

aML

Language Reference Manual

Sriramkumar Balasubramanian (**sb3457**)

Evan Drewry (**ewd2106**)

Timothy Giel (**tkg2104**)

Nikhil Helferty (**nh2407**)

October 29, 2012

Contents

1	Introduction	4
2	Lexical Conventions	4
2.1	Comments	4
2.2	Identifiers	4
2.3	Keywords	4
2.4	Literals	5
2.4.1	Integer Literals	5
2.4.2	Boolean Literals	5
2.4.3	Null	5
2.4.4	List Literals	5
2.5	Separators	5
3	Expressions	5
3.1	Primary Expressions	6
3.2	Operators	6
3.2.1	Arithmetic Operators	6
3.2.2	Relational Operators	6
3.2.3	Boolean Operators	7
3.2.4	Assignment Operators	8
3.2.5	Associative Operator	8
4	Declarations	8
4.1	Variable Declarations	8
4.2	Variable Initialization	8
4.3	Function Declaration	9
5	Statements	9
5.1	Expression statement	10
5.2	Compound statements	10
5.3	Conditional statements	10
5.4	Return statement	10
6	Scope rules	11
7	Preprocessor directives	11
8	Implicit variables and functions	11
8.1	Variables	11
8.2	Functions	11

9	Types revisited	12
9.1	List<datatype>	12
9.2	Cell	13
9.2.1	Neighborhood functions	13
9.2.2	Cell functions	13
10	Syntax summary	13
10.1	Expressions	14
10.2	Statements	15
10.3	Program Definition	16

1 Introduction

This manual describes the aML language which is used for manipulating mazes and is used to provide instructions to a bot traversing the maze.

The manual provides a reliable guide to using the language. While it is not the definitive standard, it is meant to be a good interpretation of how to use the language. This manual follows the general outline of the reference manual referred to in “The C Programming Language”, but is organized slightly differently with definitions specific to aML. The grammar in this manual is the standard for this language.

2 Lexical Conventions

A program consists of a single translation unit stored as a file. There are five classes of tokens: **identifiers**, **keywords**, **constants**, **operators**, and other separators. White space (blanks, tabs, newlines, form feeds, etc.) and comments are ignored except as they separate tokens. Some white space is required to separate adjacent identifiers, keywords, and constants.

2.1 Comments

The characters `//` introduces a single line comment. The rest of the line is commented in this case. This differs from a multi-line comment which is enclosed by the `/*` and `*/` characters. Multi-line comments do not nest.

2.2 Identifiers

An identifier is a sequence of letters and digits, beginning with a letter and can be of any length. The case of the letters is relevant in this case. No other characters can form identifiers.

eg. `abcd`, `Abcd`, `A123`, `abc1`

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:-

<code>if</code>	<code>return</code>	<code>display</code>	<code>remove</code>	<code>left</code>	<code>hasleft</code>	<code>Integer</code>
<code>then</code>	<code>main</code>	<code>print</code>	<code>add</code>	<code>right</code>	<code>hasright</code>	<code>Boolean</code>
<code>else</code>	<code>void</code>	<code>source</code>	<code>head</code>	<code>up</code>	<code>hasstop</code>	<code>List<datatype></code>
<code>load</code>	<code>function</code>	<code>visited</code>	<code>next</code>	<code>down</code>	<code>hasbottom</code>	<code>Cell</code>
<code>random</code>	<code>exit</code>	<code>isTarget</code>	<code>isEmpty</code>	<code>CPos</code>		
<code>True</code>	<code>null</code>	<code>NOT</code>	<code>AND</code>			
<code>False</code>			<code>OR</code>			

This language consists of many implicit variables and functions increasing the size of the reserved words list.

2.4 Literals

There are different kinds of literals (or constants) in aML as listed below:-

2.4.1 Integer Literals

An integer literal is taken to be decimal, and is of data type Integer. It may consist only of a sequence of digits 0-9.

eg. 0,1,22,-5

2.4.2 Boolean Literals

A boolean literal is either **True** or **False**, and is of data type Boolean.

2.4.3 Null

The null literal has no data type and is simply the word **null**. This is linked to the Cell and List<datatype> datatypes which need to be initialized to some value after declaration.

2.4.4 List Literals

The list literal can include either the Integer, Boolean, Cell or List<datatype> types (cascaded lists).

eg. <[1]>,<[1,2,3]>,<[[1,2,3],[4,5]]>,<[True, False, True]>

The Cell datatype does not have any literals associated with it. It can only be assigned to the CPos variable which will be discussed in section 8.

2.5 Separators

The semi-colon ; and the pair of braces { }, the < > and [], act as separators of the tokens. They are meant to reduce ambiguity and conflicts during the parsing phase. The semi-colon is added at the end of every statement to signify the end. The { } are used to collect groups of statements into a single compound statement block. The < > and [] are used to instantiate the List<datatype> variables.

3 Expressions

The complete syntax is provided in section 10. This section introduces the definition of the expression types which are the basic building blocks of any language.

3.1 Primary Expressions

Primary expressions are identifiers, constants, or expressions in parentheses. They also include the variable CPos which will be explained in section 9.

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

A parenthesized expression is equivalent to the expression without the parentheses. The parentheses is used to group terms especially to enforce precedence levels. CPos is the implicit variable denoting the current position of the bot on the maze. It is mostly used to assign to variables of Cell datatype.

3.2 Operators

3.2.1 Arithmetic Operators

There are six arithmetic operators: { +, -, *, /, %, ^ }. The operands of these operators must be of Integer data type. The result will also be of type Integer.

arithmetic-expression:-
 expression + expression
 expression - expression
 expression * expression
 expression / expression
 expression % expression
 expression ^ expression

Operator	Semantic	Comments
+	addition	
-	subtraction	
*	multiplication	
/	division	Integer division only. Divide by zero => error
%	modulo	
^	exponentiation	

3.2.2 Relational Operators

The relational operators all return values of Boolean type (either True or False). There are six relational operators: { ==, ~=, >, <, >=, <= }. The operators all yield **False** if the specified relation is false and **True** if it is true.

relational-expression:-

expression == expression
expression ~= expression
expression > expression
expression < expression
expression >= expression
expression <= expression

Operator	Semantic
==	equals
~=	not equals
>	greater
<	lesser
>=	greater than equals
<=	less than equals

The == operator compares the value of left expression to the right expression and evaluates to True if they are equal, False otherwise. It is vice-versa for the ~= operator. The > operator evaluates to true if the left expression is greater than the right expression, false otherwise. The < operator behaves in the opposite manner. The >= and <= operators check for equality condition as well.

For the == and ~= operators, the expressions involved must be of the same datatype. The other operators are defined only for the Integer datatype where comparison is meaningful.

3.2.3 Boolean Operators

The boolean operators all return values of Boolean type (either True or False). There are three boolean operators: logical-NOT, logical-AND and logical-OR, denoted by NOT, AND, and OR, respectively.

Not-expression:-

NOT expression

And-expression:-

expression AND expression

Or-expression:-

expression OR expression

The operand(s) to NOT, AND and OR have to evaluate to True or False, or in other words, they must either be Boolean variables or relational expressions. NOT negates the operand, AND returns True if all operands evaluate to true, False otherwise. OR returns True if at least one of the operands evaluate to true, False otherwise.

3.2.4 Assignment Operators

There is a single assignment operator in aML, $:=$, which does simple assignment. It is a binary operator and its usage is described in section 4.2.

3.2.5 Associative Operator

The `.` operator is used for function calls on variables represented by identifiers. The structure of statements involving the operator is shown in section 10.

4 Declarations

Declarations specify the interpretation given to each identifier i.e. the type of data it can point to and the associated operations that go along with it.

4.1 Variable Declarations

The data types present are defined as follows:-

datatype:-

Integer

Boolean

Cell

List<datatype>

The rule representing the declaration of variables is listed in the complete Syntax summary in section 10. The declaration of variables is similar to many strongly typed languages where the type of the variable must be specified during declaration. Examples of some variable declarations:-

- Integer x;
- Boolean flag;
- Cell node;
- List<Integer> myList;

Detailed explanation of the List<datatype> and Cell datatypes are provided in section 9.

4.2 Variable Initialization

When a variable is declared, an initial value may also be specified for the identifier being declared.

Roughly it is of the form - **lvalue** $:=$ **rvalue**;

The $:=$ operator is used to set the value of lvalue to the value of rvalue.

- `x := 10;`
- `flag := false;`
- `node := null;`
- `myList := <[1,2,3]>;`

The exact rule is provided in the Syntax summary in section 10. Initialization can also be combined with declaration in a single step. This is also shown in final section.

4.3 Function Declaration

Functions can either return a certain datatype or be void functions (return no value). A function header is specified with the **function** keyword and an identifier along with an optional argument list and return type. Functions can be “used” by function calls. But for a function to be called, it must be declared in the program.

function_declaration:-

function_header { body }

function_header:-

function identifier (args_list_{opt}) : ret_type

args_list:-

datatype identifier

datatype identifier,args_list

body:-

compound-statement

Function calls are handled in section 10. Compound statements are described in detail in the section below.

Note: The subscript ‘opt’ stands for optional argument indicating the term formed by omitting the indicated term is also possible in the grammar.

5 Statements

Statements are usually executed in sequence, with the exception of conditional statements. They are the next level of basic building blocks after expressions. Each statement ends with a semi-colon at the end which denotes the end of the logical statement. The physical statement which is equivalent to one line in the editor may be comprised of one or more logical statements.

One notable feature in aML is the lack of looping constructs. Iterations are

achieved by tail recursion of functions. The function definition shown above is represented in the bigger picture in section 10.3. The following definition gives an idea about the components of a statement. The entire definition integrated with other definitions is present in section 10.

5.1 Expression statement

expression-statement:-

expression;

Expression statement consist of assignments and function calls.

5.2 Compound statements

Compound statements are provided in the form:-

compound-statement:-

{ statement-list }

statement-list:-

statement

statement statement-list

Compound statements are generally used to form the body of code to execute in conditional statements, as well as the body of function definitions.

5.3 Conditional statements

Conditional statements have the general form:-

conditional-statement:-

if (expression) then {compound-statement};

if (expression) then {compound-statement} else {compound statement}

The else branch is optional. The program will evaluate the expression in parentheses, and if it evaluates to the Boolean value true then it executes the corresponding compound-statement, and subsequently continues on to the statement following the conditional statement. If the expression does not evaluate to true, then the compound-statement following the else branch is executed (if it exists). Branches are evaluated in order, such that only the first branch with an expression that evaluates to true will be executed, and all others skipped.

5.4 Return statement

Return statement Return statements take the form:-

return-statement:-

return expression;

The expression should evaluate to a value of the return type of the function being defined.

6 Scope rules

Programs are not multi-file in AML, so external scope is not a worry. The lexical scope of identifiers is of relevance however. In brief, subsequent to declaration a given identifier is valid for the rest of the function inside which it was declared. Re-declarations using an already declared identifier will overwrite the reference to the first identifier. No identifiers can be declared outside functions.

While user-defined variables cannot enjoy a global scope, the implicit variables on the other-hand can do so. More information on implicit variables is provided in 8.

7 Preprocessor directives

Preprocessor directives must precede any code in the program. One possible preprocessor directive takes the form: **#load filename**. This instruction ensures that the maze to be navigated is to be generated from the file. The acceptable file format is pre-defined and is independent of the language used.

Another possible directive is: **#load-random**. This leads to the maze is to be randomly generated each time the program runs.

The two directives are mutually exclusive. In the event of multiple directives, the compiler will show an error.

8 Implicit variables and functions

aML consists of many implicit variables and functions. The variables and functions along with their meaning are provided below:-

8.1 Variables

The implicit variables are as follows.

- CPos - denotes the current position of the bot on the maze. Variables of type Cell can be instantiated by referencing CPos.
- Visited - It is a dictionary like structure which maintains the 'visited' status of each cell of the maze. It is used especially for backtracking algorithms. It can never be used. The Visit() function provided accesses this data structure inherently.

8.2 Functions

The implicit functions mainly deal with the movement and functionalities of the bot.

- `move_U()` - moves the bot one cell up from the current position, returns true if it succeeds, false otherwise
- `move_D()` - moves the bot one cell down from the current position, returns true if it succeeds, false otherwise
- `move_L()` - moves the bot one cell left of the current position, returns true if it succeeds, false otherwise
- `move_R()` - moves the bot one cell right of the current position, returns true if it succeeds, false otherwise
- `revert()` - goes back to the previous position from the current position, returns true if successful, false if at the start
- `Visit(id)` - checks if the cell referred to by id has been visited or not

9 Types revisited

This section discusses the `List<datatype>` datatype and the functions associated with it. These two datatypes are in a sense less primitive than the Integer and Boolean datatypes. They come along with certain functions which can be applied to variables belonging to these datatypes. These functions are invoked or called using the `.` associative operator on the identifier. The rule regarding the functions is shown in the final section.

9.1 List<datatype>

The `List<datatype>` from its definition in section 4.1 allows cascaded lists. This is especially useful for adjacency list representation of graphs from mazes. The functions associated with the datatype allow the manipulation and traversal of the lists.

- `add()` - adds an elements to the end of the current list
eg. `mylist.add(2);`
- `remove()` - removes and returns the first element of the current list
eg. `mylist.remove();`
- `isEmpty()` - returns true if the current list has no elements, false otherwise.
eg. `mylist.isEmpty()`
- `head()` - returns the first element of the current list
eg. `mylist.head();`
- `next()` - returns the tail of the current list. The element next to the first element of the current list is the head of the tail.
eg. `mylist.next();`

9.2 Cell

The Cell datatype is unique in the sense that it cannot be set a user-defined value. At any point of time, a variable of Cell datatype can be assigned only to the CPos value. It can however be stored in a variable which will reflect that CPos value then, even if accessed at a later time.

Certain functions are provided for this datatype which makes querying the cell's content as well as it's neighborhood easier.

9.2.1 Neighborhood functions

- `left()` - returns the left cell of the current cell if it exists and the current cell has been visited
- `hasleft()` - returns True if there is a cell to the left of the current cell
- `right()` - returns the right cell of the current cell if it exists and the current cell has been visited
- `hasright()` - returns True if there is a cell to the right of the current cell
- `up()` - returns the cell located upwards of the current cell if it exists and the current cell has been visited
- `hasTop()` - returns True if there is a cell to the top of the current cell
- `down()` - returns the cell located downwards of the current cell if it exists and the current cell has been visited
- `hasbottom()` - returns True if there is a cell to the bottom of the current cell

9.2.2 Cell functions

- `isTarget()` - returns true if the cell is a target as specified in the maze
- `Source()` - returns true if the cell is the start point of the maze
- `get_Loc()` - returns the Integer ID of the cell

10 Syntax summary

The entire syntax is provided below. This section is intended for the logical understanding of the language structure rather than an exact copy of the language.

10.1 Expressions

The expression includes declaration statements as well.

expression:-

- primary_expression
- lval_expression
- NOT expression
- expression binop expression
- functions

primary-expression:-

- identifier
- literal
- (expression)
- (CPos)

lval_expression:-

- identifier := expression
- datatype identifier := expression
- datatype identifier

datatype:-

- Integer
- Boolean
- Cell
- List<datatype>

binop:-

Operators	Associativity
^	Right
/ * %	Left
> < >= <=	Left
== ~=	Left
NOT	Right
AND	Left
OR	Left
:=	Right

The binop table shows the binary operators in the decreasing order of precedence (top - bottom) along with their associativity which gives the fashion in which they are grouped together.

functions:-

- list_functions

cell_functions
maze_functions
lang_functions

list_functions:-
 identifier.add(expr)
 identifier.remove()
 identifier.isEmpty()
 identifier.head()
 identifier.next()

cell_functions:-
 identifier.left()
 identifier.right()
 identifier.up()
 identifier.down()
 identifier.hasleft()
 identifier.hasright()
 identifier.hastop()
 identifier.hasbottom()
 identifier.isTarget()

maze_functions:-
 Visit(identifier)
 get_Loc(identifier)
 revert()

lang_functions:-
 identifier(actual_args_{opt})
 display()
 print(identifier)

actual_args:-
 primary_expression
 primary_expression, actual_args

10.2 Statements

Statements are logical sentences that can be formed by the language. A compound statement is a group of statements occurring in a linear fashion one after the other.

compound-statement:-
 {statement-list}

statement-list:-
 statement
 statement statement-list

statement:-
 expression;
 return expr;
 { statement-list }
 if (expression) statement;
 if (expression) statement else statement
 exit;

If the expression to 'if' does not evaluate to True or False, an error will be thrown.

10.3 Program Definition

This subsection describes the structure of the program and functions which are the biggest building blocks in aML. Every aML must have one and only one main function through which the control passes to the program. It must also have exactly one pre-processor directive to load the maze. It can have an arbitrary number of functions though. The program structure is defined below:-

program:-
 empty_program
 pre-process program
 func-def program

empty-program:-

pre-process:-
 #load-identifier
 #load-random

func-def:-
 main():void {statement-list}
 identifier(formal-args_{opt}):return-type{statement-list}

formal-args_{opt}:-
 datatype identifier
 datatype identifier,formal-args

return-type:-
 datatype
 void