic operators can be divided into two categories. The first category includes multiplicative operators while the second includes additive operators.

multiplicative-expression:
multiplicative-expression * cast-expression
multiplicative-expression / cast-expression
multiplicative-expression % cast-expression

additive-expression:
multiplicative-expression
additive-expression + multiplicative-expression
additive-expression - multiplicative-expression

4.4.1 Multiplicative operators

The multiplicative operators `*`, `/` and `%` group from left to right.

The operands of `*` and `/` must have numeric type. The operands of `%` must have integral type. The binary `*` operator denotes multiplication. The binary `/` operator yields the quotient, and the `%` operator the remainder of the division of the first operand by the second; if the second operand is 0, the result is undefined. Otherwise, it is always true that `(a / b) * b + a % b` is equal to `a`. If both operands are non-negative, then the remainder is non-

negative and smaller than the divisor; otherwise, it is guaranteed only that the absolute value of the remainder is smaller than the absolute value of the divisor.

4.4.2 Additive Operators

The additive operators + and − group from left to right. The result of the + operator is the sum of the operands, and the result of the − operator is the difference of the operands.

4.5 Relational Operators

relational-expression:
    additive-expression
    relational-expression **<** additive-expression
    relational-expression **>** additive-expression
    relational-expression **<=** additive-expression
    relational-expression **>=** additive-expression

The relational operators group left-to-right, but this fact is not useful. The operators < (less than), > (greater than), <= (less or equal), >= (greater or equal) all yield `false` if the specified relation is false and `true` if it is true.

4.6 Equality Operators

equality-expression
    relational-expression
    equality-expression **==** relational-expression
    equality-expression **!=** relational-expression

The == (equal to) and `!=` (not equal to) operators are analogous to relational operators except for their lower precedence.

4.7 Logical Operators

logical-AND-expression:
    equality-expression
    logical-AND-expression **&&** equality-expression

logical-OR-expression:
    equality-expression
    logical-OR-expression **||** equality-expression

4.7.1 Logical AND Operator

The `&&` operator groups left-to-right. It returns `true` if both its operands are `true`, `false` otherwise. This operator performs short-circuit calculation.

4.7.2 Logical OR Operator

The `||` operator groups left-to-right. It returns `true` if either of its operands is `true`, and `false` otherwise. This operator performs short-circuit calculation.

4.8 Assignment Expression

assignment-expression:
    unary-expression = assignment-expression

In an assignment expression, the value of the right-side expression replaces that of what is referred to by the expression on the left.

4.9 List Operations

4.9.1.1 List References

A postfix expression followed by an expression in square brackets denotes a subscripted list reference. The first expression must have the type a list of `T`, where `T` is some type, and the other expression must be of `int` type or turn out to be a list of `int`.

The index is 0 based, which means `list[0]` returns the reference to the first element in `list`.

Some examples of getting list elements are given below:

```
int[] list = {10, 11, 12, 13, 14, 15};
list[1];   // Returns 11.
list[{0, 1, 2}]; // Returns a sublist {10, 11, 12}.
```

4.9.1.2 Range Generation

The range generation expression can be used to generate a range of numbers conveniently. It has the form:

expression:
    { $\text{expression}_{\text{start}}$ **:** $\text{expression}_{\text{step}}$ **:** $\text{expression}_{\text{end}}$ **}**
    { $\text{expression}_{\text{start}}$ **:** $\text{expression}_{\text{end}}$ **}**

$\text{expression}_{\text{start}}$, $\text{expression}_{\text{step}}$ and $\text{expression}_{\text{end}}$ can be of `int` or `double` type. If $\text{expression}_{\text{step}}$ is not specified, the step of this range will be 1 by default. This will generate a list of numbers (`int` or `double`) starting from $\text{expression}_{\text{start}}$, with each element produced by adding $\text{expression}_{\text{step}}$ to the previous one. If $\text{expression}_{\text{step}}$ is positive, then the last element is no greater than $\text{expression}_{\text{end}}$. If it is negative, then the last element is no less than $\text{expression}_{\text{end}}$.

If $\text{expresion}_{\text{step}}$ turns out to be 0, only $\text{expresion}_{\text{start}}$ and $\text{expression}_{\text{end}}$ will be included in the list generated.

Here are some examples of range generation:

```
int[] list1 = [0 : 3];      // {0, 1, 2, 3}
double[] list2 = [1.0 : 2.2 : 7.0];   // {1.0, 3.2, 5.4}
```

4.9.1.3 Concatenation

Two lists can be concatenated with the operator ^. A concatenation operator returns a new list which is a concatenation of the given two lists in the order they appear in the expression.

expression:
    expression ^ expression

4.10 Miscellaneous Operators

There are still some operators not covered in this chapter. Below are some of them, which will be discussed in detail in the following text.

4.10.1.1 Dot operator

A dot operator (.) is used to access a member of an object.

expression:
    expression . identifier

For example:

```
someObject.someMember;
```

accesses the member `someMember` of object `someObject`.

4.11 Precedence and Associativity

The following table lists operators with their associativity in MASL, in the order of descending precedence from top to bottom.

Operator	Associativity
(expr) [index] . ->	left to right
! ++ -- unary operator: + -	right to left
* / %	left to right
+ -	left to right
< <= > >=	left to right
== !=	left to right
&&	left to right
| |	left to right
+= -= *= /= %= =	right to left

5 Functions

5.1 Defining a Function

Much like defining a variable of some basic data type, a function can be defined by assigning a function literal to a function variable. For instance:

```
(double, double): double average =
   (double a, double b):double { return (a + b) / 2.0; }
```

in which `average` is the name for the function and `(double, double):double` is the type of the function. The whole thing on the right side of = is a function literal:

```
(double a, double b):double { return (a + b) / 2.0; },
```

which specifies a list of parameters followed by the return type of the function separated by a colon, and then a compound statement which will be executed when the function is invoked.

To make programmers of C family languages more comfortable, we introduced a syntax sugar for function definitions as illustrated below:

```
double average(double a, double b) { return (a + b) / 2.0; }
```

which is equivalent to the previous style of function definitions.

In conclusion, function definitions have the form:

function-declaration:
    function-type-specifier **identifier =** function-literal
    type-specifier **identifier (** parameter-list **)** compound-statement

where:

parameter-list:
    parameter **,** parameter-list
    parameter

function-type-specifier:
    (parameter-type-list **) :** type-specifier

parameter-type-list:
    type-specifier**,** parameter-type-list
    parameter **,** parameter-type-list
    type-specifier
    parameter

function-literal:
(parameter-list **) :** type-specifier compound-statement

parameter:
type-specifier **identifier**

The default value of a function is a function that simply returns the default value of the return data type of that function.

5.2 Invoking a Function

To invoke a function, apply a pair of parentheses, which is considered an operator, to a function and a list of arguments passed to it.

An example of function invocation is shown below:

```
average(1.0, 2.0);
```

5.2.1 By-value vs. By-Reference

When passing basic type arguments to a function, what the function access is just a copy of the arguments passed in. Thus, modification to the arguments inside the function does not affect the original data.

When passing an argument which is a list, a function or an object, however, what is actually passed is the copy of the reference to the original data. Thus the code inside a function is able to modify the value of such an argument, but it cannot modify the original reference itself, since what it accesses is merely a copy of that reference.

5.3 Functions as First Class Objects

Functions are first-class objects in MASL. That is, they can be passed as arguments to other functions or be returned by other functions, as well as assigned to some function variable. Thus the following MASL code snippet is allowed:

```
(double):double times(double scale) {
   return (double number):double { return number * scale; };
}

(double):double twice = times(2);

double processNumber(double number, (double):double functor) {
   return functor(number);
}
```

Since a MASL functions cannot be modified once it is defined, even though a copy of its reference is passed as an argument, the function passed cannot be modified by the code in the function it is passed to.

6 Objects

In MASL, an object is an entity that encapsulates a set of attributes, behaviors and states, and relates them together. An object may have different behaviors under different states, and these behaviors may in turn access the attributes of that object or change its state. In essence, an object in MASL is a Definitive State Automaton (DFA).

6.1 Object Definition

An object in MASL can be defined like this:

```
object objectName = {
   …
};
```

The statement block surrounded by the curly braces is the body of the object, which consists of the member variables, member functions and states of that object.

6.2 Member Variables

Defining a member variable uses almost the same syntax as defining a variable, except that member variable declaration should only appear in an object body:

```
object A = { int number; };
```

To access the variable `member` in object `A`, we write:

```
A.number
```

Or

```
A.number = 1
```

A member variable is a left value. This means that we can read or overwrite the value of a member variable.

A member variable can be initialized when you define it, like:

```
object A = { int number = 10; };
```

If a member variable is not initialized, it will be set to the default value for its data type. For example, the default value for an integer would be 0.

The default value of an object is `{}`, indicating an object with no members.

6.3 Member Functions

Since the function is also considered a data type in MASL, the way we define a member function is much like the way we define a member variable of other types.

```
Object A = {
```

```
   int number = 5;
   int timesNumber (int n) {
      return number * n;
   }
};
```

As we can see from the code above, a member function has access to the member variables defined in the same objects. To invoke that member function, we write:

```
A.timesNumber(6)
```

Actually, the notation above is just a syntactic sugar for:

```
timesNumber(A, 6)
```

In this way, `timesNumber` knows that the variable number is in the object `A`, i.e. `A.n`. In `timesNumber` we can also write number as `this.number`, with `this` pointing to the hidden leading parameter which should be the object the function is called on, that is `A` in this case.

6.4 States

Every object in MASL has a built-in automaton. An object may behave differently in different states, and may transfer from one state to another under some conditions. An object may have one state as its current state. During a simulation step, for each object in the container for the simulation, the code in its current state gets executed. If an object does not have a current state, however, all its parts will be defined, but it will not perform any actions in subsequent simulation process.

Here is a sample that defines several states in an object.

```
object Warrior = {

   state watch { if(enemyInSight()) -> attack; }
   state attack { fight(); if(!enemyInSight()) -> watch; }

   boolean enemyInSight() { … }
   void fight() { … }
};
```

-> is called the state transition operator, which sets the current state of the object to the one on the right side.

You may retrieve the name of the current state of an object using its built-in variable member `state`, which is a string. It is automatically updated every time the current state of the object changes.

The current states can be set upon object creation. Thus we can write:

```
object Warrior = {

   state watch { if(liveEnemyInSight()) -> attack; }
   state attack { fight(); if(!liveEnemyInSight()) -> watch; }

   boolean liveEnemyInSight() { … }
   void fight() { /* Fight with a sword. */ … }
   int health = 100;
   int level = 1;
   int x;
   int y;
} -> watch;
```

The object warrior will go into `watch` state after creation.

In fact, any statements can be written inside an object body, and these statements will be executed once when the object is initially created.

6.5 Duplication

To build a new object based on an existing one, we can use object duplication in MASL like this:

```
object Warrior1 = Warrior { x = 100; y = 200; };
object Warrior2 = Warrior { x = 50; y = 300; level = 4; };
```

Everything in `Warrior` is copied into `Warrior1` and `Warrior2`, and during the initialization of `Warrior1` and `Warrior2`, the member variables `x`, `y` and `level` are set to some other value.

MASL does not support class-based inheritance. But since an object can be built based on another by copying all its members, their member functions can be overwritten. The sample below illustrates this.

```
object SuperWarrior = Warrior {
   void fight() { /* Fight with a laser gun. */ … }
};
```

In this case, everything in `Warrior`, including states and members, is copied into `SuperWarrior`, while the member function `fight` in `Warrior` is replaced with a new one.

New members can be added to enhance existing objects as well. For example:

```
object Point = { double x; double y; };
object ColorPoint = Point { string color; }
```

6.6 Access Control

In MASL, all members within an object can be accessed from both inside and outside that object. That is, all object members have a public access level.

6.7 List Functions

Lists in MASL are objects too. A list has some members that provide useful information or operations on itself. Suppose `list` is a list whose elements are of type `T`. Then:

> `list.length` is a member variable that stores the number of elements in the list. It will be updated automatically as the number of elements in the list changes. Setting it will have no effect.
>
> `list.filter(f)` returns a sublist of list that only contains elements which meet a criterion defined using the parameter `f`. `f` is a function of type `(T):boolean`. The element being judged will be passed to it as an argument and it will return `true` when that element meets the criterion, otherwise returns `false`.

6.8 Duck Typing

As we can see from the samples of `SuperWarrior` and `ColorPoint`, members can be added to objects at runtime. This means that there is no enforcement on what type an object should be. Since MASL is a prototype-based language, it does not clearly differentiate object types – it uses duck typing instead. That is, if a member referred to on an object exists, the value of that member is simply retrieved. If it does not exist, an error will be signaled. This principle holds for both non-function members and member functions. In this way, an object variable can be assigned with any object, and any object can be passed to a parameter of `object` type. But there is a risk of mistyping, which the users should be take care of.

7 Statements

7.1 Types of Statements

A statement is a basic execution unit in MASL. In general, statements are executed in the order they are written in the programs. There are several types of statements.

```
statement:
    declaration-statement
    expression-statement
    compound-statement
    control-flow-statement
    jump-statement
```

7.1.1 Declaration Statement

Declaration statements are related to the declaration of variables. There are three kinds of declaration statements:

declaration-statement:
    basic-type-declaration **;**
    function-declaration **;**
    object-declaration **;**

basic-type-declaration:
    basic-type-specifier init-declarator-list

init-declarator-list:
    init-declarator
    init-declarator **,** init-declarator-list

init-declarator:
    identifier
    identifier = expression

function-declaration:
    type-specifier **identifier** = literal
    type-specifier **identifier (** parameter-list **)** compound-statement

object-declaration:
    object identifier = object-literal

7.1.2 Expression Statement

Expression statements are the most frequently used statements, most of which are assignments or function calls.

expression-statement:
    expression **;**
    ;

7.1.3 Compound Statement

In some situations, a block of statements need to be treated as a whole. Such a statement block is called a compound statement.

compound-statement:
    { statement-list **}**

statement-list:
    statement
    statement statement-list

7.1.4 Control Flow Statements

Control flow statements make the execution of statements depend on some conditions. Such statements includes `if`, `for` and `while` statements.

control-flow-statement:
if (expression **)** statement
if (expression **)** statement else statement
for (expression$_{\text{opt-init}}$ **;** expression$_{\text{opt-condition}}$ **;** expression$_{\text{opt-update}}$ **)** statement
for (type-specifier identifier **:** expression **)** statement
while (expression **)** statement
do compound-statement **while (** expression **)**

Specifically, in the statement:

for (expression$_{\text{opt-init}}$ **;** expression$_{\text{opt-condition}}$ **;** expression$_{\text{opt-update}}$ **)** statement

Any or all of the three expressions may be omitted. And if expression$_{\text{opt-condition}}$ is omitted, the condition for iteration will be forever `true`, creating an infinite loop.

In the other form of `for` loop:

for (type-specifier identifier **:** expression **)** statement

The expression should turn out to be an iterable object, i.e. an object that can return an iterator, such as a list. And in every iteration of the loop, a variable named with the identifier will hold the value of the element pointed to by the iterator of expression.

7.1.5 Jump Statement

Jump statements in MASL can transfer the control flow instantly to another location.

jump-statement:
continue ;
break ;
return expression$_{\text{opt}}$;

A `continue` statement may appear only within a loop. It causes control to pass to the loop-continuation portion of the smallest enclosing loop.

A `break` statement may appear only in a loop, and it will terminate the execution of the smallest enclosing loop statement. Control will pass to the statement following the terminated loop statement.

A function returns to its caller by a `return` statement. When return is followed by an expression, the value is returned to the caller of the function. The expression is converted to the type of the return value of the function it appears in if the implicit conversion is viable.

7.2 Structure of a MASL Source File

In the top level of a MASL source file, the declaration statements, the expression statements, the control flow statements and compound statements may appear in any order, as long as the variables in a statement are still within its scope (see Section 7.3). These statements will be executed in sequence at runtime.

```
program:
    statement-list
```

7.3 Scope

7.3.1 Lexical Scoping

MASL supports lexical scoping for variable bindings. Thus, the scope of a variable is effective from the end of its declaration statement till the end of the current block i.e. the inner-most component statement it is defined in.

If a variable is defined in the head of a block, such as the loop-continuation portion of a loop, or the parameter list of a function, then the variable is accessible in the entire body block.

Besides, code in a block is able to access the variables defined in an outer enclosing block. It is not true in reverse, however. This holds for a state versus an object definition, a function and the top level of a MASL source file, etc.

7.3.2 Object Member Accessibility

While conforming to lexical scoping rules, object member accessibility is also determined by the way it is created. In Section 6.5 we said that when an object is built based on another, all the members of the latter one will be copied into the former one. Thus, when trying to access a member of an object, MASL will check all the members defined specifically in that object as well as the base object it is copied from.

There is no inheritance in MASL, and one member name in an object must correspond to at most one member. If an attempt is made to write to a member with the same name as a member from the base object, that member is simply overwritten instead of being hidden.

7.3.3 Global Object

In fact, the top level code of an MASL source file is part of the definition of the global object, which is implicit to the user. There is only one global object per MASL program. Thus, any variables defined outside all objects and functions become members of the global variable.

The statements in the top level is outside any state of the global object, and are thus executed only once before the global enters a state. None of the global object states will be exposed to the user. They are used by the underlying simulation engine, which is not addressed in this language reference manual.

8 Grammar

program:
    statement-list

statement:
    declaration-statement
    expression-statement
    compound-statement
    control-flow-statement
    jump-statement

declaration-statement:
    basic-type-declaration **;**
    function-declaration **;**
    object-declaration **;**

basic-type-declaration:
    basic-type-specifier init-declarator-list

init-declarator-list:
    init-declarator
    init-declarator **,** init-declarator-list

init-declarator:
    identifier
    identifier = expression

function-declaration:
    function-type-specifier **identifier =** function-literal
    type-specifier **identifier (** parameter-list **)** compound-statement

parameter-list:
    parameter **,** parameter-list
    parameter

function-type-specifier:
    (parameter-type-list **) :** type-specifier

parameter-type-list:
    type-specifier**,** parameter-type-list
    parameter **,** parameter-type-list
    type-specifier
    parameter

function-literal:

(parameter-list **) :** type-specifier compound-statement

parameter:
type-specifier **identifier**

object-declaration:
object identifier = object-literal

object-literal:
object-name object-body
object-body

object-body:
{ state-declaration-list statement **}**

state-declaration-list:
state-declaration
state-declaration state-declaration-list

state-declaration:
state identifier compound-statement

expression-statement:
expression **;**
;

compound-statement:
{ statement-list **}**

statement-list:
statement
statement statement-list

control-flow-statement:
if (expression **)** statement
if (expression **)** statement else statement
for ($\text{expression}_{\text{opt-init}}$ **;** $\text{expression}_{\text{opt-condition}}$ **;** $\text{expression}_{\text{opt-update}}$ **)** statement
for (type-specifier identifier **:** expression **)** statement
while (expression **)** statement
do compound-statement **while (** expression **)**

jump-statement:
continue ;
break ;
return $\text{expression}_{\text{opt}}$;

primary-expression:
    identifier
    literal
    (expression **)**

postfix-expression:
    primary-expression
    postfix-expression **[** expression **]**
    postfix-expression **(** argument-expression-list$_{opt}$ **)**
    postfix-expression **. identifier**
    postfix-expression **++**
    postfix-expression **--**

argument-expression-list:
    expression
    expression**,** argument-expression-list

unary-expression:
    postfix-expression
    ++ unary-expression
    -- unary-expression
    unary-operator cast-expression

unary-operator: one of
    + - !

multiplicative-expression:
    multiplicative-expression ***** cast-expression
    multiplicative-expression **/** cast-expression
    multiplicative-expression **%** cast-expression

additive-expression:
    multiplicative-expression
    additive-expression **+** multiplicative-expression
    additive-expression **-** multiplicative-expression

relational-expression:
    additive-expression
    relational-expression **<** additive-expression
    relational-expression **>** additive-expression
    relational-expression **<=** additive-expression
    relational-expression **>=** additive-expression

equality-expression
    relational-expression

equality-expression == relational-expression
equality-expression != relational-expression

logical-AND-expression:
equality-expression
logical-AND-expression && equality-expression

logical-OR-expression:
equality-expression
logical-OR-expression || equality-expression

assignment-expression:
logical-OR-expression
unary-expression = assignment-expression