aML

a-Mazing Language

Sriramkumar Balasubramanian (**sb3457**)
Evan Drewry (**ewd2106**)
Timothy Giel (**tkg2104**)
Nikhil Helferty (**nh2407**)

December 19, 2012

Contents

1 Introduction

A maze is a puzzle in the form of a series of branching passages through which a solver must a route. Actual mazes have existed since ancient times, serving as a means to confuse the traveler from finding his or her way out. Since then, the idea behind mazes has been extrapolated to construct a set of puzzles designed to challenge people to solve or find the route.

While the concept of maze solving might seem too restricted, maze exploration in general can be extrapolated to other fields like Graph theory and topology. Apart from this, there exist more than one way to solve mazes, which has led to the rise of the time and space analysis of these approaches. Also solving a maze can be likened to exploring a map which paves way for many practical uses of a language for solving mazes.

Having justified the existence of a language to solve mazes, we now introduce AML (A-mazing Language) which can be used to solve mazes by feeding instructions to a bot which is located at the entrance to the maze at time 0. The maze in question can either be defined by the user in the form of text files or can be randomly generated by the standard library functions. AML is designed to not only make the process of solving mazes easier to a programmer, but also to introduce programming to the common man through mazes.

AML's design ensures the freedom of the user to implement many maze solving algorithms, while ensuring the ease of use of the language to traverse mazes. The language serves as an instruction set to the bot, hence the movement of the bot determines accessing of various data.

2 Tutorial

The syntax of aML is similar to a simplified version of Java, or C. The available instructions allow you to move a bot around a maze. You can define your own functions in order to program bots with complex behavior. aML will provide a simple Graphical User Interface of the bot navigating the maze (either randomly generated or provided in a .txt file).

aML has a limited set of data types for variables to take. The most basic types are integer, analogous to the Java int, and bool, like the Java boolean. A third, slightly more complex datatype is cell. This represents a cell in the maze. The programmer can't construct new cells as that would alter the maze, they can only set a cell variable equal to an existing cell of the maze in order to find out information about it. The fourth, and most complex datatype, is List<datatype> (such as List<integer>), which is a First In First Out List that behaves much like a linked list.

Users can define their own functions in aML. A function can either return a datatype (function x():integer) or be void and return nothing. Functions can be recursive (no looping constructs are offered). A special function is main, which must be void and parameterless. main is always the first function in any given program, and is the function that aML will call when the program is run.

2.1 Making Your Own Maze

If users wish to build a custom maze for the bot to navigate, then the maze text file must adhere to a certain format. The text file must be a sequence of integers delimited by whitespace. The first integer is the number of rows in the maze; the second, the number of columns. Then an integer follows for every cell in the maze: either a 0 for a "hole" (unwalkable), a 1 for a walkable cell, a 2 for the starting cell of the bot, and a 3 for a target cell for the bot (note it is possible to have multiple targets). The format can be clearly illustrated by the following example:

5 6

0 1 1 1 0 0
1 1 2 0 1 1
0 0 1 1 1 0
0 1 1 0 1 3
0 3 1 0 1 1

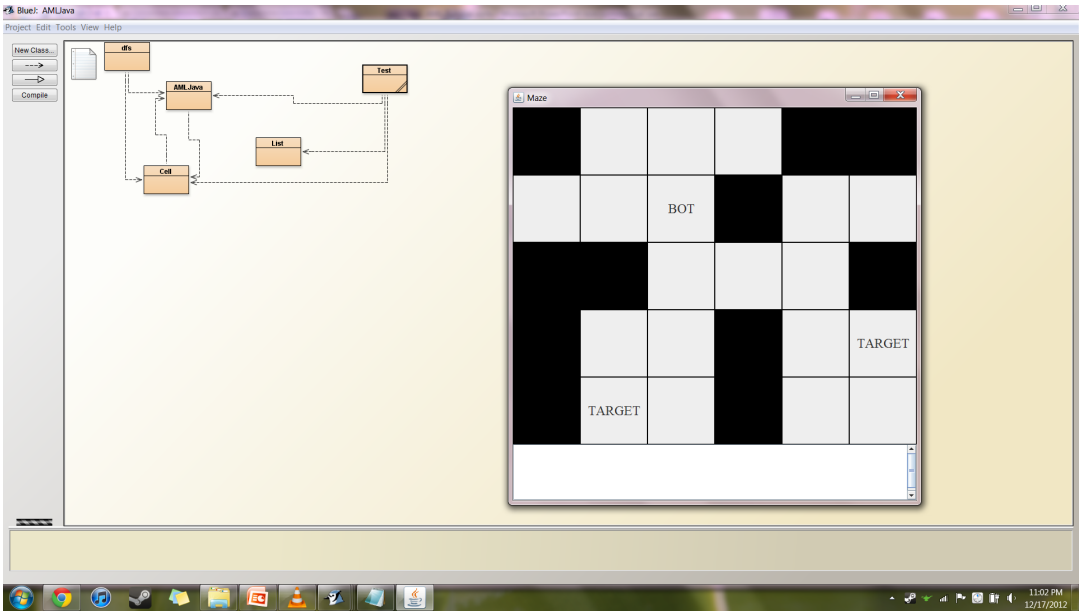

Figure 1: GUI representation of the text maze

2.2 Example 1: Simple Program, Compilation

Here is a very simple aML example:

```
#load-random

// function that is run by program initially
main():void {
        goRight();
}

function goRight():void {
        cell c := (CPos); // variables at start
        move_R(); // moves the bot to the right
        c := (CPos);
        if (NOT isTarget(c)) {
                goRight();
        }
}
```

The first instruction in any aML program is the #load instruction. This can either be #load-random, which means aML will generate a random maze for you, or #load<filename> which means you have a maze stored in filename.txt that you wish to be used. The main() function follows, and calls the recursive void function goRight(). goRight() instructs the bot to move right and check if the current cell (designated by the special variable CPos, standing for "current position") is the target. If not, it calls itself again. Obviously in most cases this bot will not be very effective and recurse endlessly (which amL will not stop from happening!), as in the case shown here:

Another syntax rule in aML is that any variables in a function must be declared and initialized at the start of the function, prior to any other instructions. This is why cell c is initially set to a value before being reset after the use of the special move_R() function (which instructs the bot to move right, if possible).

In order to compile an aML program, first construct aml.exe, which will compile the aML source code to an executable java program. In order to do this, use the provided makefile in the source files and simply type make into the command line. This will construct aml.exe. Then, if the example was in a file called example.aml, compile it by typing aml -c example.aml into the command line. This will construct a java program. Execute this by typing java example and the program will run.

Figure 2: GUI representation of the text maze

2.3 Example 2: Depth-First Search

It is possible to use aML to write much more complex functions than the first example. For example, here is a program that implements depth-first search:

```
#load-random
main():void{
        DFS();
}

function DFS():void{
        cell node := (CPos);

        if (isTarget(node)){
                exit();
        };

        if(myvisited(node)){
                DFS();
```

```
        };
        else{
                if (isSource(node)){
                        exit();
                };

                revert();
                DFS();
        }
}

function myvisited(cell node):bool{
        if (node.hasleft() AND NOT visited(node.left())){
                move_L();
        } else { if(node.hastop() AND NOT visited(node.up())){
                move_U();
        } else { if (node.hasright() AND NOT visited(node.right())) {
                move_R();
        } else { if(node.hasbottom() AND NOT visited(node.down())){
                move_D();
        } else {
                return false;
        }}}}
        return true;
}
```

Note the use of special functions such as node.hasright(), node.right(), revert() (which backtracks) and visited(cell c).

2.4 Example 3: Greatest Common Denominator

aML can also be used to implement a simple mathematical function such as greatest common denominator, as in the following:

```
#load-random

main():void{
        integer x := gcd(7,49);
        print(x);
```

```
        exit();
}

function gcd(integer n, integer m):integer{
        if(n = m){
                return n;
        };
        else{
                if (n > m) {
                        return gcd(n - m, m);
                }
                else{
                        return gcd(m - n,n);
                };
        };
}
```

3 Language Reference Manual

3.1 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.

3.2 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.

3.2.1 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.

3.2.2 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

3.2.3 Keywords

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

if	return	display	remove	left	hasleft	integer
then	main	print	add	right	hasright	bool
else	void	isSource	head	up	hastop	list
load	function	visited	next	down	hasbottom	cell
random	exit	isTarget	isEmpty	CPos		
true	null	NOT	AND			
false			OR			

This language consists of many implicit variables and functions increasing the size of the reserved words list. There are a few keywords like display,null and next whose functionalities are not defined yet. But they are reserved for future use.

3.2.4 Literals

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

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

bool Literals A bool literal is either **True** or **False**, and is of data type bool

list Literals The list literal can include either the integer, bool, cell or list<datatype> types (cascaded lists).
eg. <[1]>,<[1,2,3]>,<[[1,2,3],[4,5]>,<[true, false, true]>
.
As can be seen above the list literals consist of the form list<integer>, list<list<....<list<integer>>...>>, list<bool> or list<list ... <list <bool>> ... >>. Details on list<datatype> and cell datatypes are provided in section 3.10.

3.2.5 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 of the logical statement. The { } are used to collect groups of statements into a single compound statement block. The < > and [ ] are used to instantiate the list<datatype> variables.

3.3 Syntax Notation

In all of the syntactic notation used in this manual, the non-terminal symbols are denoted in *italics*, and literal words and characters in **bold.** Alternative symbols are listed on separate lines. An optional terminal or non-terminal symbol is indicated by subscripting it with 'opt'.
eg. $expression_{opt}$ denotes an optional expression

3.4 Identifier interpretation

aML interprets the identifier based on it's type. Each identifier has a storage associated with it where a certain value is stored. The type of the identifier determines the meaning of the values located in the storage of the identifier. In aML each identifier's storage exists as long as the block enclosing the identifier is being executed.
aML supports a 3 fundamental types:-

- integer - Objects declared as integers use 64 bits for storage and are signed. They are primarily used for arithmetic operations.
- bool - Objects declared as bools act as binary variables and can either take the value **true** or **false**.
- cell - A cell object stores the attributes of a cell of a maze.

There is one derived type list<type> which is used for storing lists of objects of the fundamental types as well as the list type. By this recursive definition, aML allows cascading of these lists.
More details on the cell and list<type> datatypes is provided in section 3.11.

The complete data type definitions present in aML are as follows:-

datatype:-
integer

bool
cell
list<*datatype*>

Note:- Each datatype is different from each other and no two different datatypes can be combined together in a valid operation defined in aML. Therefore there are no type-conversion rules defined for aML.

3.5 Expressions

The complete syntax is provided in section 3.12. This section introduces the definition of the expression types which are the basic building blocks of any language.

3.5.1 Primary Expressions

Primary expressions are identifiers, constants, or expressions in parentheses. They also include the variable CPos which will be explained in section 3.11.

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

An identifier is a primary expression provided it's type is specified in it's declaration.
A literal is a primary expression. The type of the literal may include integer, bool or list<type>. The syntax notation for literal including the definition of list literals is given in detail in section 3.12.
A paranthesized expression is a primary expression whose type and value are equal to those of the non-paranthesized one.
CPos refers to the current position of the bot in the maze. It is a tracking variable and is used primarily to assign values to identifiers of cell datatypes. null is a constant which is assigned by default to identifiers of the list<type> and cell datatypes. It signifies no storage allotted to the identifier yet.

3.5.2 Operators

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	

Relational Operators The relational operators all return values of bool 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. For the cell datatype, the == and ~= compare the cell location in the map to which both the operands point to. As for the list<type> datatype, the two operators check if two variables referencing list datatypes point to the same list object.

bool Operators The bool operators all return values of bool type (either True or False). There are three bool 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 bool 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.

Assignment Operators There is a single assignment operator in aML, **:=**, which does simple assignment. It is a binary operator which assigns the value of the right operand to the storage of the left operand.

assignment-expression:-
identifier **:=** *expression*

The type of the expression must be identical to the type of 'lvalue'.

Associative Operator The **.** operator is used for function calls on variables represented by identifiers. The structure of statements involving the operator is shown in section3.12.

3.6 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. Declarations can be divided into variable and function declarations. Variable declarations refers to the declaration of identifiers whose type belongs to one of the datatypes mentioned and is different from function declarations both syntactically and semantically.

3.6.1 Variable Declarations

The rule representing the declaration of identifiers is listed in the complete Syntax summary in section 3.12. The declaration of identifiers is similar to many strongly typed languages where thet type associated with the identifier must be specified during declaration. In aML variable declaration is allowed only at the beginning of the main method and other functions. Without any loss of generality variable declaration is not allowed to intermix with statements and also it is encourage that while declaring variables at the top, they are assigned to literal values initally, or function calls, but not other variables. They can be assigned to subsequent variables using assignment statements in the body of the function.

declaration-expression:-
datatype identifer := *literal*
datatype identifier := **(CPos)**

datatype identifier := lang_functions

Examples of some declarations are given below:-

- integer x;
- bool flag;
- cell node;
- list<integer> mylist;

3.6.2 Variable Initialization

When an identifier is declared, an initial value must also be specified. The identifier can be re-initalized after it's declaration using assignment statements.

init-expression:-
identifier := *expression*

Care must be taken to ensure that the identifer's type must be consistent with the type of the expression.

A few examples of variable initializations are provided below;

- x := 10;
- flag := false;
- node := null;
- mylist.head() := 1;

The exact rule is provided in the Syntax summary in section 3.12. Initialization can also be combined with declaration in a single step. This is also shown in final section.

3.6.3 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 **{** *vdecl_list body* **}**

function_header:-
  function *identifier (args_list*$_{opt}$*)* **:** *return_type*

args_list:-
  datatype identifier
  datatype identifier, *args_list*

vdecl:
  datatype identifier := *litera*
  datatype identifier := **(CPos)**
  datatype identifier := *lang_functions*

vdecl_list:
  empty declaration
  vdecl vdecl_list

body:-
  compound-statement

Function calls are handled in section 3.12. Compound statements are described in detail in the section below.
Since function calls are part of compound statements, aML allows recursive functions, which is necessary owing to the absence of any looping constructs in this language. Also compound-statements do not allow function definitions, so functions cannot be declared within functions.

3.7 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 3.12.3. The following definition gives an idea about the components of a statement. The entire definition integrated with other definitions is present in section 3.12.

3.7.1 Expression statement

expression-statement:-
  expression;
Expression statement consist of assignments and function calls.

3.7.2 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.

3.7.3 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 bool 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.

3.7.4 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.

3.8 Scope rules

Programs are not multi-file in AML, so external scope is not a worry. The lexical scope of identifers 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 are not permitted. 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 3.10.

3.9 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 with name **filename**. (The file must be placed in the 'maps' directory). 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.

3.10 Implicit identifiers and functions

aML consists of many implicit identifers or variables and functions. By implicit, it follows that these identifiers can be used without prior declaration as is the case for any user defined identifier or function. However they cannot be modified by the user. Their usage is mostly restricted to bool queries and assigning their values to user-defined identifiers. The variables and functions along with their meaing are provided below:-

3.10.1 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.

3.10.2 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
- visited(id) - checks if the cell refered to by id has been visited or not

3.11 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 bool 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.

3.11.1 list<datatype>

The list<datatype> from it's definition in section3.6.1 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();

3.11.2 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 dataype 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.

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

cell functions

- isTarget(id) - returns true if the cell is a target as specified in the maze
- isSource(id) - returns true if the cell is the start point of the maze
- get_Loc(id) - returns the integer ID of the cell

Here id refers to an identifer pointing to a cell datatype.

3.12 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.

3.12.1 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)

literal:-
primitive_literal
$<$[*list_literal*$_{opt}$]$>$

primitive_literal:-
integer_literal
bool_literal

list_literal:-
sub_list
[*list_literal*]
list_literal,[*sub_list*]

sub_list:-

primitive_literal
primitive_literal,sub_list

init-expression:-
declarator := expression

datatype:-
integer
bool
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.**remove()**
identifier.**isEmpty()**
identifier.**head()**

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

maze_functions:-
visited(*identifier*)
get_Loc(*identifier*)

lang_functions:-
identifier(*actual_args*$_{opt}$)

actual_args:-
primary_expression
primary_expression, *actual_args*

3.12.2 Statements

Statements are logical sentences that can be formed by the language. A compound statement is a group of statements occuring in a linear fashion one after the other.
compound-statement:-
{*statement-list*}

statement-list:-
statement
statement statement-list

statement:-

expression;
return *expression*;
{ ***statement-list*** **}**
if (*expression*) *statement*;
if (*expression*) *statement* **else** *statement*
exit();
print(*expression*);
move_functions;
lang_functions;
identifier.**add**(*expression*);

move_functions:-

move_To(*identifier*)
move_U()
move_D()
move_L()
move_R()
revert()

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

3.12.3 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

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

func-def:-
main():void {*vdecl_ list statement-list***}**
function *identifier*(*formal-args*$_{opt}$)**:***return-type***{***vdecl_ list statement-list***}**

formal-args$_{opt}$*:-*
datatype identifier
*datatype identifier***,***formal-args*

return-type:-
datatype
void

vdecl:
datatype identifier := *literal*
datatype identifier := **(CPos)**
datatype identifier := *lang_functions*

vdecl_ list:
empty declaration
vdecl vdecl_ list

4 Project Plan

The project was a group effort from all individuals. After our initial meeting, we met as often as necessary during the project work period. Our group meetings were generally focused on discussing further ideas for the project and designating certain work assignments for group members. These meetings also served as a way of clarifying any issues or questions involving the project. Email communication was also an important tool throughout the project.

4.1 Team Responsibilities

There were no specific responsibilities given to any of the team members. The completion of the project was a collaborative effort with each member being responsible for completing different parts of each step during the semester. Once a step was completed, the entire group would check to make sure that nothing needed to be edited further. With that being said, there were certain parts that each member was responsible for completing. For the completion of the LRM, Evan was responsible for Lexical Conventions and Expressions, Tim was responsible for the Introduction and Declarations, Nikhil worked on Statements, Scope Rules, and Preprocessor Directives, and Ram did Implicit Variables and Functions, Types Revisited, and the Syntax Summary.

4.2 Project Timeline

The following dates were set as hard deadlines for project goals:

```
9-26-2012 Project Proposal
10-31-2012 Language Reference Manual
12-11-2012 Implementation of Code
12-18-2012 Project Presentation
12-19-2012 Final, Completed Project
```

Any other deadlines were "soft" deadlines that were worked around schedules.

4.3 Software Development Environment

The project was developed on Windows using OCaml 4.00 and the most recent version of Java. We used the Git version control system through the

website GitHub.com. The project was tested and completed in UNIX and Eclipse and will run in both.

4.4 Project Log

These were the major milestones of our project over the semester:

```
9-13-2012 Group Finalized, First Meeting
9-20-2012 Rough Draft of Proposal Created
9-26-2012 Project Proposal Completed
10-2-2012 Proposal Feedback with Bonan Liu
10-13-2012 Grammar Finalization
10-20-2012 LRM Breakdown/Assignment
10-28-2012 LRM Rough Draft
10-31-2012 Language Reference Manual Completed
11-6-2012 Feedback of LRM Received/Discussed
11-13-2012 Symantic Analysis
11-18-2012 Implementation of Compiler
12-11-2012 Implementation of all Code
12-16-2012 Project Finalization
12-18-2012 Project Presentation
12-19-2012 Final, Completed Project
```

5 Architecture Design

aML was designed in such a fashion that the aML syntax was similar to the java syntax making the translation to java much easier. Hence java was chosen to be the language to be translated to. The standard library functions were provided in java, where the maze was visualized using the java Swing GUI.

Keeping this in mind, the architecture of aML was designed as shown below tracing the steps from character streams typed by the user all the way to compiled java bytecode.

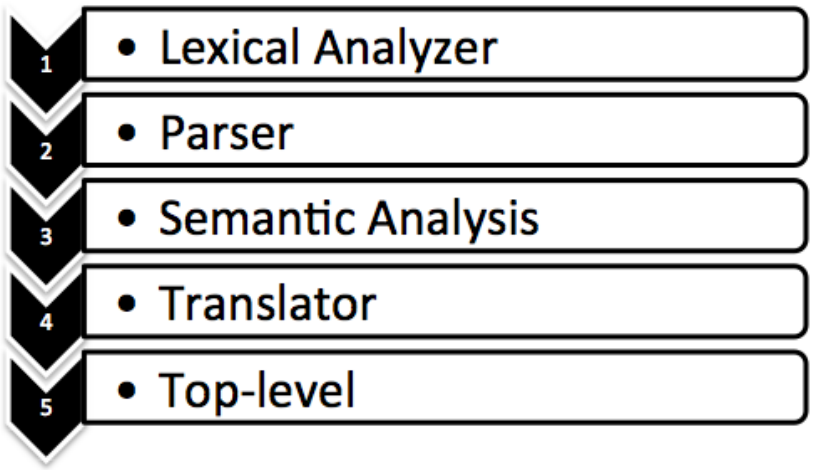

Figure 3: Block diagram of the aML translator

1. The lexical analyzer accepts a character stream and converts into a token stream. These tokens are defined in scanner.mll - the set of acceptable words in our language. The token stream is passed onto the Parser.

2. The Parser ensures that the token stream input received is consistent with the grammar rules defined in parser.mly – the order in which the tokens can combine with each other forming the syntax of the language.

The parser while checking for grammar rule consistency forms the Abstract Syntax Tree or AST, with the help of the ast structures defined in ast.ml. This incidentally also contains the one-one mapping of ast nodes to translated java code. The ast structure is passed onto the Semantic Analyzer.

3. The Semantic Analyzer ensures that the syntax is actually meaningful. It ensures that the program defined by the user while conforming to the grammar actually make sense. For example the assignment of variables to values, the semantic analyzer ensures that the types have to be consistent for this to be a valid operation. The semantic analysis for aML is present in sast.ml.

4. Once the semantic analysis is done and successful without throwing any exceptions, the verified ast structure can be translated to a .java file and compiled to .class (java bytecode). This is done in the file compile.ml.

5. The file toplevel.ml is provided for the user's convenience to run the program using the command line interface. The exact usage is shown on typing ./aml in the command line.

6 Test Plan

For our test suite, we decided to include both unit and functional tests for maximum coverage of our language's features. Both are necessary because alone, these two types of testing give only a limited guarantee of correctness, but together they ensure that everything works how it is supposed to. Unit tests test individual features in isolation, for example the correct translation of the addition operator or a function declaration. These are important because before we can even hope to ensure that our full codebase works properly, we must ensure that each unitary component provides the correct output for a given input. Because we didn't want to accidentally omit a feature of our language from the unit tests, our strategy for writing unit tests was to go through the LRM and create a test case for each feature described therein.

Functional tests are at the other end of the testing spectrum – they validate not a simple input-to-output conversion, but rather the result of complex interactions between many parts of the code. An example functional test case could be something like a GCD algorithm or a simple search algorithm. The reason we need functional tests on top of unit tests is that there is no way to verify that the pieces of our code work properly *together* if we use only unit testing, which verifies that the pieces of our code work properly *apart*.

We decided to implement and automate our tests using bash script. All of this can be found in the tests subdirectory. Each individual test case has two components: (1) an aML source file containing the aML code relevant to the test case, and (2) a bash script file of the same name (but with a .test extension rather than a .aml extension) that contains the expected output from the test case as well as any other necessary information pertaining to the test case in question (for example, setting COMPILE_ONLY=true if the test case is meant to cause an error on compilation). The bash file "test-base" contains all the functions necessary for automating the testing process, including functions to compile and run an individual test case, to print error/info data to the console, and to run the full suite at one time.

7 Lessons Learned

- One of the things that we learned is that a group should always start early so that there is time to work out issues before deadlines. By starting a little earlier on some sections, we may have been able to avoid having to rush to fix certain issues before parts of the project were due.
- Another thing we learned is the need to be flexible with your ideas and not plan for a lot of features too early. A good approach would have been to start simple and build off of simple features as we move along. Planning for too many features caused us to have to change our thinking at certain points during the project.

7.1 Sriramkumar Balasubramanian

- Keep in frequent communication.
- Start earlier than you will.
- Spread the work out.
- Sleepless coding is inefficient and/or error-prone.

7.2 Evan Drewry

- Being super organized is a very important factor in coding project as a team, especially when the team isn't always working together in the same room. Lack of organization can lead to confusion or wasted time for other group members who are also working on the project.
- Testing incorrect programs is as important as testing correct ones. If we only included meaningful and well-formed programs in our test suite, we would have no guarantee that our compiler responds appropriately to malformed input. Instead, we would only know that it respond appropriately to correct input.

7.3 Tim Giel

- I learned that it is incredibly difficult to find a suitable time for everyone in a group to meet, especially when there are more people. With everyone's schedule constantly changing and workloads piling up as the semester went on, it became increasingly difficult to find a time for everyone. Fortunately, we were all pretty flexible and were able to meet as a group a lot which definitely helped in getting to know one another and how each of us work, which helped us work on our project more efficiently.

- I also learned that it is very tough when you don't have as much programming experience as others. While we all were essentially learning two new languages (aML and OCaml), my lack of experience made me have to work a little harder than the other group members I think.

7.4 Nikhil Helferty

- Keep in frequent communication.

- Start earlier than you will.

- Spread the work out.

- Sleepless coding is inefficient and/or error-prone.

8 Appendix

8.1 Lexical Analyzer

Listing 1: scanner.mll

```
{open Parser}

let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']

rule token =
  parse [' ' '\t' '\r' '\n'] { token lexbuf }
| '+' { PLUS }
| '-' { MINUS}
| '*' { TIMES }
| '/' { DIVIDE }
| '%' { MOD }
| '^' { EXP }
| '.' { ASSOC }
| '(' { LPAREN }
| ')' { RPAREN }
| '{' { LBRACE }
| '}' { RBRACE }
| '[' { LSQUARE }
| ']' { RSQUARE }
| '<' { LSR }
| '>' { GTR }
| ';' { STMTEND }
| ',' { COMMA }
| ':' { RTYPE }
| '#' { HASH }
| ">=" { GTREQL }
| "<=" { LSREQL }
| "~=" { NEQ }
| "=" { EQ }
| ":=" { ASSIGN }
| "true" { TRUE }
| "false" { FALSE }
| "null" { NULL }
| "NOT" { NOT }
| "AND" { AND }
| "OR" { OR }
| "load" { LOAD }
| "random" { RANDOM }
```

```
40 | "return" { RETURN }
41 | "exit" { EXIT }
42 | "function" { FUNC }
43 | "main" { ENTRY }
44 | "void" { VOID }
45 | "print" { PRINT }
46 | "integer" { INTEGER }
47 | "bool" { BOOLEAN }
48 | "list" { LIST }
49 | "cell" { CELL }
50 | "if" { IF }
51 | "else" { ELSE }
52 | "display" { DISPLAY }
53 | "move_U" { MOVEUP }
54 | "move_D" { MOVEDOWN }
55 | "move_L" { MOVELEFT }
56 | "move_R" { MOVERIGHT }
57 | "move_To" { MOVETO }
58 | "get_Loc" { LOC }
59 | "isTarget" { ISTARGET }
60 | "visited" { VISIT }
61 | "isSource" { SOURCE }
62 | "revert" { REVERT }
63 | "left" { LEFT }
64 | "right" { RIGHT }
65 | "up" { UP }
66 | "down" { DOWN }
67 | "hasleft" { HASLEFT }
68 | "hasright" { HASRIGHT }
69 | "hastop" { HASTOP }
70 | "hasbottom" { HASBTM }
71 | "CPos" { CUR_POS }
72 | "add" { LISTADD }
73 | "remove" { LISTREMOVE }
74 | "clear" { LISTCLEAR }
75 | "head" { LISTHEAD }
76 | "isEmpty" { LISTEMPTY }
77 | ['-']?['1'-'9']digit*|'0' as amlex { NUM_LITERAL(int_of_string amlex) }
78 | letter(letter|digit)* as amlex { ID(amlex) }
79 | "/*" { multicmnt lexbuf}
80 | "//" { singlecmnt lexbuf}
81 | eof { EOF }
82
83 and multicmnt =
84       parse "*/" { token lexbuf}
```

```
85 |_ { multicmnt lexbuf}
86
87 and singlecmnt =
88       parse "\n" { token lexbuf}
89 |_ { singlecmnt lexbuf}
```

8.2 Parser

Listing 2: parser.mly

```
%{ open Ast
       let parse_error pErr =
       print_endline pErr;
       flush stdout
%}

%token LPAREN RPAREN LBRACE RBRACE LSQUARE RSQUARE
%token PLUS MINUS TIMES DIVIDE MOD EXP
%token ASSOC ASSIGN
%token GTR LSR GTREQL LSREQL NEQ EQ
%token TRUE FALSE
%token STMTEND COMMA RTYPE HASH
%token EXIT RETURN FUNC ENTRY VOID LOAD RANDOM NULL
%token INTEGER BOOLEAN CELL LIST
%token IF ELSE PRINT DISPLAY
%token MOVEUP MOVEDOWN MOVELEFT MOVERIGHT MOVETO CUR_POS
%token ISTARGET VISIT SOURCE REVERT LOC
%token LEFT RIGHT UP DOWN HASLEFT HASRIGHT HASTOP HASBTM
%token LISTADD LISTREMOVE LISTCLEAR LISTHEAD LISTEMPTY
%token AND OR NOT
%token <string> ID
%token <int> NUM_LITERAL
%token EOF

%nonassoc ELSE
%left GTR LSR GTREQL LSREQL NEQ EQ
%left PLUS MINUS
%left TIMES DIVIDE
%left MOD
%right ASSIGN EXP
%left OR
%left AND
%right NOT

%start program
%type <Ast.program> program

%%

program:
  /* empty code */ { [] }
| program pre_process { $2 :: $1 }
```

```
43 | program func_decl { $2 :: $1 }
44
45 pre_process:
46        HASH LOAD LSR ID GTR { Load($4) }
47 | HASH LOAD MINUS RANDOM { Load("random") }
48
49 func_decl:
50        ENTRY LPAREN RPAREN RTYPE VOID LBRACE vdecl_list stmt_list RBRACE
51               { Main({
52                      mainId = "main";
53                      mainVars = List.rev $7;
54                      body = $8;
55                      })
56               }
57 | FUNC ID LPAREN formal_args RPAREN RTYPE return_type
58    LBRACE vdecl_list stmt_list RBRACE
59               { Func({
60                      funcId = $2;
61                      formalArgs = List.rev $4;
62                      reType = $7;
63                      localVars = List.rev $9;
64                      statements = $10;
65               })
66               }
67
68 return_type:
69   VOID { Void }
70 | data_type { Data($1) }
71
72 data_type:
73        INTEGER { Integer }
74        |CELL { Cell }
75        |BOOLEAN { Bool }
76        |formal_list { $1 }
77
78 formal_list:
79        |LIST LSR data_type GTR { List($3) }
80
81 formal_args:
82        /* no arguments */ { [] }
83        |data_type ID { [FormalVar($1, $2)] }
84        |formal_args COMMA data_type ID { FormalVar($3, $4) :: $1 }
85
86 vdecl_list:
87 /* No variable declaration */ { [] }
```

```
88 | vdecl_list vdecl { $2 :: $1 }
89
90 vdecl:
91  data_type ID ASSIGN vars STMTEND { Define($1,$2,Vars($4)) }
92 | data_type ID ASSIGN LPAREN CUR_POS RPAREN STMTEND { Define($1,$2,Pointer) }
93 | data_type ID ASSIGN ID LPAREN actual_args RPAREN STMTEND
94       { Define($1,$2,Funcall($4,List.rev $6)) }
95
96 stmt_list:
97   stmt { [$1] }
98 | stmt stmt_list { $1 :: $2 }
99
100 stmt:
101   expr STMTEND { Expr($1) }
102 | RETURN expr STMTEND { Return($2) }
103 | impl_fns STMTEND{ $1 }
104 | move_stmt STMTEND { Move($1) }
105 | ID ASSOC LISTADD LPAREN expr RPAREN STMTEND { ListAdd($1,$5) }
106 | MOVETO LPAREN ID RPAREN STMTEND{ MoveTo($3) }
107 | LBRACE stmt_list RBRACE { StmtBlk($2) }
108 | IF LPAREN expr RPAREN stmt STMTEND { If($3, $5, StmtBlk([])) }
109 | IF LPAREN expr RPAREN stmt ELSE stmt { If($3, $5, $7) }
110
111 impl_fns:
112 | REVERT LPAREN RPAREN { Revert }
113 | EXIT LPAREN RPAREN{ Exit }
114 | DISPLAY LPAREN RPAREN{ Display }
115 | PRINT LPAREN expr RPAREN{ Print($3) }
116
117 move_stmt:
118 | MOVEUP LPAREN RPAREN { 1 }
119 | MOVEDOWN LPAREN RPAREN { 2 }
120 | MOVERIGHT LPAREN RPAREN { 3 }
121 | MOVELEFT LPAREN RPAREN { 4 }
122
123 expr:
124   vars { Vars($1) }}
125
126 \begin{lstlisting}[caption=parser.mly]
127 %{ open Ast
128        let parse_error pErr =
129        print_endline pErr;
130        flush stdout
131 %}
132
```

```
%token LPAREN RPAREN LBRACE RBRACE LSQUARE RSQUARE
%token PLUS MINUS TIMES DIVIDE MOD EXP
%token ASSOC ASSIGN
%token GTR LSR GTREQL LSREQL NEQ EQ
%token TRUE FALSE
%token STMTEND COMMA RTYPE HASH
%token EXIT RETURN FUNC ENTRY VOID LOAD RANDOM NULL
%token INTEGER BOOLEAN CELL LIST
%token IF ELSE PRINT DISPLAY
%token MOVEUP MOVEDOWN MOVELEFT MOVERIGHT MOVETO CUR_POS
%token ISTARGET VISIT SOURCE REVERT LOC
%token LEFT RIGHT UP DOWN HASLEFT HASRIGHT HASTOP HASBTM
%token LISTADD LISTREMOVE LISTCLEAR LISTHEAD LISTEMPTY
%token AND OR NOT
%token <string> ID
%token <int> NUM_LITERAL
%token EOF

%nonassoc ELSE
%left GTR LSR GTREQL LSREQL NEQ EQ
%left PLUS MINUS
%left TIMES DIVIDE
%left MOD
%right ASSIGN EXP
%left OR
%left AND
%right NOT

%start program
%type <Ast.program> program

%%

program:
  /* empty code */ { [] }
| program pre_process { $2 :: $1 }
| program func_decl { $2 :: $1 }

pre_process:
       HASH LOAD LSR ID GTR { Load($4) }
| HASH LOAD MINUS RANDOM { Load("random") }

func_decl:
       ENTRY LPAREN RPAREN RTYPE VOID
       LBRACE vdecl_list stmt_list RBRACE
```

```
178                 { Main({
179                         mainId = "main";
180                         mainVars = List.rev $7;
181                         body = $8;
182                 })
183                 }
184 | FUNC ID LPAREN formal_args
185 | ID { Id($1) }
186 | NULL { Null }
187 | LPAREN expr RPAREN { Paran($2) }
188 | expr PLUS expr { BinOpr(Add,$1,$3) }
189 | expr MINUS expr { BinOpr(Sub,$1,$3) }
190 | expr TIMES expr { BinOpr(Mul,$1,$3) }
191 | expr DIVIDE expr { BinOpr(Div,$1,$3) }
192 | expr EXP expr { BinOpr(Pow,$1,$3) }
193 | expr MOD expr { BinOpr(Mod,$1,$3) }
194 | expr EQ expr { BinOpr(Eql,$1,$3) }
195 | expr NEQ expr { BinOp}
196
197 \begin{lstlisting}[caption=parser.mly]
198 %{ open Ast
199         let parse_error pErr =
200         print_endline pErr;
201         flush stdout
202 %}
203
204 %token LPAREN RPAREN LBRACE RBRACE LSQUARE RSQUARE
205 %token PLUS MINUS TIMES DIVIDE MOD EXP
206 %token ASSOC ASSIGN
207 %token GTR LSR GTREQL LSREQL NEQ EQ
208 %token TRUE FALSE
209 %token STMTEND COMMA RTYPE HASH
210 %token EXIT RETURN FUNC ENTRY VOID LOAD RANDOM NULL
211 %token INTEGER BOOLEAN CELL LIST
212 %token IF ELSE PRINT DISPLAY
213 %token MOVEUP MOVEDOWN MOVELEFT MOVERIGHT MOVETO CUR_POS
214 %token ISTARGET VISIT SOURCE REVERT LOC
215 %token LEFT RIGHT UP DOWN HASLEFT HASRIGHT HASTOP HASBTM
216 %token LISTADD LISTREMOVE LISTCLEAR LISTHEAD LISTEMPTY
217 %token AND OR NOT
218 %token <string> ID
219 %token <int> NUM_LITERAL
220 %token EOF
221
222 %nonassoc ELSE
```

```
%left GTR LSR GTREQL LSREQL NEQ EQ
%left PLUS MINUS
%left TIMES DIVIDE
%left MOD
%right ASSIGN EXP
%left OR
%left AND
%right NOT

%start program
%type <Ast.program> program

%%

program:
  /* empty code */ { [] }
| program pre_process { $2 :: $1 }
| program func_decl { $2 :: $1 }

pre_process:
       HASH LOAD LSR ID GTR { Load($4) }
| HASH LOAD MINUS RANDOM { Load("random") }

func_decl:
       ENTRY LPAREN RPAREN RTYPE VOID
       LBRACE vdecl_list stmt_list RBRACE
              { Main({
                     mainId = "main";
                     mainVars = List.rev $7;
                     body = $8;
                     })
              }
| FUNC ID LPAREN formal_argsr(Neq,$1,$3) }
| expr GTR expr { BinOpr(Gtr,$1,$3) }
| expr LSR expr { BinOpr(Lsr,$1,$3) }
| expr GTREQL expr { BinOpr(Geq,$1,$3) }
| expr LSREQL expr { BinOpr(Leq,$1,$3) }
| NOT expr { BinOpr(Not,$2,$2) }
| expr AND expr { BinOpr(And,$1,$3) }
| expr OR expr { BinOpr(Or,$1,$3) }
| ID ASSIGN expr { Assign($1,$3) }
| ID LPAREN actual_args RPAREN { Funcall($1,List.rev $3) }
| ID ASSOC LISTREMOVE LPAREN RPAREN { Assoc(Remove,$1) }
| ID ASSOC LISTCLEAR LPAREN RPAREN { Assoc(Next,$1) }
| ID ASSOC LISTHEAD LPAREN RPAREN { Assoc(Head,$1) }
```

```
| ID ASSOC LISTEMPTY LPAREN RPAREN { Assoc(Empty,$1) }
| ID ASSOC UP LPAREN RPAREN { Assoc(Up,$1) }
| ID ASSOC DOWN LPAREN RPAREN { Assoc(Down,$1) }
| ID ASSOC LEFT LPAREN RPAREN { Assoc(Left,$1) }
| ID ASSOC RIGHT LPAREN RPAREN { Assoc(Right,$1) }
| ID ASSOC HASLEFT LPAREN RPAREN { Assoc(Hleft,$1) }
| ID ASSOC HASRIGHT LPAREN RPAREN { Assoc(Hright,$1) }
| ID ASSOC HASTOP LPAREN RPAREN { Assoc(Htop,$1) }
| ID ASSOC HASBTM LPAREN RPAREN { Assoc(Hbtm,$1) }
| LOC LPAREN ID RPAREN { Loc($3) }
| SOURCE LPAREN ID RPAREN { Src($3) }
| ISTARGET LPAREN ID RPAREN { Target($3) }
| VISIT LPAREN expr RPAREN { Visit($3) }
| LPAREN CUR_POS RPAREN}

\begin{lstlisting}[caption=parser.mly]
%{ open Ast
       let parse_error pErr =
       print_endline pErr;
       flush stdout
%}

%token LPAREN RPAREN LBRACE RBRACE LSQUARE RSQUARE
%token PLUS MINUS TIMES DIVIDE MOD EXP
%token ASSOC ASSIGN
%token GTR LSR GTREQL LSREQL NEQ EQ
%token TRUE FALSE
%token STMTEND COMMA RTYPE HASH
%token EXIT RETURN FUNC ENTRY VOID LOAD RANDOM NULL
%token INTEGER BOOLEAN CELL LIST
%token IF ELSE PRINT DISPLAY
%token MOVEUP MOVEDOWN MOVELEFT MOVERIGHT MOVETO CUR_POS
%token ISTARGET VISIT SOURCE REVERT LOC
%token LEFT RIGHT UP DOWN HASLEFT HASRIGHT HASTOP HASBTM
%token LISTADD LISTREMOVE LISTCLEAR LISTHEAD LISTEMPTY
%token AND OR NOT
%token <string> ID
%token <int> NUM_LITERAL
%token EOF

%nonassoc ELSE
%left GTR LSR GTREQL LSREQL NEQ EQ
%left PLUS MINUS
%left TIMES DIVIDE
%left MOD
```

```
%right ASSIGN EXP
%left OR
%left AND
%right NOT

%start program
%type <Ast.program> program

%%

program:
  /* empty code */ { [] }
| program pre_process { $2 :: $1 }
| program func_decl { $2 :: $1 }

pre_process:
       HASH LOAD LSR ID GTR { Load($4) }
| HASH LOAD MINUS RANDOM { Load("random") }

func_decl:
       ENTRY LPAREN RPAREN RTYPE VOID
       LBRACE vdecl_list stmt_list RBRACE
              { Main({
                     mainId = "main";
                     mainVars = List.rev $7;
                     body = $8;
                     })
              }
| FUNC ID LPAREN formal_args { Pointer }

prim_vars:
  NUM_LITERAL { Lit_Int($1) }
| TRUE { Lit_Bool(true) }
| FALSE { Lit_Bool(false) }

vars:
  prim_vars { $1 }
| LSR complete_list GTR{ Lit_List($2) }

complete_list:
  LSQUARE RSQUARE{ [] }
| LSQUARE var_list RSQUARE { $2 }
| LSQUARE complete_list RSQUARE { [Lit_List($2)] }
| LSQUARE complete_list COMMA LSQUARE var_list RSQUARE RSQUARE
       { Lit_List($2) :: [Lit_List($5)] }
```

```
358
359 var_list:
360   prim_vars { [$1] }
361 | prim_vars COMMA var_list { $1::$3 }
362
363 actual_args:
364 /* no arguments*/ { [] }
365 | expr { [$1] }
366 | actual_args COMMA expr { $3 :: $1 }
```

8.3 Abstract Syntax Tree

Listing 3: ast.ml

```
type binopr = Add | Sub | Mul | Div | Mod | Eql | Neq
| Lsr | Leq | Gtr | Geq | Pow | And | Or | Not

type assoc = Remove| Next | Head | Empty | Up | Down
| Left | Right | Hleft | Hright | Htop | Hbtm

type datatype =
       Integer
       | Bool
       | Cell
       | List of datatype

type return_type =
       Void
| Data of datatype

type formal_args = FormalVar of datatype * string

type vars =
  Lit_Int of int
| Lit_Bool of bool
| Lit_List of vars list

type expr =
  Id of string
| Vars of vars
| Paran of expr
| BinOpr of binopr * expr * expr
| Assoc of assoc * string
| Assign of string * expr
| Funcall of string * expr list
| Loc of string
| Target of string
| Src of string
| Visit of expr
| Pointer
| Null

type vdecl =
       Define of datatype * string * expr

type stmt =
```

```
 StmtBlk of stmt list
| Expr of expr
| Display
| Move of int
| MoveTo of string
| Exit
| Revert
| Print of expr
| Return of expr
| ListAdd of string * expr
| If of expr * stmt * stmt

type main = {
              mainId : string;
              mainVars : vdecl list;
              body : stmt list;
       }

type func = {
              funcId : string;
              formalArgs : formal_args list;
              reType : return_type;
              localVars : vdecl list;
              statements : stmt list;
       }

type funcs =
  Main of main
| Func of func
| Load of string

type program =
       funcs list

let string_of_dt = function
              Integer -> "int"
       | Cell -> "Cell"
       | List(e) -> "List␣"
       | Bool -> "Boolean"

let string_of_assoc = function
              Remove -> "remove"
       | Next -> "clear"
       | Head -> "peek"
       | Empty -> "isEmpty"
```

```
88          | Up -> "up"
89          | Down -> "down"
90          | Left -> "left"
91          | Right -> "right"
92          | Hleft -> "hasLeft"
93          | Hright -> "hasRight"
94          | Htop -> "hasTop"
95          | Hbtm -> "hasBottom"
96
97  let rec string_of_rt = function
98          Void -> "void"
99          | Data(e) -> string_of_dt e
100
101 let string_of_op = function
102           Add -> "+"
103         | Sub -> "-"
104         | Mul -> "*"
105         | Div -> "/"
106         | Eql -> "=="
107         | Neq -> "!="
108         | Lsr -> "<"
109         | Leq -> "<="
110         | Gtr -> ">"
111         | Geq -> ">="
112         | Pow -> "^"
113         | Mod -> "%"
114         | And -> "&&"
115         | Or -> "||"
116         | Not -> "!"
117
118 let rec evalListexpr = function
119         | [] -> ""
120         | hd::[] -> string_of_var hd
121         | hd::tl -> string_of_var hd ^ "," ^ evalListexpr tl
122 and string_of_var = function
123         | Lit_Int(f) -> string_of_int f
124         | Lit_Bool(f) -> string_of_bool f
125         | Lit_List(f) -> "new␣List(␣new␣Object␣[]␣{"
126        ^ evalListexpr f ^ "})"
127
128 let rec string_of_expr = function
129        Vars(e) ->
130                        string_of_var e
131         | Id(s) -> s
132         | BinOpr(o, e1, e2) ->
```

```
133                 begin match o with
134                         | Pow -> "Math.pow(" ^ string_of_expr e1
135                   ^ "␣,␣" ^ string_of_expr e2 ^ ")"
136                         | Not -> "!" ^ string_of_expr e1
137                         | _ ->
138                                 string_of_expr e1 ^ "␣" ^ (
139                   match o with
140                         Add -> "+"
141                       | Sub -> "-"
142                       | Mul -> "*"
143                       | Div -> "/"
144                       | Eql -> "=="
145                       | Neq -> "!="
146                       | Lsr -> "<"
147                       | Leq -> "<="
148                       | Gtr -> ">"
149                       | Geq -> ">="
150                       | And -> "&&"
151                       | Or -> "||"
152                       | Mod -> "%"
153                       | Pow -> "^"
154                       | Not -> "!"
155                       ) ^ "␣" ^ string_of_expr e2
156                       end
157         | Assign(v, e) -> v ^ "␣=␣" ^ string_of_expr e
158         | Funcall(f, el) -> f ^ "("
159         ^ String.concat ",␣" (List.map string_of_expr el) ^ ")"
160         | Assoc(f, e) ->
161            begin
162                match f with
163                | Remove ->"(Cell)"^ e ^ "." ^ string_of_assoc f ^"()"
164                | Next -> e ^ "." ^ string_of_assoc f ^"()"
165                | Head -> "(Cell)"^ e ^ "." ^ string_of_assoc f ^"()"
166                | Empty -> e ^ "." ^ string_of_assoc f ^"()"
167                | _ -> "AMLJava." ^ string_of_assoc f ^"()"
168            end
169         | Paran(e1) -> "␣(␣" ^ string_of_expr e1 ^ "␣)␣"
170         | Loc(e) -> e^".get_Loc()"
171         | Target(e) -> e^".isTarget()"
172         | Src(e) -> e^".isSource()"
173         | Visit(e) -> string_of_expr e ^ ".getVisited()"
174         | Pointer -> "AMLJava.current"
175         | Null -> "null"
176
177 let rec string_of_stmt = function
```

```
178           StmtBlk(stmts) -> "{\n"
179         ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
180         | Expr(expr) -> string_of_expr expr ^ ";\n";
181         | ListAdd(s,t) -> s ^ ".add(" ^ string_of_expr t ^ ");\n"
182         | Move(e) ->
183            begin
184                match e with
185                    | 1 -> "AMLJava.move_U();\n"
186                    | 2 -> "AMLJava.move_D();\n"
187                    | 3 -> "AMLJava.move_R();\n"
188                    | 4 -> "AMLJava.move_L();\n"
189                    | _ -> ""
190            end
191         | Exit -> "return;\n"
192         | Revert -> "AMLJava.revert();\n"
193         | Display -> "AMLJava.display();\n"
194         | Print(e) -> "System.out.println␣(("
195         ^ string_of_expr e ^ "));\n"
196         | Return(expr) -> "return␣" ^ string_of_expr expr ^ ";\n";
197         | If(e, s, StmtBlk([])) -> "if␣(" ^ string_of_expr e
198         ^ ")\n" ^ string_of_stmt s
199         | If(e, s1, s2) -> "if␣(" ^ string_of_expr e ^ ")\n"
200         ^ string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
201         | MoveTo(x) -> "AMLJava.move(" ^ x ^");\n"
202
203 let string_of_vdecl = function
204         Define(dtt, nm, v) -> string_of_dt dtt ^ "␣" ^ nm
205         ^ "␣=␣" ^ string_of_expr v ^ ";\n"
206
207 let string_of_fparam = function
208         FormalVar(dt,s) -> string_of_dt dt ^ "␣" ^ s
209
210 let string_of_func (func) =
211         "Function␣name␣:␣" ^ func.funcId ^ "\n" ^
212         "Formal␣Parameter(s)␣:␣"
213         ^ String.concat "," (List.map string_of_fparam func.formalArgs)
214         ^ "\n" ^
215         "Return␣Type:␣" ^ "\n" ^ string_of_rt func.reType
216
217 let string_of_fdecl = function
218         | Func(fdecl) ->
219         "\npublic␣static␣"^ string_of_rt fdecl.reType ^"␣"
220                ^ fdecl.funcId ^ "(" ^ String.concat ",␣"
221                (List.map string_of_fparam fdecl.formalArgs) ^ "){\n" ^
222          String.concat "" (List.map string_of_vdecl fdecl.localVars) ^
```

```
223         String.concat "" (List.map string_of_stmt fdecl.statements) ^
224         "}\n"
225       | Main(fdecl) ->
226         String.concat "" (List.map string_of_vdecl fdecl.mainVars) ^
227         String.concat "" (List.map string_of_stmt fdecl.body) ^
228         "}\n"
229       | Load(str) ->
230       begin
231           match str with
232           | "random" ->
233 "public␣static␣void␣main(String[]␣args){\nAMLJava.buildMaze(\""
234 ^ str ^"\");"
235           | _ ->
236 "public␣static␣void␣main(String[]␣args){\nAMLJava.buildMaze(\""
237 ^ str ^".txt\");"
238       end
239 let string_of_program (funcs) prog_name =
240    "import␣java.util.*;\n\npublic␣class␣" ^ prog_name
241    ^ "{\n" ^ (String.concat "\n" (List.map string_of_fdecl funcs))
242    ^ "}"
```

8.4 Semantic Analyzer

Listing 4: sast.ml

```
open Ast

type env = {
       mutable functions : funcs list ;
}

let eql_fname id = function
  Func(fn) -> fn.funcId = id
| _ -> false

let eql_mname id = function
       Main(fn) -> fn.mainId = id
| _ -> false

let rec count_fn_id id = function
       | [] -> 0
       | hd::tl ->
       begin
           match hd with
           | Func(fn) -> if fn.funcId = id then
                  1 + count_fn_id id tl
              else
                  count_fn_id id tl
           | Main(fn) ->  if fn.mainId = id then
                  1 + count_fn_id id tl
              else
                  count_fn_id id tl
       | _ -> count_fn_id id tl
       end

(*determines if the given function exists*)
let isFunction func env =
       let id = (match func with
       Func(f) -> f.funcId|Main(f) -> f.mainId | _ -> "_DNE") in
                      if count_fn_id id env.functions = 1 then
                             true
                      else
                             let e = "Duplicate␣function␣name:␣" ^ id in
                                    raise (Failure e)

(*Determine if a function with given name exists*)
let isFunction_name id env =
```

```
43    List.exists (eql_fname id) env.functions
44
45 let isMain_name id env =
46    List.exists (eql_mname id) env.functions
47
48 (*Returns the function that has the given name*)
49 let getFunc_fname id env =
50        try
51               let afunc =
52           List.find (eql_fname id) env.functions in
53                       afunc (*Found a function with name like that*)
54        with Not_found ->
55        raise(Failure("Function␣" ^ id ^ "has␣not␣yet␣been␣declared" ) )
56
57 let get_main id env =
58        try
59               let afunc =
60           List.find (eql_fname id) env.functions in
61                       afunc
62        with Not_found ->
63        raise(Failure(id ^ "has␣not␣yet␣been␣declared" ) )
64
65 (*this is for generic functions only*)
66 let is_formal_param func fpname =
67    List.exists (function FormalVar(_,name) -> name = fpname) func.formalArgs
68
69 (*Determines if a formal parameter with the given name 'fpname' exists in the given function*)
70 let exists_formal_param func fpname =
71        match func with
72        | Func(func) -> is_formal_param func fpname
73        | _ -> false (*not applicable*)
74
75 (*for generic functions only*)
76 let is_variable_decl func vname =
77    List.exists (function Define(_,name,_) -> name = vname) func.localVars
78
79 let is_variable_decl_main func vname =
80    List.exists (function Define(_,name,_) -> name = vname) func.mainVars
81
82
83 (*Determines if a variable declaration with the given name 'vname' exists in the given functioin
84 let exists_variable_decl func vname =
85        match func with
86        | Func(func) -> is_variable_decl func vname
87        | _ -> false
```

```
88
89 (*this gets formal paramters for a generic function*)
90 let get_fpdt func fpname =
91        try
92               let fparam =
93            List.find (function FormalVar(_,name) ->
94                name = fpname) func.formalArgs in
95                       let FormalVar(dt,_) = fparam in
96                              dt (*return the data type*)
97        with Not_found ->
98        raise (Failure ("Formal␣Parameter␣" ^ fpname
99        ^ "␣should␣exist␣but␣was␣not␣found␣in␣␣function␣"
100       ^ func.funcId)) (*this shouldn't not happen*)
101
102 (*gets the variable type - only for generic functions*)
103 let get_var_type func vname =
104       try
105              let var =
106           List.find (function Define(_,vn,_) ->
107               vn = vname) func.localVars in
108                      let Define(dt,_,_) = var in
109                             dt (*return the data type*)
110       with Not_found -> raise (Failure ("Variable␣"
111    ^ vname ^ "␣should␣exist␣but␣was␣not␣found␣in␣the␣␣function␣"
112    ^ func.funcId)) (*this shouldn't not happen*)
113
114 let get_var_type_main func vname =
115       try
116              let var =
117           List.find (function Define(_,vn,_) ->
118               vn = vname) func.mainVars in
119                      let Define(dt,_,_) = var in
120                             dt (*return the data type*)
121       with Not_found ->
122       raise (Failure ("Variable␣" ^ vname
123       ^ "␣should␣exist␣but␣was␣not␣found␣in␣" ^ func.mainId))
124       (*this shouldn't not happen*)
125
126
127 let get_type_main main name =
128       if is_variable_decl_main main name (*It's a variable*)
129              then get_var_type_main main name
130       else
131              let e = "Variable␣" ^ name
132       ^ "␣is␣being␣used␣without␣being␣declared␣in␣main␣"
```

```
133         ^ main.mainId in
134                         raise (Failure e)
135
136 (*Returns the type of a given variable name *)
137 let get_type func name =
138         if is_variable_decl func name (*It's a variable*)
139                 then get_var_type func name
140         else
141                 if is_formal_param func name then
142                         get_fpdt func name
143                 else (*Variable has not been declared as it was not found*)
144                         let e = "Variable " ^ name
145             ^ " is being used without being declared in function "
146             ^ func.funcId in
147                                 raise (Failure e)
148
149 (*Determines if the given identifier exists*)
150 let exists_id name func =
151     (is_variable_decl func name) or (is_formal_param func name)
152
153 let exists_id_main name func = (is_variable_decl_main func name)
154
155 (*see if there is a function with given name "func"*)
156 let find_function func env =
157         try
158                 let _ = List.find (eql_fname func) env.functions in
159                         true (*return true on success*)
160         with Not_found -> raise Not_found
161
162 let isDup_fp_single func = function
163         FormalVar(_,my_name) ->
164                 function c ->
165                         function FormalVar(_,name) ->
166                 if my_name = name then
167                     if c = 0 then c+1
168                     else let e =
169                         "Duplicate formal parameter in function: "
170                         ^ func.funcId^ "\n" in
171                             raise (Failure e)
172                 else c
173
174 (*This check for duplicate formal parameters in a function*)
175 let cisDup_fp func =
176         let isdup f = List.fold_left (isDup_fp_single func f) 0 func.formalArgs
177         in let _ = List.map isdup func.formalArgs
```

```
178         in false
179
180 let dup_vdecl_single func = function
181         Define(_,mn,_) ->
182     function c ->
183         function Define(_,tn,_) ->
184             if mn = tn then
185                 if c = 0 then c+1
186                     else let e =
187                         "Duplicate␣variable␣declaration␣'"
188                         ^ mn ^"'␣in␣function␣:␣"
189                         ^ func.funcId in
190                         raise (Failure e)
191                         (*throw error on duplicate formal parameter.*)
192             else c
193
194 (*checks if there is a duplicate variable declaration for functions*)
195 let dup_vdecl = function
196   Main(func) -> false
197 | Load(func) -> false
198 | Func(func) ->
199         let isdup var =
200         List.fold_left (dup_vdecl_single func var) 0 func.localVars in
201                 let _ = List.map (
202                         function Define(_,varname,_) ->
203                                 List.map (
204                                         function FormalVar(_,formal_nm) ->
205                                                 if formal_nm = varname then
206                             let e =
207                                 "Redeclaring␣a␣formal␣parameter␣'"
208                                 ^ formal_nm
209                                 ^"'␣not␣allowed␣in␣function␣:␣"
210                                 ^ func.funcId ^"\n" in
211                                                                         raise(Failure e)
212                                                 else false
213                                 ) func.formalArgs
214                 ) func.localVars in
215                         let _ = List.map(isdup) func.localVars in
216                                 false
217
218 let is_int s =
219         try ignore (int_of_string s); true
220         with _ -> false
221
222 let rec int_flatten = function
```

```
223     | Lit_List(xs) -> List.concat (List.map int_flatten xs)
224                 | Lit_Int(x) -> [x] ;;
225
226 let rec bool_flatten = function
227     | Lit_List(xs) -> List.concat (List.map bool_flatten xs)
228                 | Lit_Bool(x) -> [x] ;;
229
230 let is_int_list ls =
231         try ignore (int_flatten ls); true
232         with _ -> false ;;
233
234 let is_bool_list ls =
235         try ignore (bool_flatten ls); true
236         with _ -> false ;;
237
238 let is_list ls =
239         is_int_list ls || is_bool_list ls
240
241 let is_string_bool = function "true" -> true
242         | "false" -> true | _ -> false
243
244 let rec is_num func env = function
245           Vars(e) -> begin
246                                                                 match e with
247                                                                 | Lit_Int(_) -> true
248                                                                 | _ -> false
249                                                         end
250         | Id(s) -> (function Integer -> true | _ -> false) (get_type func s)
251         | BinOpr(_,e1,e2) -> (is_num func env e1) && (is_num func env e2)
252         | Funcall(f,_) ->
253         let fn = (getFunc_fname f) env in
254             begin
255                 match fn with
256                     | Func(f) ->
257                           (string_of_rt f.reType) =
258                               (string_of_dt Integer)
259                     | _ -> false
260             end
261         | _ -> false
262
263 let rec is_num_main func env = function
264           Vars(e) ->
265         begin
266             match e with
267             | Lit_Int(_) -> true
```

```
268             | _ -> false
269          end
270         | Id(s) ->
271         (function Integer -> true | _ -> false)
272         (get_type_main func s)
273         | BinOpr(_,e1,e2) -> (is_num_main func env e1)
274         && (is_num_main func env e2)
275         | Funcall(f,_) -> let fn = (getFunc_fname f) env in
276         begin
277             match fn with
278                 | Func(f) -> (string_of_rt f.reType) =
279                     (string_of_dt Integer)
280                 | _ -> false
281         end
282         | _ -> false
283
284 let rec get_lit_type = function
285         | Lit_Int(_) -> Integer
286         | Lit_Bool(_) -> Bool
287         | Lit_List(e) -> List(get_lit_type (List.hd e))
288
289 let isArithmetic = function
290         | Add -> true
291         | Sub -> true
292         | Mul -> true
293         | Div -> true
294         | Mod -> true
295         | Pow -> true
296         | _ -> false
297
298 let isEql = function
299         | Eql -> true
300         | Neq -> true
301         | _ -> false
302
303 let isLogic = function
304         | And -> true
305         | Or -> true
306         | Not -> true
307         | _ -> false
308
309 let rec get_expr_type e func env=
310         match e with
311                 | Id(s) -> Data(get_type func s)
312                 | Vars(s) -> Data(get_lit_type s)
```

```
313             | BinOpr(op,e1,e2) ->
314         let t1 = get_expr_type e1 func env
315         and t2 = get_expr_type e2 func env in
316                  if isLogic op then
317                         begin
318                                 match t1,t2 with
319                                 | Data(Bool),Data(Bool) -> Data(Bool)
320                                 | _,_ -> raise
321                    (Failure "Invalid␣Types␣used␣in␣a␣Logical␣expression")
322                         end
323                    else if isEql op then
324                         begin
325                                 match t1,t2 with
326                                 | Data(Integer),Data(Integer) -> Data(Bool)
327                                 | Data(Bool),Data(Bool) -> Data(Bool)
328                                 | Data(List(x)),Data(List(y)) -> Data(Bool)
329                                 | Data(Cell), Data(Cell) -> Data(Bool)
330                                 | _,_ -> raise
331                    (Failure "Invalid␣Types␣used␣in␣an␣equality␣expression")
332                         end
333                    else if isArithmetic op then
334                         begin
335                                 match t1,t2 with
336                                 | Data(Integer),Data(Integer) -> Data(Integer)
337                                 | _,_ -> raise
338                    (Failure "Invalid␣Types␣used␣in␣an␣arithmetic␣expression")
339                         end
340                    else
341                         begin
342                                 match t1,t2 with
343                                 | Data(Integer),Data(Integer) -> Data(Bool)
344                                 | _,_ -> raise
345                    (Failure "Invalid␣Types␣used␣in␣a␣relational␣expression")
346                         end
347             | Funcall(fname,expr) ->
348             let fn = getFunc_fname fname env in
349                 begin
350                     match fn with
351                     | Func(f) -> f.reType
352                     | _ -> Ast.Data(Integer)
353                 end
354             | Paran(e) -> get_expr_type e func env
355             | Assign(_,_) -> Void
356             | Assoc(a,b) ->
357         if exists_id b func then
```

```
358                 begin
359                     match a with
360                     | Left -> Data(Cell)
361                     | Right -> Data(Cell)
362                     | Up -> Data(Cell)
363                     | Down -> Data(Cell)
364                     | Hleft -> Data(Bool)
365                     | Hright -> Data(Bool)
366                     | Htop -> Data(Bool)
367                     | Hbtm -> Data(Bool)
368                     | Empty -> Data(Bool)
369                     | Remove -> Data(Cell)
370                     | _ -> Void
371                 end
372                 else
373                     raise(Failure(b ^ "␣not␣defined␣"))
374                 | Visit(s) -> Data(Bool)
375                 | Target(b) ->
376                 if exists_id b func then
377                     Data(Bool)
378                 else
379                 raise(Failure("Invalid␣expression␣"^ b))
380                 | Src(b) ->
381             if exists_id b func then
382                 Data(Bool)
383             else
384                 raise(Failure("Invalid␣expression␣"^ b))
385                 | Pointer -> Data(Cell)
386                 | Loc(b) ->
387             if exists_id b func then
388                 Data(Cell)
389             else
390                 raise(Failure("Invalid␣expression␣"^ b))
391                 | Null -> Void
392
393 let rec get_expr_type_main e func env=
394         match e with
395                 | Id(s) -> Data(get_type_main func s)
396                 | Vars(s) -> Data(get_lit_type s)
397                 | BinOpr(op,e1,e2) ->
398             let t1 = get_expr_type_main e1 func env
399             and t2 = get_expr_type_main e2 func env in
400                         if isLogic op then
401                               begin
402                                       match t1,t2 with
```

```
403                                 | Data(Bool),Data(Bool) -> Data(Bool)
404                                 | _,_ -> raise
405                     (Failure "Invalid␣Types␣used␣in␣a␣Logical␣expression")
406                                 end
407                         else if isEql op then
408                                 begin
409                                         match t1,t2 with
410                                         | Data(Integer),Data(Integer) -> Data(Bool)
411                                         | Data(Bool),Data(Bool) -> Data(Bool)
412                                         | Data(List(x)),Data(List(y)) -> Data(Bool)
413                                         | Data(Cell), Data(Cell) -> Data(Bool)
414                                         | _,_ -> raise
415                     (Failure "Invalid␣Types␣used␣in␣an␣equality␣expression")
416                                 end
417                         else if isArithmetic op then
418                                 begin
419                                         match t1,t2 with
420                                         | Data(Integer),Data(Integer) -> Data(Integer)
421                                         | _,_ -> raise
422                     (Failure "Invalid␣Types␣used␣in␣an␣arithmetic␣expression")
423                                 end
424                         else
425                                 begin
426                                         match t1,t2 with
427                                         | Data(Integer),Data(Integer) -> Data(Bool)
428                                         | _,_ -> raise
429                     (Failure "Invalid␣Types␣used␣in␣a␣relational␣expression")
430                                 end
431             | Funcall(fname,expr) ->
432         let fn = get_main fname env in
433                 begin
434                     match fn with
435                     | Func(f) -> f.reType
436                     | _ -> Ast.Data(Integer)
437                 end
438             | Paran(e) -> get_expr_type_main e func env
439             | Assign(_,_) -> Void
440             | Assoc(a,b) -> if exists_id_main b func then
441                 begin
442                     match a with
443                     | Left -> Data(Cell)
444                     | Right -> Data(Cell)
445                     | Up -> Data(Cell)
446                     | Down -> Data(Cell)
447                     | Hleft -> Data(Bool)
```

```
448                     | Hright -> Data(Bool)
449                     | Htop -> Data(Bool)
450                     | Hbtm -> Data(Bool)
451                     | Empty -> Data(Bool)
452                     | Remove -> Data(Cell)
453                     | _ -> Void
454                 end
455                 else
456                     raise(Failure(b ^ "␣not␣defined␣"))
457                 | Visit(s) -> Data(Bool)
458                 | Target(b) -> if exists_id_main b func then
459                 Data(Bool)
460             else
461                 raise(Failure("Invalid␣expression␣"^ b))
462                 | Src(b) -> if exists_id_main b func then
463             Data(Bool)
464         else
465                 raise(Failure("Invalid␣expression␣"^ b))
466                 | Pointer -> Data(Cell)
467                 | Loc(b) -> if exists_id_main b func then
468             Data(Cell)
469         else
470                 raise(Failure("Invalid␣expression␣"^ b))
471                 | Null -> Void
472
473 (*Makes sure that the given arguments *)
474 (*in a function call match the function signature*)
475 (*fname of function being called*)
476 (*exprlist - list of expr in funcation call*)
477 (*env - the enviroment*)
478 let rec check_types_args cfunc env formalArgs = function
479         | [] -> true
480         | hd::tl -> begin
481         match List.hd formalArgs with
482             | FormalVar(dt,_) ->
483                 if string_of_rt (Data(dt)) =
484                     string_of_rt (get_expr_type hd cfunc env) then
485                     check_types_args cfunc env (List.tl formalArgs) tl
486                 else
487                     raise(Failure("Argument␣type␣mismatch"))
488         end
489
490 let rec check_types_argsmain cfunc env formalArgs = function
491         | [] -> true
492         | hd::tl -> begin
```

```
493         match List.hd formalArgs with
494         | FormalVar(dt,_) ->
495             if string_of_rt (Data(dt)) =
496                 string_of_rt (get_expr_type_main hd cfunc env) then
497                 check_types_argsmain cfunc env (List.tl formalArgs) tl
498             else
499                 raise(Failure("Argument type mismatch"))
500     end
501
502 let check_types fname exprlist cfunc env =
503         let func = getFunc_fname fname env in
504                 match func with
505                         Func(func) ->
506                                         if List.length exprlist =
507                         List.length func.formalArgs then
508     if check_types_args cfunc env func.formalArgs exprlist then
509         true
510     else
511         raise(Failure("Argument types do not match"))
512 else
513     raise
514     (Failure("Number of arguments do not match with function signature"))
515                 | _ -> true
516
517 let check_types_main fname exprlist cfunc env =
518         let func = getFunc_fname fname env in
519                 match func with
520                         Func(func) ->
521 if List.length exprlist = List.length func.formalArgs then
522     if check_types_argsmain cfunc env func.formalArgs exprlist then
523         true
524     else
525         raise(Failure("Argument types do not match"))
526 else
527     raise
528     (Failure("Number of arguments do not match with function signature"))
529                 | _ -> true
530
531 (*check if variable declation is valid*)
532 let valid_vdecl func env =
533     match func with
534         | Load(func) -> false
535         | Func(func) ->
536 let _ = List.map (function Define(dt,nm,value) ->
537 let e = "Invalid variable declaration for '" ^ nm ^ "' in function " ^ func.funcId ^ "\n" in
```

```
let be = e ^ "The␣only␣allowed␣values␣for␣initializing␣boolean␣variables␣are␣'true'␣and␣'false.'
match dt with
Cell -> if string_of_expr value = "AMLJava.current" then true else raise (Failure e)
| List(g) -> begin
match value with
| Vars(f) -> if is_list f then true else raise (Failure e)
| Id(f) -> if (get_type func f) = List(g) then true else raise (Failure e)
| Funcall(fname,list) -> let fn = (getFunc_fname fname) env in
begin
match fn with
| Func(f1) -> if (string_of_rt f1.reType) = (string_of_dt dt) then
if check_types fname list func env then
true
else
raise(Failure e)
else raise (Failure e)
| _ -> raise (Failure e)
end
           | _ -> false
       end
| Integer -> begin
               match value with
| Vars(f) -> begin
match f with
| Lit_Int(t) -> true
| _ -> raise (Failure e)
end
| Id(f) -> if (get_type func f) = Integer then true else raise (Failure e)
    | Funcall(fname,list) -> let fn = (getFunc_fname fname) env in
    begin
        match fn with
                   | Func(f1) -> if (string_of_rt f1.reType) = (string_of_dt dt) then
                       if check_types fname list func env then
                           true
    else
        raise(Failure e)
                       else raise (Failure e)
                   | _ -> raise (Failure e)
end
                   | _ -> false
    end
                   | Bool -> begin
                       match value with
| Vars(f) -> begin
match f with
```

```
583     | Lit_Bool(t) -> true
584     | _ -> raise (Failure be)
585                   end
586     | Id(f) -> if (get_type func f) = Bool then true else raise (Failure be)
587     | Funcall(fname,list) -> let fn = (getFunc_fname fname) env in
588     begin
589         match fn with
590                   | Func(f1) -> if (string_of_rt f1.reType) = (string_of_dt dt) then
591                       if check_types fname list func env then
592                           true
593     else
594         raise(Failure e)
595                       else raise (Failure e)
596                   | _ -> raise (Failure e)
597 end
598                   | _ -> false
599     end ) func.localVars
600     in
601 true
602 | Main(func) ->
603 let _ = List.map (function Define(dt,nm,value) ->
604 let e = "Invalid␣variable␣declaration␣for␣'" ^ nm ^ "'␣in␣" ^ func.mainId ^ "\n" in
605 let be = e
606 ^ "The␣only␣allowed␣values␣for␣initializing
607 boolean␣variables␣are␣'true'␣and␣'false.'␣\\n" in
608 match dt with
609 Cell -> if string_of_expr value = "AMLJava.current" then true else raise (Failure e)
610 | List(g) -> begin
611     match value with
612 | Vars(f) -> if is_list f then true else raise (Failure e)
613 | Id(f) -> if (get_type_main func f) = List(g) then true else raise (Failure e)
614 | Funcall(fname,list) -> let fn = (getFunc_fname fname) env in
615 begin
616 match fn with
617 | Func(f1) -> if (string_of_rt f1.reType) = (string_of_dt dt) then
618     if check_types_main fname list func env then
619         true
620 else
621 raise(Failure e)
622                       else raise (Failure e)
623                   | _ -> raise (Failure e)
624                   end
625                   | _ -> false
626 end
627                   | Integer -> begin
```

```
                    match value with
| Vars(f) -> begin
match f with
    | Lit_Int(t) -> true
    | _ -> raise (Failure e)
                end
    | Id(f) -> if (get_type_main func f) = Integer
    then true else raise (Failure e)
    | Funcall(fname,list) -> let fn =
        (getFunc_fname fname) env in
    begin
        match fn with
                | Func(f1) ->
    if (string_of_rt f1.reType) = (string_of_dt dt) then
    if check_types_main fname list func env then
                        true
    else
        raise(Failure e)
                    else raise (Failure e)
                | _ -> raise (Failure e)
end
                | _ -> false
    end
                | Bool -> begin
                    match value with
| Vars(f) -> begin
match f with
    | Lit_Bool(t) -> true
    | _ -> raise (Failure be)
                end
    | Id(f) -> if (get_type_main func f) = Bool then true else raise (Failure be)
    | Funcall(fname,list) -> let fn = (getFunc_fname fname) env in
    begin
        match fn with
                | Func(f1) -> if (string_of_rt f1.reType) = (string_of_dt dt) then
                    if check_types_main fname list func env then
                        true
    else
        raise(Failure e)
                    else raise (Failure e)
                | _ -> raise (Failure e)
end
                | _ -> false
    end ) func.mainVars
    in
```

```
673 true
674
675
676 (*Checks if the given statement list has return stmt last*)
677 let has_return_stmt list =
678     if List.length list = 0
679         then false
680         else match (List.hd (List.rev list)) with
681           Return(_) -> true
682                     | _ -> false
683
684 (*checks the given stmt list to determine if it has if/else statement that include a return valu
685 (*both the if body part AND the else part*)
686 let rec if_else_has_return_stmt stmt_list =
687     let if_stmts = List.filter (function If(_,_,_) -> true | _ -> false) stmt_list in
688     let rets = List.map (
689         function
690             If(_,s1,s2) ->
691                 begin match s1,s2 with
692     StmtBlk(lst1),StmtBlk(lst2) -> (has_return_stmt lst1
693                     || if_else_has_return_stmt lst1)
694     && (has_return_stmt lst2
695                     || if_else_has_return_stmt lst2)
696                     | _ -> raise(Failure("An␣unexpected␣error␣has␣occured."))
697                     (*shouldn't happen*)
698                     end
699                     | _ -> false
700     ) if_stmts in
701     List.fold_left (fun b v -> b || v) false rets
702
703     (*Checks that a return statement is present in the given function. *)
704 let has_return_stmt func =
705     let stmt_list = func.body in
706     if List.length stmt_list = 0
707             then false
708             else match List.hd (List.rev stmt_list) with
709               Return(e) ->
710                   raise(Failure("Return␣statement␣is␣not␣permitted␣in␣main␣method"))
711                      | _ -> false
712
713 let rec count_rets = function
714     | [] -> 0
715     | hd::tl -> begin
716         match hd with
717         | Return(_) -> 1 + count_rets tl
```

```
718         | _ -> count_rets tl
719               end
720
721 let has_multiple_ret func =
722     let count = count_rets func.statements in
723     if count > 1 then
724         raise(Failure("Multiple␣return␣statements"))
725     else
726         if count = 1 && if_else_has_return_stmt func.statements then
727             raise(Failure("Multiple␣return␣statements"))
728     else
729         false
730
731 let has_return func =
732     let stmt_list = func.statements in
733     if List.length stmt_list = 0
734           then false
735           else match List.hd (List.rev stmt_list) with
736             Return(e) -> true
737                                | _ -> false
738
739 let rec checkret_type func env ret = function
740     | [] -> true
741     | hd::tl -> begin
742         match hd with
743             | Return(e) -> if get_expr_type e func env = ret then
744                 checkret_type func env ret tl
745     else
746         raise(Failure("return␣type␣mismatch"))
747                   | _ -> checkret_type func env ret tl
748     end
749
750 let valid_return_stmt env = function
751     | Main(func) ->
752           let ifelse_has_return = if_else_has_return_stmt func.body in (*whether if/else block
753           let has_return = has_return_stmt func in (*if a function's last stmt is a return stmt
754           if has_return or ifelse_has_return
755                     then raise (Failure "Main␣function␣cannot␣have␣a␣return␣value")
756                     else true
757     | Func(func) ->
758           let ifelse_has_return = if_else_has_return_stmt func.statements in (*whether if/else
759           let has_return = has_return func in
760           let _ = has_multiple_ret func in  (*if a function's last stmt is a return stmt*)
761           if func.reType = Void then
762               if (has_return && not ifelse_has_return) or (not has_return && ifelse_has_return)
```

```
763                 raise(Failure("Invalid␣return␣expression␣in␣function␣" ^ func.funcId ^ ":␣func
764         else
765             true
766             else
767                 if (has_return && not ifelse_has_return) or (not has_return && ifelse_has_retu
768                     if checkret_type func env func.reType func.statements then
769                         true
770         else
771             raise(Failure("Expected␣return␣type␣:␣" ^ string_of_rt func.reType))
772                     else
773                         raise(Failure( func.funcId ^ "␣does␣not␣return␣any␣expression"))
774   | _ -> true
775
776         let rec valid_expr (func : Ast.func) expr env =
777             match expr with
778     Vars(_) -> true
779   | Id(s) -> if exists_id s func then true else raise (Failure ("Undeclared␣identifier␣" ^ s ^
780   | BinOpr(_,e1,e2) -> let exprtype = get_expr_type expr func env in
781   true
782   | Assign(id, e1) ->
783         if exists_id id func
784             then let dt = get_type func id and _ = valid_expr func e1 env and exprtype = get_e
785             match dt,exprtype with
786                 | Integer,Data(Integer) -> true
787                 | Bool,Data(Bool) -> true
788                 | List(x),Data(List(y)) -> if x = y then true else raise(Failure ("DataTypes␣d
789                 | List(x),Void -> (e1 = Null)
790                 | Cell, Data(Cell) -> true
791                 | _,_ -> raise(Failure ("DataTypes␣do␣not␣match␣up␣in␣an␣assignment␣expression
792                 else raise( Failure ("Undeclared␣identifier␣" ^ id ^ "␣is␣used" ))
793                 | Funcall(fname, exprlist) -> if isFunction_name fname env then
794                     let _has_valid_exprs = List.map (fun e -> valid_expr func e env) exprlist
795                     if check_types fname exprlist func env then (*check that the types match u
796                         true
797         else
798             raise(Failure("Actual␣and␣Formal␣Parameters␣do␣not␣match"))
799                     else
800                         raise(Failure ("Undefined␣function␣"^ fname ^"␣is␣used"))
801                 | Paran(e) -> valid_expr func e env
802   | Assoc(_,s) -> if exists_id s func then true else raise (Failure ("Undeclared␣identifier␣"
803   | Loc(s) -> if exists_id s func then
804     if (get_type func s = Cell) then
805         true
806   else
807     raise(Failure("Not␣a␣cell␣type"))
```

```
808         else
809             raise (Failure ("Undeclared␣identifier␣" ^ s ^ "␣is␣used"))
810     | Target(s) -> if exists_id s func then
811         if (get_type func s = Cell) then
812             true
813     else
814         raise(Failure("Not␣a␣cell␣type"))
815         else
816             raise (Failure ("Undeclared␣identifier␣" ^ s ^ "␣is␣used"))
817     | Visit(x) -> (valid_expr func x env) && (get_expr_type x func env = Data(Cell))
818     | _ -> false (*should not happen - added this to turn off compiler warnings about incomplete
819
820
821             let rec valid_expr_main (func : Ast.main) expr env =
822                 match expr with
823       Vars(_) -> true
824     | Id(s) -> if exists_id_main s func then true else raise (Failure ("Undeclared␣identifier␣"
825     | BinOpr(_,e1,e2) -> let exprtype = get_expr_type_main expr func env in
826     true
827     | Assign(id, e1) ->
828             if exists_id_main id func
829                 then let dt = get_type_main func id and _ = valid_expr_main func e1 env and exprty
830                 match dt,exprtype with
831                     | Integer,Data(Integer) -> true
832                     | Bool,Data(Bool) -> true
833                     | List(x),Data(List(y)) -> if x = y then true else raise(Failure ("DataTypes␣d
834                     | List(x),Void -> (e1 = Null)
835                     | Cell, Data(Cell) -> true
836                     | _,_ -> raise(Failure ("DataTypes␣do␣not␣match␣up␣in␣an␣assignment␣expression
837                    else raise( Failure ("Undeclared␣identifier␣" ^ id ^ "␣is␣used" ))
838                     | Funcall(fname, exprlist) -> if isFunction_name fname env then
839                       let _has_valid_exprs = List.map (fun e -> valid_expr_main func e env) expr
840                       if check_types_main fname exprlist func env then (*check that the types ma
841                           true
842             else
843                 raise(Failure("Actual␣and␣Formal␣Parameters␣do␣not␣match"))
844                       else
845                           raise(Failure ("Undefined␣function␣"^ fname ^"␣is␣used"))
846                     | Paran(e) -> valid_expr_main func e env
847     | Assoc(_,b) -> valid_expr_main func (Id(b)) env
848     | Loc(x) -> (valid_expr_main func (Id(x)) env) && (get_type_main func x = Cell)
849     | Target(x) -> (valid_expr_main func (Id(x)) env) && (get_type_main func x = Cell)
850     | Visit(x) -> (valid_expr_main func x env) && (get_expr_type_main x func env
= Data(Cell))
851     | _ -> false (*should not happen - added this to turn off compiler warnings about incomplete
```

```
852
853
854             let dup_letter_single func = function
855                 Define(_,mn,_) ->
856                     function c ->
857                         function Define(_,tn,_) ->
858                             if mn = tn
859                     then
860                         if c = 0
861                             then c+1
862                             else let e = "Duplicate␣variable␣declaration␣'"^ mn ^"'␣in␣function␣:␣"
in
863                             raise (Failure e) (*throw error on duplicate formal parameter.*)
864                         else c
865
866                         (*Checks the body of a function/main *)
867             let valid_body func env =
868                 match func with
869         | Func(func) ->
870                 let rec check_stmt =
871                     function
872                         StmtBlk(st_list) ->
873                             let _ = List.map(fun(x) -> check_stmt x) st_list in (*Check statements
874                 true
875         | Expr(st) ->
876                 if valid_expr func st env then
877                     true
878                             else
879                                 raise(Failure ("Invalid␣expression␣"^ string_of_expr st ^"␣in␣funct
880         | Return(st) -> (get_expr_type st func env) = func.reType
881                     | Display -> true
882                     | Revert -> true
883                     | Exit -> true
884                     | Print(e) -> valid_expr func e env
885                     | Move(e) -> (e >= 1) && ( e <= 4)
886                     | MoveTo(s) -> valid_expr func (Id(s)) env
887                     | ListAdd(id,ex) -> if (valid_expr func (Id(id)) env) && (valid_expr func ex e
888 begin
889 match get_expr_type ex func env with
890                             | Data(x) -> List(x) = get_type func id
891                             | _ -> false
892 end
893 else
894 false
895                             | If(predicate,stmt1,stmt2) ->
```

```
                                    let pred_type = get_expr_type predicate func env in
                                    let _vpred = (*Check predicate*)
                                        match pred_type with
| Data(Bool) -> true
| _ -> raise
(Failure("predicate␣expression␣must␣be␣a␣valid
boolean␣expression␣that␣evaluates␣to␣true/false"))
                                    in
if (check_stmt stmt1) && (check_stmt stmt2)
then true
else raise(Failure("Invalid␣expression␣used␣in␣if␣statement␣in␣function␣" ^ func.funcId ^ "\n"))
in
let _ = List.map(check_stmt) func.statements in
true
| Main(func) ->
       let rec check_stmt =
           function
               StmtBlk(st_list) ->
                   let _ = List.map(fun(x) -> check_stmt x) st_list in
(*Check statements in the block. Err will be thrown for an invalid stmt*)
       true
| Expr(st) ->
       if valid_expr_main func st env then
           true
else
raise(Failure ("Invalid␣expression␣"
^ string_of_expr st ^"␣in␣function␣" ^func.mainId ^ "\n"))
| Return(st) -> false
| Display -> true
| Revert -> true
| Exit -> true
| Print(e) -> valid_expr_main func e env
| Move(e) -> (e >= 1) && ( e <= 4)
| MoveTo(s) -> valid_expr_main func (Id(s)) env
| ListAdd(id,ex) -> if (valid_expr_main func (Id(id)) env)
&& (valid_expr_main func ex env) then
                       begin
    match get_expr_type_main ex func env with
                       | Data(x) -> List(x) = get_type_main func id
                       | _ -> false
end
                       else
       false
           | If(predicate,stmt1,stmt2) ->
                   let pred_type = get_expr_type_main predicate func env in
```

```
                    let _vpred = (*Check predicate*)
                        match pred_type with
                | Data(Bool) -> true
                | _ -> raise
                (Failure("predicate␣expression␣must␣be␣a␣valid
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣boolean␣expression␣that␣evaluates␣to␣true/false"))
                                                    in
            if (check_stmt stmt1) && (check_stmt stmt2)
            then true
                else raise
                (Failure("Invalid␣expression␣used␣in
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣if␣statement␣in␣function␣" ^ func.mainId ^ "\n"))
                        in
let _ = List.map(check_stmt) func.body in
true
                | _ -> true


            let cisDup_fp func =
                let isdup f = List.fold_left (isDup_fp_single func f) 0 func.formalArgs
    in let _ = List.map isdup func.formalArgs
                in false

let isDup_fp = function
    | Func(func) -> cisDup_fp func
    | _ -> true

let check_function f env =
    let dup_fname = isFunction f env in
    let dup_formals = isDup_fp f in
    let vlocals = (not (dup_vdecl f)) && (valid_vdecl f env) (*make sure that we've no dup varial
    let vbody = valid_body f env in
    let vret = valid_return_stmt env f in
    (*let _ = env.functions <- f :: env.functions (*add function name to environment *) in*)
(not dup_fname) && (not dup_formals) && vlocals && vbody &&vret

let check_main f env =
    let dup_fname = isFunction f env in
    let vlocals = (not (dup_vdecl f)) && (valid_vdecl f env) in
    let vbody = valid_body f env in
    let vret = valid_return_stmt env f in
    (*let _ = env.functions <- (f) :: env.functions (*add function name to environment *) in*)
(not dup_fname) && vlocals && vbody && vret

let valid_func env = function
```

```
986     Func(f) -> let afunc = Func(f) in check_function afunc env
987     | Main(f) -> let afunc = Main(f) in check_main afunc env
988     | Load(f) -> true
989 
990     (*Checks to make sure that the main function exists*)
991 let exists_main env =
992     if (isMain_name "main" env) then
993         if not (isFunction_name "main" env) then
994             true
995 else
996 raise(Failure("A␣generic␣function␣cannot␣be␣called␣'Main'"))
997 else raise(Failure("'main'␣does␣not␣exist!␣No␣Entry␣point␣to␣the␣program!"))
998 
999 let rec numLoad = function
1000 | [] -> 0
1001 | hd::tl -> begin
1002 match hd with
1003 | Load(s) -> 1 + numLoad tl
1004 | _ -> numLoad tl
1005 end
1006 
1007 let checkLoad list = begin
1008 match List.hd list with
1009     | Load(str) -> begin
1010         match List.hd (List.tl list) with
1011 | Main(fn) -> true
1012 | _ -> raise(Failure ("'main'␣must␣be␣after␣load"))
1013 end
1014 | _ -> raise(Failure("'load'␣must␣be␣at␣the␣start␣of␣the␣program"))
1015     end
1016 
1017 let check_program funclist =
1018 let (environ : env) = { functions = funclist} in
1019 let _loadchecker = numLoad funclist = 1 in
1020 let _loadmain = checkLoad funclist in
1021 let _dovalidation =
1022     List.map ( fun(f) -> valid_func environ f) funclist in
1023     (*Do the semantic analysis*)
1024 let _mainexists =
1025     exists_main environ (*ensure that a main function exists*) in
1026 let _ =
1027     print_endline
1028     "\nSemantic␣analysis␣successfully␣completed.\nCompiling...\n" in
1029 true
```

8.5 Top-Level Command Line Interface

Listing 5: toplevel.ml

```
type action = Ast | Compile | SA

(* Custom exceptions. *)
exception NoInputFile
exception InvalidArgument

(* Compiler usage instructions. *)
let usage = Printf.sprintf "Usage: aml [-a|-s|-c] SOURCE_FILE"

(* Get the name of the program from the file name. *)
let get_prog_name source_file_path =
       let split_path =
       (Str.split (Str.regexp_string "/") source_file_path) in
       let file_name =
       List.nth split_path ((List.length split_path) - 1) in
       let split_name =
       (Str.split (Str.regexp_string ".") file_name) in
              List.nth split_name ((List.length split_name) - 2)

(* Main entry point *)
let _ =
       try
              let action = if Array.length Sys.argv > 1 then
                     match Sys.argv.(1) with
                                   | "-a" -> Ast
                                   | "-s" -> SA (*semantic analysis testing*)
                                   | "-c" -> Compile
                                   | _ -> raise InvalidArgument
                     else raise InvalidArgument in
              let prog_name =
                     if Array.length Sys.argv > 2 then
                            get_prog_name Sys.argv.(2)
                     else raise NoInputFile in
              let input_chan = open_in Sys.argv.(2) in
       let lexbuf = Lexing.from_channel input_chan in
       let reversed_program = Parser.program Scanner.token lexbuf in
              let program = List.rev reversed_program in
              match action with
                     | Ast ->
              let listing =
                  Ast.string_of_program program prog_name in
              Printf.printf "%s" listing
```

```
43                      | SA -> ignore (Sast.check_program program);
44                      | Compile ->
45                if Sast.check_program program then
46                    let listing = Compile.translate program prog_name in
47                       print_string listing
48                    else raise(Failure("\nInvalid program.\n"))
49          with
50                  | InvalidArgument ->
51                  ignore (Printf.printf "InvalidArgument\n %s\n" usage)
52                  | NoInputFile ->
53                  ignore (Printf.printf
54                  "The second argument must be the name of an aml file\n %s\n"
55                  usage)
```

8.6 Java Standard Library

Listing 6: AMLJava.java

```
1  import java.util.*;
2  import java.io.*;
3  import javax.swing.*;
4  import java.awt.*;
5  import javax.swing.text.*;
6
7  /*
8   * Standard Library of Java code for amL
9   * Programming Languages and Translators, Fall 2012
10  *
11  * Sriramkumar Balasubramanian (sb3457)
12  * Evan Drewry (ewd2106)
13  * Timothy Giel (tkg2104)
14  * Nikhil Helferty (nh2407)
15  *
16  * Includes functions for the bot to move around the maze, or
17  * obtain information about its surrounding environment.
18  *
19  * Includes a rudimentary Swing GUI for the user to see
20  * what the bot does when the program is run.
21  *
22  * Makes use of the custom Cell object to represent a given cell of the maze.
23  *
24  */
25 public class AMLJava extends JFrame
26 {
27     static int width; // width of the maze
28     static int height; // height of the maze
```

```
29      static Cell current; // the current cell the bot is at
30
31      // 2D representation of the maze - maze[row][col] is Cell at
32      //row row, column col, with top left being 0, 0
33      static Cell [][] maze;
34
35      // is a stack of Cells - consecutive moves that have been done,
36      //not counting "reverted" moves - used to backtrack in revert()
37      static Stack<Cell> moves;
38      static JTextArea textArea;
39
40      // build the representation of the maze from the text file
41      public static void buildMaze(String mazeFileName) {
42          if (mazeFileName.equals("random")) randomGenMaze();
43          else {
44              File mazeFile = new File(mazeFileName);
45              try {
46                  Scanner scan = new Scanner(mazeFile);
47                  height = scan.nextInt();
48                  width = scan.nextInt();
49                  maze = new Cell[height][width];
50                  for (int row = 0; row < height; row++) {
51                      for (int col = 0; col < width; col++) {
52                          int temp = scan.nextInt();
53                          maze[row][col] = new Cell(temp, row, col);
54                          if (temp == 2) current = maze[row][col];
55                      }
56                  }
57                  scan.close();
58              }
59              catch (FileNotFoundException e) {
60                  System.out.println("File␣Not␣Found");
61                  return;
62              }
63          }
64          moves = new Stack<Cell>();
65          new AMLJava(); // initiliase the Swing GUI
66      }
67
68      public static void randomGenMaze() {
69          width = (int)(Math.random() * 20) + 5;
70          height = (int)(Math.random() * 20) + 5;
71          maze = new Cell[height][width];
72          int targetRow = (int)(Math.random()*(height-1));
73          int targetCol = (int)(Math.random()*(width-1));
```

```
74          maze[targetRow][targetCol] = new Cell(3, targetRow, targetCol);
75          // randomly generate a target cell
76          int stepLength = (int)((Math.random()*width)/2
77                  + (Math.random()*height)/2) + width/2 + height/2;
78                  // determine how many steps it'll iterate back from the
79                  // target until a start cell is picked
80          boolean pathGenerated = false;
81          while (!pathGenerated) pathGenerated = pathGen(maze[targetRow][targetCol], stepLength);
82          // start point has been picked - for remaining null cells,
83          //randomly pick between a step or a hole
84          for (int row = 0; row < height; row++) {
85              for (int col = 0; col < width; col++) {
86                  if (maze[row][col] == null) {
87                      double prob = Math.random();
88                      if (prob < .5) maze[row][col] = new Cell(1, row, col);
89                      else maze[row][col] = new Cell(0, row, col);
90                  }
91              }
92          }
93      }
94
95      public static boolean pathGen(Cell c, int steps) {
96          int cRow = c.getRow();
97          int cCol = c.getCol();
98          if (steps == 0) { // make this cell the starting one
99              current = maze[cRow][cCol] = new Cell(2, cRow, cCol);
100             return true;
101         }
102         LinkedList<String> dirs = new LinkedList<String>();
103         dirs.add("up");
104         dirs.add("down");
105         dirs.add("left");
106         dirs.add("right");
107         if (cRow == 0) dirs.remove("up");
108         if (cRow == (height-1)) dirs.remove("down");
109         if (cCol == 0) dirs.remove("left");
110         if (cCol == (width-1)) dirs.remove("right");
111         if (dirs.size() == 0) return false;
112         String randDir = dirs.get((int)(Math.random() * dirs.size()));
113         // pick a random direction from the current cell
114         if (randDir.equals("up")) {
115             if (maze[cRow-1][cCol] == null) maze[cRow-1][cCol] = new Cell(1, cRow-1, cCol);
116             steps--;
117             return pathGen(maze[cRow-1][cCol], steps);
118         }
```

```
119         else if (randDir.equals("down")) {
120             if (maze[cRow+1][cCol] == null) maze[cRow+1][cCol] = new Cell(1, cRow+1, cCol);
121             steps--;
122             return pathGen(maze[cRow+1][cCol], steps);
123         }
124         else if (randDir.equals("left")) {
125             if (maze[cRow][cCol-1] == null) maze[cRow][cCol-1] = new Cell(1, cRow, cCol-1);
126             steps--;
127             return pathGen(maze[cRow][cCol-1], steps);
128         }
129         else {
130             if (maze[cRow][cCol+1] == null) maze[cRow][cCol+1] = new Cell(1, cRow, cCol+1);
131             steps--;
132             return pathGen(maze[cRow][cCol+1], steps);
133         }
134 
135     }
136 
137 
138     // creates the Swing GUI
139     public AMLJava() {
140         setDefaultCloseOperation(EXIT_ON_CLOSE);
141         setTitle("Maze");
142         JPanel mazePanel = new JPanel(new GridLayout(height, width));
143         // height bounds # of rows, width bounds # of columns
144         for(int row = 0; row < height; row++) {
145             for (int col = 0; col < width; col++) {
146                 mazePanel.add(maze[row][col]); // add the cell to the maze
147             }
148         }
149         add(mazePanel, BorderLayout.CENTER);
150         // now add the text area to display moves explicitly
151         textArea = new JTextArea(5, 1);
152         textArea.setEditable(false);
153         textArea.setFont(new Font("Times␣New␣Roman", Font.PLAIN, 16));
154         DefaultCaret caret = (DefaultCaret)textArea.getCaret();
155         caret.setUpdatePolicy(DefaultCaret.ALWAYS_UPDATE);
156         JPanel textPanel = new JPanel();
157         textPanel.add(textArea);
158         textPanel.setBackground(Color.WHITE);
159         JScrollPane scrollPane = new JScrollPane(textPanel);
160         scrollPane.setPreferredSize(new Dimension(500, 100));
161         scrollPane.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
162         add(scrollPane, BorderLayout.SOUTH);
163         pack();
```

```
164         setVisible(true);
165     }
166 
167     // moves the bot up from its current position
168     // if successful returns true, otherwise false
169     public static boolean move_U() {
170         if (hasTop()) {
171             move(maze[current.getRow()-1][current.getCol()]);
172             textArea.append("Bot␣moved␣UP\n");
173             if (current.isTarget()) textArea.append("Bot␣moved␣on␣to␣a␣target!\n");
174             return true;
175         }
176         else {
177             textArea.append("Bot␣failed␣to␣move␣UP\n");
178             return false;
179         }
180     }
181 
182     public static boolean move_D(){
183         if (hasBottom()) {
184             move(maze[current.getRow()+1][current.getCol()]);
185             textArea.append("Bot␣moved␣DOWN\n");
186             if (current.isTarget()) textArea.append("Bot␣moved␣on␣to␣a␣target!\n");
187             return true;
188         }
189         else {
190             textArea.append("Bot␣failed␣to␣move␣DOWN\n");
191             return false;
192         }
193     }
194 
195     public static boolean move_L() {
196         if (hasLeft()) {
197             move(maze[current.getRow()][current.getCol()-1]);
198             textArea.append("Bot␣moved␣LEFT\n");
199             if (current.isTarget()) textArea.append("Bot␣moved␣on␣to␣a␣target!\n");
200             return true;
201         }
202         else {
203             textArea.append("Bot␣failed␣to␣move␣LEFT\n");
204             return false;
205         }
206     }
207 
208     public static boolean move_R() {
```

```
209         if (hasRight()) {
210             move(maze[current.getRow()][current.getCol()+1]);
211             textArea.append("Bot␣moved␣RIGHT\n");
212             if (current.isTarget()) textArea.append("Bot␣moved␣on␣to␣a␣target!\n");
213             return true;
214         }
215         else {
216             textArea.append("Bot␣failed␣to␣move␣RIGHT\n");
217             return false;
218         }
219     }
220 
221     // private move function eliminating duplicate code
222     // moves the bot from "current" cell to next cell in parameter,
223     //updates the GUI accordingly
224     public static void move(Cell next) {
225         moves.push(current);
226         try {
227             Thread.sleep(500);
228         }
229         catch (InterruptedException e) { }
230         current.setText("");
231         if (current.isTarget()) current.setText("TARGET");
232         if (current.isSource()) current.setText("START");
233         current = next;
234         current.visited();
235         current.setText("BOT");
236     }
237 
238     // returns true if there is a cell the bot can go on above it
239     // false otherwise
240     public static boolean hasTop() {
241         if (current.getRow() > 0) { // if not at top
242             if (maze[current.getRow()-1][current.getCol()].getValue() != 0)
243                 return true;
244             // if not a "hole"
245         }
246         return false;
247     }
248 
249     public static boolean hasBottom() {
250         if (current.getRow() < (height-1)) {
251             if (maze[current.getRow()+1][current.getCol()].getValue() != 0)
252                 return true;
253         }
```

```
254         return false;
255     }
256 
257     public static boolean hasLeft() {
258         if (current.getCol() > 0) {
259             if (maze[current.getRow()][current.getCol()-1].getValue() != 0)
260                 return true;
261         }
262         return false;
263     }
264 
265     public static boolean hasRight() {
266         if (current.getCol() < (width-1)) {
267             if (maze[current.getRow()][current.getCol()+1].getValue() != 0)
268                 return true;
269         }
270         return false;
271     }
272 
273     // returns the cell to the right of the bot's current position if it exists
274     public static Cell right() {
275         if (hasRight()) {
276             Cell c = maze[current.getRow()][current.getCol()+1];
277             return c;
278         }
279         else return null;
280     }
281 
282     public static Cell up() {
283         if (hasTop()) {
284             Cell c = maze[current.getRow()-1][current.getCol()];
285             return c;
286         }
287         else return null;
288     }
289 
290     public static Cell down() {
291         if (hasBottom()) {
292             Cell c = maze[current.getRow()+1][current.getCol()];
293             return c;
294         }
295         else return null;
296     }
297 
298     public static Cell left() {
```

```
299         if (hasLeft()) {
300             Cell c = maze[current.getRow()][current.getCol()-1];
301             return c;
302         }
303         else return null;
304     }
305 
306     // returns whether or not the cell at row, col has been visited
307     public static boolean visit(int row, int col) {
308         return maze[row][col].getVisited();
309     }
310 
311     // overloaded version of visit that instead accepts a single integer (cell ID)
312     // cell ID is calculated as follows = (width of maze) * (cell row) + (cell column)
313     public static boolean visit(int id) {
314         return visit(id/width, id%width);
315     }
316 
317     // "reverts" the previous move if possible (backtracks), returns true
318     // if no moves committed returns false
319     public static boolean revert() {
320         if (moves.empty()) {
321             textArea.append("Bot␣failed␣to␣REVERT␣(at␣starting␣position)\n");
322             return false; // no moves executed!
323         }
324         else {
325             try {
326                 Thread.sleep(500);
327             }
328             catch (InterruptedException e) { }
329             if (current.isTarget()) current.setText("TARGET");
330             else current.setText("");
331             current = moves.pop();
332             current.setText("BOT");
333             textArea.append("Bot␣BACKTRACKED\n");
334             return true;
335         }
336 
337 
338 }
```

Listing 7: Cell.java

```
1 import javax.swing.*;
2 import java.awt.*;
3 
```

```
4
5  /*
6   * Cell object written for amL
7   *
8   * Programming Languages and Translators, Fall 2012
9   *
10  * Sriramkumar Balasubramanian (sb3457)
11  * Evan Drewry (ewd2106)
12  * Timothy Giel (tkg2104)
13  * Nikhil Helferty (nh2407)
14  *
15  * Includes where the cell is, whether it has been visited,
16  * the "value" of the cell (is at walkable, is it a target, etc.),
17  * as well as information about displaying it in the Swing GUI.
18  *
19  */
20 public class Cell extends JLabel
21 {
22     private int row; // the row of the cell (top left is row 0, column 0)
23     private int column; // the column of the cell
24
25     // value of the cell: 0 if spot is a "hole", 1 if walkable,
26     // 2 if start point, 3 if target
27     private int value;
28     private boolean visited; // whether or not the bot has visited this point
29
30     public Cell(int value, int r, int c)
31     {
32         setHorizontalAlignment(JLabel.CENTER);
33         setFont(new Font("Times␣New␣Roman", Font.PLAIN, 24));
34         setBorder(BorderFactory.createLineBorder(Color.BLACK));
35         if (value == 2) {
36             visited = true;
37             setText("BOT");
38         }
39         else visited = false;
40         if (value == 3) setText("TARGET");
41         if (value == 0) setBackground(Color.BLACK);
42         setOpaque(true);
43         setPreferredSize(new Dimension(120, 120));
44         this.value = value;
45         row = r;
46         column = c;
47     }
48
```

```
49      // is this cell the target for the bot?
50      public boolean isTarget() {
51          if (value == 3) return true;
52          else return false;
53      }
54
55      // is this the source (start point of the bot)
56      public boolean isSource() {
57          if (value == 2) return true;
58          else return false;
59      }
60
61      // returns the unique integer ID of the cell
62      // unique ID calculated as follows:
63      // (number columns) * (row of cell) + column of cell
64      // note that it will not work if AMLJava is not running successfully
65      //(this should not be a problem)
66      public int get_Loc() {
67          return AMLJava.width * (row) + column;
68      }
69
70      // getter functions
71      public int getRow() { return row; }
72      public int getCol() { return column; }
73      public int getValue() { return value; }
74      public boolean getVisited() { return visited; }
75
76      public void visited() { visited = true; } // set visited to true
77  }
```

Listing 8: List.java

```
1  import java.util.*;
2
3  public class List extends LinkedList
4  {
5
6
7      public List(Object [] arr)
8      {
9          super();
10         for (int i = 0; i < arr.length; i++) add(arr[i]);
11     }
12
13 }
```

8.7 Test Suite

Listing 9: test-base

```
#!/bin/bash

function info() { echo -e "\033[00;32m[INFO] $1\033[00m"; }

function error() { echo -e "\033[00;31m[ERROR] $1\033[00m"; }

function do_test() {
        TEST_NAME=`basename $1 .test`
        TEST_SRC=${TEST_NAME}.aml
        COMPILE_ONLY=false
        GUITEST=false

        . ${TEST_NAME}.test

        if [ ! -f "$TEST_SRC" ]; then
            error "Source file '$TEST_SRC' not found."
            return 1
        fi

        compile $TEST_NAME

        if $COMPILE_ONLY; then
            checkoutput $TEST_NAME
            return $?
        fi

        if [ ! -f "./bin/$TEST_NAME.class" ]; then
            error "Binary file 'bin/$TEST_NAME.class' not found."
            return 1
        fi

        run $TEST_NAME
        checkoutput $TEST_NAME
        return $?
}

function run_all() {
    for test in *.test
    do
        do_test $test
    done
}
```

```
43
44 function compile() {
45         echo "Compiling␣'$1'..."
46     if [ ! -d "bin" ]; then
47         mkdir bin
48     fi
49     cd bin
50     copydependencies
51         ../$AML_BINARY -c ../$1.aml >log_stdout 2>log_stderr
52     if [ -f "$1.java" ]; then
53         javac -classpath ../../ ./$1.java
54     fi
55     cd ..
56 }
57
58 function run() {
59         echo "Running␣'$1'..."
60     cd bin
61     if $GUITEST; then
62         java $1 >log_stdout 2>log_stderr
63     else
64         java $1 >log_stdout 2>log_stderr &
65         sleep 3
66         kill $!
67     fi
68     cd ..
69 }
70
71 function checkoutput() {
72     . $1.test
73
74     if [ -f bin/log_stdout ]; then
75         ACTUAL_OUT=‘cat bin/log_stdout‘
76     else
77         ACTUAL_OUT=""
78     fi
79
80     if [ -f bin/log_stderr ]; then
81         ACTUAL_ERR=‘cat bin/log_stderr‘
82     else
83         ACTUAL_ERR=""
84     fi
85
86     rm log_stdout &> /dev/null
87     rm log_stderr &> /dev/null
```

```
88
89
90      if [ "$OUT" = "$ACTUAL_OUT" ] && [ "$ERR" = "$ACTUAL_ERR" ]; then
91          info "$1␣PASSED"
92          return 0
93      else
94          echo expected err: "$ERR"
95          echo actual err: "$ACTUAL_ERR"
96          echo expected out: "$OUT"
97          echo actual out: "$ACTUAL_OUT"
98          error "$1␣FAILED"
99          return 1
100     fi
101 }
102
103 function clean() {
104     rm -rf bin
105 }
106
107 function copydependencies() {
108     if [ ! -f AMLJava.class ] || [ ! -f Cell.class ] || [ ! -f List.class ]; then
109         cp ../../AMLJava.java .
110         cp ../../Cell.java .
111         cp ../../List.java .
112         javac AMLJava.java
113     fi
114     if [ ! -f maze.txt ]; then
115         cp ../maze.txt .
116     fi
117 }
```

Listing 10: run-all-tests

```
1 #!/bin/bash
2
3 AML_BINARY=$1
4 if [ ! -f "$1" ]; then
5     echo "Usage:␣run-all-tests␣<AML␣BINARY>"
6     exit 1
7 fi
8 . test-base
9 run_all
10 exit $?
```

Listing 11: run-test

```
#!/bin/bash

if [ ! -f "$1" ] || [ $# -lt 2 ]; then
    echo "Usage: run-test <AML-BINARY> <TEST-NAME>"
    exit 1
fi

. test-base
AML_BINARY=$1

shift
while [ $1 ]
do
    do_test $1.test
    shift
done
exit $?
```

Listing 12: ./bfs.aml

```
#load<maze>

main():void{
        list<cell> toGo := <[]>;
        cell node := (CPos);
        toGo.add(node);
        BFS(toGo);
}

function BFS (list<cell> toGo):void{
        cell node := (CPos);
        if(NOT toGo.isEmpty()){
                node := toGo.remove();
                if (isTarget(node)){
                        move_To(node);
                        toGo.clear();
                        exit();
                };
                if (visited(node) AND NOT isSource(node)){
                        BFS(toGo);
                }
                else{
                        move_To(node);
                        addToGo(node, toGo);
                        revert();
                        BFS(toGo);
```

```
                }
        };

}

function addToGo(cell node, list<cell> toGo):void{
        cell tempNode := (CPos);
        if (node.hasleft()){
                tempNode := node.left();
                toGo.add(tempNode);
        };
        if (node.hastop()){
                tempNode := node.up();
                toGo.add(tempNode);
        };
        if (node.hasright()){
                tempNode := node.right();
                toGo.add(tempNode);
        };
        if (node.hasbottom()){
                tempNode := node.down();
                toGo.add(tempNode);
        };
}
```

Listing 13: ./bfs.test

```
#!/bin/bash

DESC="bfs algorithm"
OUT=""
ERR=""
GUITEST=true
```

Listing 14: ./binop_divide.aml

```
#load<maze>

main():void
{
    print(8/2);
}
```

Listing 15: ./binop_divide.test

```
#!/bin/bash

```

```
DESC="division␣binop"
OUT="4"
ERR=""
```

Listing 16: ./binop_minus.aml

```
#load<maze>

main():void
{
    print(7 - 9);
}
```

Listing 17: ./binop_minus.test

```
#!/bin/bash

DESC="subtraction␣binop"
OUT="-2"
ERR=""
```

Listing 18: ./binop_modulo.aml

```
#load<maze>

main():void
{
    print(9%7);
}
```

Listing 19: ./binop_modulo.test

```
##!/bin/bash

DESC="division␣binop"
OUT="2"
ERR=""
```

Listing 20: ./binop_multiply.aml

```
#load<maze>

main():void
{
    print(7*9);
}
```

Listing 21: ./binop_multiply.test

```
#!/bin/bash

DESC="multiplication␣binop"
OUT="63"
ERR=""
```

Listing 22: ./binop_plus.aml

```
#load<maze>

main():void
{
    print(7+9);
}
```

Listing 23: ./binop_plus.test

```
#!/bin/bash

DESC="addition␣binop"
OUT="16"
ERR=""
```

Listing 24: ./binop_power.aml

```
#load<maze>

main():void
{
    print(2^4);
}
```

Listing 25: ./binop_power.test

```
#!/bin/bash

DESC="exponentiation␣binop"
OUT="16.0"
ERR=""
```

Listing 26: ./bool_and.aml

```
#load<maze>

main():void
{
```

```
    print(true AND false);
    print(true AND true);
    print(false AND false);
}
```

Listing 27: ./bool_and.test

```
#!/bin/bash

DESC="AND␣binop"
OUT=$'false\ntrue\nfalse'
ERR=""
```

Listing 28: ./boolean_literal.aml

```
#load<maze>

main():void
{
    print(true);
    print(false);
}
```

Listing 29: ./boolean_literal.test

```
#!/bin/bash

DESC="boolean␣literals"
OUT=$'true\nfalse'
ERR=""
```

Listing 30: ./bool_eq.aml

```
#load<maze>

main():void
{
    print(7 = 9);
    print(7=7);
}
```

Listing 31: ./bool_eq.test

```
#!/bin/bash

DESC="==␣binop"
OUT=$'false\ntrue'
ERR=""
```

Listing 32: ./bool_gt.aml

```
#load<maze>

main():void
{
    print(7>9);
    print(7>7);
    print(9>7);
}
```

Listing 33: ./bool_gte.aml

```
#load<maze>

main():void
{
    print(7>=9);
    print(7>=7);
    print(9>=7);
}
```

Listing 34: ./bool_gte.test

```
#!/bin/bash

DESC="gte␣binop"
OUT=$'false\ntrue\ntrue'
ERR=""
```

Listing 35: ./bool_gt.test

```
#!/bin/bash

DESC="gt␣binop"
OUT=$'false\nfalse\ntrue'
ERR=""
```

Listing 36: ./bool_lt.aml

```
#load<maze>

main():void
{
    print(7<9);
    print(7<7);
    print(9<7);
}
```

Listing 37: ./bool_lte.aml

```
#load<maze>

main():void
{
    print(7<=9);
    print(7<=7);
    print(9<=7);
}
```

Listing 38: ./bool_lte.test

```
#!/bin/bash

DESC="lte␣binop"
OUT=$'true\ntrue\nfalse'
ERR=""
```

Listing 39: ./bool_lt.test

```
#!/bin/bash

DESC="lt␣binop"
OUT=$'true\nfalse\nfalse'
ERR=""
```

Listing 40: ./bool_ne.aml

```
#load<maze>

main():void
{
    print(7~=9);
    print(7~=7);
}
```

Listing 41: ./bool_ne.test

```
#!/bin/bash

DESC="ne␣binop"
OUT=$'true\nfalse'
ERR=""
```

Listing 42: ./bool_not.aml

```
#load<maze>
```

```

main():void
{
    print(NOT false);
    print(NOT true);
}
```

Listing 43: ./bool_not.test

```
#!/bin/bash

DESC="NOT␣op"
OUT=$'true\nfalse'
ERR=""
```

Listing 44: ./bool_or.aml

```
#load<maze>

main():void
{
    print(true OR false);
    print(true OR true);
    print(false OR false);
}
```

Listing 45: ./bool_or.test

```
#!/bin/bash

DESC="OR␣binop"
OUT=$'true\ntrue\nfalse'
ERR=""
```

Listing 46: ./clean-tests

```
#!/bash/bin

. test-base
clean
```

Listing 47: ./cpos.aml

```
#load<maze>

main():void
{
```

```
    cell i := (CPos);
    print(get_Loc(i));
}
```

Listing 48: ./cpos.test

```
#!/bin/bash

DESC="CPos␣variable"
OUT="8"
ERR=""
```

Listing 49: ./decl_boolean.aml

```
#load<maze>

main():void
{
    Boolean i;
}
```

Listing 50: ./decl_boolean.test

```
#!/bin/bash

DESC="boolean␣decl"
OUT="syntax␣error"
ERR="Fatal␣error:␣exception␣Parsing.Parse_error"
COMPILE_ONLY=true
```

Listing 51: ./decl_cell.aml

```
#load<maze>

main():void
{
    Cell i;
}
```

Listing 52: ./decl_cell.test

```
#!/bin/bash

DESC="cell␣decl"
OUT="syntax␣error"
ERR="Fatal␣error:␣exception␣Parsing.Parse_error"
COMPILE_ONLY=true
```

Listing 53: ./decl_integer.aml

```
#load<maze>

main():void
{
    Integer i;
}
```

Listing 54: ./decl_integer.test

```
#!/bin/bash

DESC="integer␣decl"
OUT="syntax␣error"
ERR="Fatal␣error:␣exception␣Parsing.Parse_error"
COMPILE_ONLY=true
```

Listing 55: ./decl_list.aml

```
#load<maze>

main():void
{
    List i;
}
```

Listing 56: ./decl_list.test

```
#!/bin/bash

DESC="list␣decl"
OUT="syntax␣error"
ERR="Fatal␣error:␣exception␣Parsing.Parse_error"
COMPILE_ONLY=true
```

Listing 57: ./dfs.aml

```
#load<maze>

main():void{
        DFS();
}

function DFS():void{
        cell node := (CPos);

        if (isTarget(node)){
```

```
                exit();
        };

        if(myvisited(node)){
                DFS();
        }
        else{
                if (isSource(node)){
                        exit();
                };

                revert();
                DFS();
        }
}


function myvisited(cell node):bool{
        if (node.hasleft() AND NOT visited(node.left())){
                move_L();
        }
        else{
                if(node.hastop() AND NOT visited(node.up())){
                        move_U();
                }
                else{
                        if (node.hasright() AND NOT visited(node.right())) {
                                move_R();
                        }
                        else{
                                if(node.hasbottom() AND NOT visited(node.down())){
                                        move_D();
                                }
                                else{
                                        return false;
                                }
                        }
                }
        }
        return true;
}
```

Listing 58: ./dfs.test

```
#!/bin/bash

```

```
DESC="dfs algorithm"
OUT=""
ERR=""
GUITEST=true
```

Listing 59: ./divide_by_zero.aml

```
#load<maze>

main():void
{
    print(4/0);
}
```

Listing 60: ./divide_by_zero.test

```
#!/bin/bash

DESC="divide by zero"
OUT=""
ERR='Exception in thread "main" java.lang.ArithmeticException: / by zero
        at divide_by_zero.main(divide_by_zero.java:6)'
```

Listing 61: ./empty_program.aml

Listing 62: ./empty_program.test

```
#!/bin/bash

DESC="empty file"
OUT=""
ERR='Fatal error: exception Failure("hd")'
COMPILE_ONLY=true
```

Listing 63: ./factorial.aml

```
#load-random

main():void{
        integer n := 10;
        print(fac(n));
}

function fac(integer n):integer{
        if(n=1){
                return 1;
```

```
        }
        else{
                return n*fac(n - 1);
        }

}
```

Listing 64: ./factorial.test

```
#!/bin/bash

DESC="factorial␣algorithm"
OUT="3628800"
ERR=""
```

Listing 65: ./function_after_main.aml

```
#load<maze>

main():void
{
    func();
}

function func():void
{
    print(true);
}
```

Listing 66: ./function_after_main.test

```
#!/bin/bash

DESC="function␣after␣main"
OUT="true"
ERR=""
```

Listing 67: ./function_before_main.aml

```
#load<maze>

function func():void
{
    print(true);
}

main():void
```

```
{
    func();
}
```

Listing 68: ./function_before_main.test

```
#!/bin/bash

DESC="function before main"
OUT=""
ERR='Fatal error: exception Failure("'"'main'"' must be after load")'
COMPILE_ONLY=true
```

Listing 69: ./function_bool.aml

```
#load<maze>

main():void
{
    print(func());
}

function func():bool
{
    return true;
}
```

Listing 70: ./function_bool.test

```
#!/bin/bash

DESC="boolean function"
OUT="true"
ERR=""
```

Listing 71: ./function_int.aml

```
#load<maze>

main():void
{
    print(func());
}

function func():integer
{
    return 5;
}
```

Listing 72: ./function_int.test

```
#!/bin/bash

DESC="int function"
OUT="5"
ERR=""
```

Listing 73: ./gcd.aml

```
#load<maze>
main():void
{
    integer x := gcd(7,49);
    print(x);
    exit();
}

function gcd(integer n, integer m):integer
{
    if(n = m) {
        return n;
    } else {
        if (n > m) {
            return gcd(n - m, m);
        } else {
            return gcd(m - n,n);
        }
    }
}
```

Listing 74: ./gcd.test

```
#!/bin/bash

DESC="gcd algorithm"
OUT="7"
ERR=""
```

Listing 75: ./init_boolean.aml

```
#load<maze>

main():void
{
    bool i := true;
    print(i);
}
```

Listing 76: ./init_boolean.test

```
#!/bin/bash

DESC="initialize␣a␣boolean"
OUT="true"
ERR=""
```

Listing 77: ./init_cell.aml

```
#load<maze>

main():void
{
    cell i := (CPos);
    get_Loc(i);
}
```

Listing 78: ./init_cell.test

```
#!/bin/bash

DESC="initialize␣a␣cell"
OUT=""
ERR=""
```

Listing 79: ./init_integer.aml

```
#load<maze>

main():void
{
    integer i := 5;
    print(i);
}
```

Listing 80: ./init_integer.test

```
#!/bin/bash

DESC="initialize␣an␣int"
OUT="5"
ERR=""
```

Listing 81: ./init_list.aml

```
#load<maze>

```

```
main():void
{
    list<integer> i := <[1,2,3]>;
    print(i);
}
```

Listing 82: ./init_list.test

```
#!/bin/bash

DESC="initialize␣a␣list"
OUT="[1,␣2,␣3]"
ERR=""
```

Listing 83: ./integer_literal.aml

```
#load<maze>

main():void
{
    print(7);
}
```

Listing 84: ./integer_literal.test

```
#!/bin/bash

DESC="integer␣literal"
OUT="7"
ERR=""
```

Listing 85: ./invalid_return_type.aml

```
#load<maze>

main():void
{
}

foo():int
{
    return 1;
}
```

Listing 86: ./invalid_return_type.test

```
#!/bin/bash
```

```

DESC="invalid type"
OUT="syntax error"
ERR="Fatal error: exception Parsing.Parse_error"
COMPILE_ONLY=true
```

Listing 87: ./keyword_as_identifier2.aml

```
#load<maze>

main():void
{
    Integer source := 7;
}
```

Listing 88: ./keyword_as_identifier2.test

```
#!/bin/bash

DESC="keyword as identifier"
OUT="syntax error"
ERR="Fatal error: exception Parsing.Parse_error"
COMPILE_ONLY=true
```

Listing 89: ./keyword_as_identifier.aml

```
#load<maze>

main():void
{
    Integer print := 7;
}
```

Listing 90: ./keyword_as_identifier.test

```
#!/bin/bash

DESC="keyword as identifier"
OUT="syntax error"
ERR="Fatal error: exception Parsing.Parse_error"
COMPILE_ONLY=true
```

Listing 91: ./list_literal.aml

```
#load<maze>

main():void
```

```
{
    print(<[1,2,3]>);
}
```

Listing 92: ./list_literal.test

```
#!/bin/bash

DESC="list literal"
OUT="[1, 2, 3]"
ERR=""
```

Listing 93: ./load_missing_maze.aml

```
#load<bogus>

main():void
{
    cell i := (CPos);
    print(get_Loc(i));
}
```

Listing 94: ./load_missing_maze.test

```
#!/bin/bash

DESC="attempts to load a missing maze"
OUT=""
ERR='Exception in thread "main" java.lang.NullPointerException
       at load_missing_maze.main(load_missing_maze.java:7)'
OUT="File Not Found"
```

Listing 95: ./main_with_args.aml

```
#load<maze>

main(Integer x):void
{
}
```

Listing 96: ./main_with_args.test

```
#!/bin/bash

DESC="main with args"
OUT="syntax error"
ERR="Fatal error: exception Parsing.Parse_error"
COMPILE_ONLY=true
```

Listing 97: ./maze.txt

```
6 3

0 0 1 0 1 1
1 1 2 1 0 0
1 1 3 0 0 1
```

Listing 98: ./mazevis.aml

```
#load<maze>

main():void{
        exit();
}
```

Listing 99: ./mazevis.test

```
#!/bin/bash

DESC="dfs␣algorithm"
OUT=""
ERR=""
GUITEST=true
```

Listing 100: ./missing_return_type.aml

```
#load<maze>

main()
{
}
```

Listing 101: ./missing_return_type.test

```
#!/bin/bash

DESC="missing␣return␣type"
OUT="syntax␣error"
ERR="Fatal␣error:␣exception␣Parsing.Parse_error"
COMPILE_ONLY=true
```

Listing 102: ./missing_semicolon.aml

```
#load<maze>

main():void
{
}
```

Listing 103: ./missing_semicolon.test

```
#!/bin/bash

DESC="missing␣semicolon"
OUT="syntax␣error"
ERR="Fatal␣error:␣exception␣Parsing.Parse_error"
COMPILE_ONLY=true
```

Listing 104: ./mulret.aml

```
#load-random

main():void{
        fn(1);
}

function fn(integer n):integer{
        return 1;
        n := 2;
        return 3;
}
```

Listing 105: ./mulret.test

```
#!/bin/bash

DESC="dead␣code;␣multiple␣returns"
OUT=""
ERR='Fatal error: exception Failure("Multiple␣return␣statements")'
COMPILE_ONLY=true
```

Listing 106: ./multi_line_comment.aml

```
#load<maze>

main():void
{
/* this is
 * a multiline
 * comment
 */
   print(true);
}
```

Listing 107: ./multi_line_comment.test

```
#!/bin/bash
```

```
2
3 DESC="multiline␣comment"
4 OUT="true"
5 ERR=""
```

Listing 108: ./nested_multi_line_comment.aml

```
1 #load<maze>
2
3 main():void
4 {
5 /* this is a
6  *
7 /* this is
8  * a multiline
9  * comment
10  */
11  * multiline
12  * comments do
13  * not nest
14  */
15 }
```

Listing 109: ./nested_multi_line_comment.test

```
1 #!/bin/bash
2
3 DESC="nested␣multiline␣comment"
4 OUT="syntax␣error"
5 ERR="Fatal␣error:␣exception␣Parsing.Parse_error"
6 COMPILE_ONLY=true
```

Listing 110: ./no_main.aml

```
1 #load<maze>
2
3 foo():void
4 {
5 }
```

Listing 111: ./no_main.test

```
1 #!/bin/bash
2
3 DESC="no␣main␣method"
4 OUT="syntax␣error"
5 ERR="Fatal␣error:␣exception␣Parsing.Parse_error"
6 COMPILE_ONLY=true
```

Listing 112: ./non_void_main.aml

```
#load<maze>

main():Boolean
{
}
```

Listing 113: ./non_void_main.test

```
#!/bin/bash

DESC="non-void main"
OUT="syntax error"
ERR="Fatal error: exception Parsing.Parse_error"
COMPILE_ONLY=true
```

Listing 114: ./no_preprocessor.aml

```
main():void
{

}
```

Listing 115: ./no_preprocessor.test

```
#!/bin/bash

DESC="missing preprocessor map load"
OUT="syntax error"
ERR="Fatal error: exception Parsing.Parse_error"
COMPILE_ONLY=true
```

Listing 116: ./order_of_operations.aml

```
#load<maze>

main():void
{
    print(7+9/3^0);
}
```

Listing 117: ./order_of_operations.test

```
#!/bin/bash

DESC="order of operations"
OUT="16.0"
ERR=""
```

Listing 118: ./rec.aml

```
#load-random

main():void{
	rec();
}

function rec():void{
	move_R();
	move_D();
	rec();
}
```

Listing 119: ./rec.test

```
#!/bin/bash

DESC="recursive␣call"
OUT=""
ERR=""
```

Listing 120: ./returns_wrong_type.aml

```
#load<maze>

main():void
{
    return 3;
}
```

Listing 121: ./returns_wrong_type.test

```
#!/bin/bash

DESC="returns␣wrong␣type"
OUT=""
ERR='Fatal error: exception Failure("Return␣statement␣is␣not␣permitted␣in␣main␣method")'
COMPILE_ONLY=true
```

Listing 122: ./single_line_comment.aml

```
#load<maze>

main():void
//this is a single line comment
{
    print(true);
}
```

Listing 123: ./single_line_comment.test

```
#!/bin/bash

DESC="single-line␣comment"
OUT="true"
ERR=""
COMPILE_ONLY=false
```

Listing 124: ./wrongtype.aml

```
#load-random

main():void
{
        bool x := fn();
        exit();
}

function fn():bool{
        return 1;
}
```

Listing 125: ./wrongtype.test

```
#!/bin/bash

DESC="wrong␣type"
OUT=""
ERR='Fatal error: exception Failure("return␣type␣mismatch")'
COMPILE_ONLY=true
```