# M.A.S.L.
# (Multi-Agent Simulation Language)

*COMSW4115 Programming Languages & Translators*

*Final Project Report*


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



1 Introduction ..... 4
1.1 Overview and Motivation ..... 4
1.2 Objective ..... 4
2 Language Tutorial ..... 4
2.1 Getting Started with the Compiler ..... 5
2.1.1 Environment Requirement ..... 5
2.1.2 Working with MASL Source Files and Compiler ..... 5
2.2 Writing a MASL Program ..... 5
2.2.1 "Hello, world!" ..... 5
2.2.2 Basic Data types and Variables ..... 5
2.2.3 Functions ..... 6
2.2.4 List ..... 6
2.2.5 Classes & Objects ..... 7
2.2.6 Program Structures and Simulation ..... 8
2.3 Putting Them All Together ..... 8
Computing Greatest Common Divisor ..... 8
2.3.1 Finding Even Numbers in a List ..... 9
2.3.2 Conway's Game of Life ..... 9
3 MASL Language Reference Manual ..... 11
3.1 Overview ..... 11
3.2 Conventions ..... 11

3.3 Lexical Conventions ..... 12
3.3.1 Tokens and Whitespaces ..... 12
3.3.2 Identifiers ..... 12
3.3.3 Keywords ..... 13
3.3.4 Comments ..... 13
3.4 Types and Values ..... 13
3.4.1 Data Types and Literals ..... 13
3.4.2 Variables ..... 15
3.4.3 Type System ..... 15
3.5 Expressions ..... 16
3.5.1 Primary Expressions ..... 16
3.5.2 Postfix Expressions ..... 16
3.5.3 Unary Operators ..... 16
3.5.4 Casts Expression ..... 16
3.5.5 Algorithmic Operators ..... 17
3.5.6 Relational Operators ..... 17
3.5.7 Equality Operators ..... 17
3.5.8 Logical Operators ..... 17
3.5.9 Assignment Expression ..... 18
3.5.10 Miscellaneous Operators ..... 18
3.5.11 Precedence and Associativity ..... 19
3.6 Functions ..... 19
3.6.1 Defining a Function ..... 19
3.6.2 Invoking a Function ..... 20
3.6.3 Functions as First Class Objects ..... 21
3.7 Classes ..... 21
3.7.1 Class Definition ..... 22
3.7.2 Member Variables ..... 22
3.7.3 Member Functions ..... 22
3.7.4 States ..... 23
3.7.5 Access Control ..... 24
3.8 Statements ..... 24

3.8.1 Types of Statements ........ 24
3.8.2 Structure of a MASL Source File ........ 26
3.8.3 Scope ........ 27
4 Project Plan ........ 27
4.1 Project process ........ 27
4.1.1 Planning and Specification ........ 27
4.1.2 Development ........ 27
4.1.3 Testing ........ 28
4.2 Team Responsibilities ........ 28
4.3 Project Timeline ........ 28
4.4 Software Development Environment ........ 29
4.5 Project Log ........ 29
5 Architecture Design ........ 30
5.1 Component of the MASL Compiler ........ 30
........ 30
5.2 Work of Each Member ........ 31
6 Test Plan ........ 31
7 Lessons Learned ........ 41
8 Appendix ........ 42
8.1 Scanner.mll ........ 42
8.2 parser.mly ........ 45
8.3 ast.ml ........ 49
8.4 semantic.ml ........ 52
8.5 translate.ml ........ 67
8.6 toplevel.ml ........ 72
8.7 astutils.ml ........ 75

# 1 Introduction

## 1.1 Overview and Motivation

The Agent-Based Model (ABM) describes a system where the interactions between autonomous agents (individuals) are simulated and the global patterns and effects of such interactions as a whole can be observed and assessed. ABMs have already been employed in various applications including analyzing traffic congestion, modeling social networks, predicting species populations and distributions, etc.

To facilitate building ABMs without having to start from scratch or engaging complex domain toolkits, Multi-Agent Simulation Language, or MASL, is proposed. MASL provides concise syntax and other convenient facilities for users to better focus on describing and solving the problems at hand.

More specifically, our language mainly focuses on the simulation of cellular automata, which is a specific category of ABM. The space of a cellular automaton is a discrete grid, where agents exist in individual cells of that grid and behave according to their observations of cells nearby.

## 1.2 Objective

Before discussing the objectives we expect to achieve with MASL, here is a summary of essential elements within a single agent in an ABM.

- Properties representing its state
- Actions that may change its properties
- Connections to some other agents in the environment, which may vary over time
- Heuristic rules that trigger certain actions based on the states of other agents connected to it

Given this, some most important objectives of designing MASL are listed below.

- Provide general programming constructs for specifying the states, actions and decision making process of agents, so that the ABMs implemented in MASL will not be limited to certain domains.
- Provide facilities (including succinct syntax and optimized underlying data storage) to define and access connections among agents efficiently.
- Let the users focus on agents and their connections, and handle the details of running simulations behind the scene.

# 2 Language Tutorial

# 2.1 Getting Started with the Compiler

### 2.1.1 Environment Requirement

A MASL source program will be finally compiled into a Java .class file. So a compiler of Java 1.6 or above is required to make the MASL compiler work.

### 2.1.2 Working with MASL Source Files and Compiler

A MASL source program is stored in a single text file with the suffix '.masl'. The toplevel of the MASL compiler, masl, can be used to compile a .masl file into a Java .class file, or simply translate it into a .java source file.

To compile a MASL source file into a Java class, use:

```
masl -c masl_prog.masl
```

The class generated will be named masl_prog, stored in the file masl_prog.class, and you can run the program with command "java masl_prog".

To translate a MASL source file into a Java source file, use:

```
masl -t masl_prog.masl
```

The generated Java source file will be named masl_prog.java.

# 2.2 Writing a MASL Program

### 2.2.1 "Hello, world!"

Writing a "hello world" program in MASL is quite easy:

```
printStr("Hello,world!");
```

This line can then be put into a .masl file and get compiled and run. Though extremely short, this program implies several important aspects of MASL, which will be elaborated in the following sections.

### 2.2.2 Basic Data types and Variables

MASL comes with four data basic types - integer, double, char and bool. To define variables of these types:

```
int year = 2008;
double pi = 3.14, earth2moon = 3.8e5;
```

```
char letter = 'a';
bool flag = true, on = false;
```

### 2.2.3 Functions

Functions in MASL is just like those in the C-family languages:

```
bool greaterThan3(int a) {
      return a > 3;
}
```

A function takes zero or more arguments and optionally returns a value, and together they characterize the interface of a function. What is different is that functions in MASL can be stored in variables and passed around, like:

```
fun ((int):bool) f = greaterThan;
```

somehow in the flavor of a functional language. Some natural usages of this feature will be discussed below.

### 2.2.4 List

Lists will come in handy if you want to store an array of elements of the same type in some logically related container. A list can contain any type of elements, including functions and lists themselves.

```
[int] fib = [int] {1, 1, 2, 3, 5, 8, 13, 21};
[[double]] matrix = {
      [double] { 1.0, 0.0, 0.0 },
      [double] { 0.0, 1.0, 0.0 }
      [double] { 0.0, 0.0, 1.0 }
};
```

Strings in MASL are essentially a list of characters. To define a string variable:

```
[char] greeting = [char] {'h', 'e', 'l', 'l', 'o'};
```

which is equivalent to the shorthand form:

```
[char] greeting = "hello";
```

Lists also provide some useful functions, like:

```
int s = fib:.size();
[int] above3 = fib:.filter(greaterThan);
[int] above5 = fib:.filter(fun (int n):bool { return n > 5 });
```

In the second, the function `greaterThan()` previously defined is passed into the list function filter. This will find all elements in fib that are greater than 3, and store them in the list above3.

Functions can not only be passed by their name - MASL also supports the concept of function literals, which can be passed around as well, as illustrated in the third case above.

### 2.2.5 Classes & Objects

Besides lists, classes fall in another category of compound types in MASL. A class can be defined like::

```
class Rectangle {
      double x;
      double y;
      double width;
      double  height;
}
```

And then we can create an instance of this class i.e. an object and access its member variables like:

```
class Rectangle r1 = class Rectangle();
r1.x = 0;
```

More sophisticated objects can have functions as member variables as well. However, the objects cannot access scope of the class where the other members are defined.

An object can have one or more states. A state consists of a name and some code that defines the behavior of that object under that state. For instance:

```
class Guard {

state Defend {
if(enemySighted()) this->Attack;
}

state Attack {
if(!enemyEliminated()) shoot();
else this->Defend;
}

bool enemySighted() { /*...*/ }
bool enemyEliminated() { /*...*/ }
}

class Guard g = class Guard();
g->Defend;
```

So when a guard (represented by Guard) is in defending state (state Defend), it will attack any enemies sighted. If it is in attacking state (state Attacking), it will keep shooting the enemy until it is eliminated, when it will go back to defending mode again.

As shown above, setting an object into a particular state can be done by using the -> operator. To check if an object is in a certain state, we can use:

```
g@Defend
```

which returns true if g is in state Defend, false otherwise.

The code in the current state of an object will be executed to update an object in each step of a simulation, which will be discussed below.

### 2.2.6 Program Structures and Simulation

A MASL program is simply a list of statement that will be executed in order. That is why the single line "hello world" program is able to work.

A MASL program, compiled from a single .masl source file, is essentially a MASL simulation. A simulation contains a set of agents represented by objects in MASL, and they will be able to communicate with each other and update themselves according to their states throughout the simulation.

Almost everything you write in a program is the preparation for actually running the simulation. To run a simulation, call the run() function from the top level of a program:

```
run(container);
```

where the argument container is a MASL list of class elements representing the agents of the simulation. By calling this function, the program will enter an infinite loop where the simulation described above will keep going.

## 2.3 Putting Them All Together

#### Computing Greatest Common Divisor

```
int gcd(int a, int b) {
if(a == b) return a;
else
      if(a > b) return gcd(a - b, b);
      else return gcd(a, b - a);
}
```

### 2.3.1 Finding Even Numbers in a List

```
bool isEvenNum(int num) { return (num%2 == 0); }
[int] list = [int]{1, 2, 3, 4, 5, 6};
[int] evenList = list:.filter(isEvenNum);
for(int i : evenList) printInt(i);
```

### 2.3.2 Conway's Game of Life

```
// Define the cell in the game as a class.
class Cell {


	// If the cell is alive, it will only live on with 2 or 3 live 
neighbors.
	state Live {
		r = 0.0;
		g = 0.0;
		b = 0.0;
		lastLive = live;
		live = true;
		int liveNeighborsNum = countLiveNeighbors(neighbors);
		if(liveNeighborsNum < 2 || liveNeighborsNum > 3) {
			r = 1.0;
			g = 1.0;
			b = 1.0;
			this->Dead;
		}
	}

	// If the cell is dead, only when exactly 3 live neighbors will render 
it alive.
	state Dead {
		r = 1.0;
		g = 1.0;
		b = 1.0;
		lastLive = live;
		live = false;
		if(countLiveNeighbors(neighbors) == 3) {
			r = 0.0;
			g = 0.0;
			b = 0.0;
			this->Live;
		}
	}

	// All neighbors of this cell
	[class Cell] neighbors;

	// State in last round
	bool lastLive = false;
	bool live = false;

	// A function to used to inspect whether a cell is alive or not
	bool isLive(class Cell c) {
```

```
        if (c.isUpdated) {
            return c.lastLive;
        } else {
            return c.live;
        }
    }

    // A routine counting the number of alive neighbors.
    int countLiveNeighbors([class Cell] n) {
        return n:.count(isLive);
    }

}

nx = 100;
ny = 100;
cellSize = 10;
interval = 100;

// Container of all cells
[class Cell] container;

// Matrix representation of all cells above
[[class Cell]] matrix;

// Initialize cells
for(int i = 0; i < nx; i = i + 1) {
    [class Cell] row;
    for (int j = 0; j < ny; j = j + 1) {
        class Cell c = class Cell();
        c.x = i;
        c.y = j;
        c->Dead;
        container:.append(c);
        row:.append(c);
    }
    matrix:.append(row);
}

// Initialize neighbors of each cell
for(class Cell c : container) {
    int x = c.x;
    int y = c.y;

    // Index of all neighbors
    [[int]] neighborIdx = [[int]]{
    [int]{x - 1, y - 1}, [int]{x - 1, y}, [int]{x - 1, y + 1},
    [int]{x, y - 1}, [int]{x, y + 1},
    [int]{x + 1, y - 1}, [int]{x + 1, y}, [int]{x + 1, y + 1}
    };

    // Handle index overflow and underflow, and
    // put the corresponding cell into current cell's neighbor list
    for([int] list : neighborIdx) {
        if(list:[0] < 0) {
            list:.set(0, list:[0] + nx);
        } else if (list:[0] > nx - 1) {
```

```
                        list:.set(0, nx - list:[0]);
                  }
                  if(list:[1] < 0) {
                        list:.set(1, list:[1] + ny);
                  } else if (list:[1] > ny - 1) {
                        list:.set(1, ny - list:[1]);
                  }
                  (c.neighbors):.append(matrix:[list:[0]]:[list:[1]]);
      }
}

// Create initial pattern
[[int]] liveIdx = [[int]]{[int]{1, 5},[int]{2, 5},[int]{1, 6},[int]{2,
6},[int]{11, 5},[int]{11, 6},[int]{11, 7},[int]{12, 4},[int]{13, 3},[int]{14,
3},[int]{12, 8},[int]{13, 9},[int]{14, 9},[int]{15, 6},[int]{17, 6},[int]{17,
5},[int]{17, 7},[int]{16, 4},[int]{16, 8},[int]{18, 6},[int]{21, 5},[int]{21,
4},[int]{21, 3},[int]{22, 3},[int]{22, 4},[int]{22, 5},[int]{23, 6},[int]{23,
2},[int]{25, 2},[int]{25, 1},[int]{25, 6},[int]{25, 7},[int]{35, 3},[int]{35,
4},[int]{36, 3},[int]{36, 4}};

for([int] l : liveIdx) {
      matrix:[l:[0]]:[l:[1]]->Live;
      matrix:[l:[0]]:[l:[1]].live = true;
      matrix:[l:[0]]:[l:[1]].lastLive = true;
}

run(container);
```

# 3 MASL Language Reference Manual

## 3.1 Overview

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

The chapters of this document come as follows.

Chapter 2 discusses the lexical conventions of MASL for identifiers, keywords and comments. Chapter 3 introduces the data types of MASL. Chapter 3.1 is about the expressions and operators. Chapter 4.1 and Chapter 5.1 focuses on functions and classes in MASL respectively, both of which in fact share a lot of features in syntax and behavior as basic data types. Chapter 7 discusses the control flow facilities of MASL, and classifies different types of statements and discusses what constitutes a MASL program. Chapter 8 provides a formal definition for the syntax of MASL using context-free grammar.

## 3.2 Conventions

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

```
int year = 2012;
```

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

control-flow-statement:
  **if (** expression **)** statement
  **if (** expression **)** statement **else** statement
  **for (** expressionopt **;** expressionopt **;** expressionopt **)** statement
  **for (** identifier **:** expression **)** statement
  **while (** expression **)** statement
  **do** statement **while (** expression **)**

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

## 3.3 Lexical Conventions

This chapter gives some basic knowledge of MASL lexicon. More lexical issues will be discussed in appropriate contexts later.
Currently a MASL source program is written with ASCII only, so the characters mentioned in the following text all refer to those in ASCII.

### 3.3.1 Tokens and Whitespaces

A token is a sequence of characters that specify an entity or mark a language construct in MASL. Tokens include identifiers, literals, keywords, operators and separators, each of which will be discussed later.
Whitespaces, including spaces, tabs and newlines, can be used to separate two adjacent tokens. Sometimes such separation is optional, but in other cases whitespaces are mandatory.

### 3.3.2 Identifiers

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

```
month  Year  Matrix3x3   _message
```

But the following ones are not:

```
someone@somewhere  9lives
```

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

```
masl MASL Masl
```

### 3.3.3 Keywords

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

All the keywords in MASL are listed below:

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

### 3.3.4 Comments

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

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

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

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

## 3.4 Types and Values

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

### 3.4.1 Data Types and Literals

#### 3.4.1.1 Basic Data Types

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

##### 3.4.1.1.1 Integers

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

```
142857
```

##### 3.4.1.1.2 Characters

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

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

‘\n’ New line character
‘\t’ Horizontal table character
‘\0’ Null character

##### 3.4.1.1.3 Floating Numbers

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

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

##### 3.4.1.1.4 Booleans

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

##### 3.4.1.1.5 Void

The data type void is used to represent “nothing”. Sometimes a MASL function does not return a value, and in this case, we say the return type of that function is void, equivalent to saying the function returns nothing.
There is only one literal for void: void itself.

##### 3.4.1.1.6 Lists

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

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

At runtime, we can read, write or remove any elements of a list, and also add new elements to a list at specified positions. These will be discussed in Section 4.11.

###### 3.4.1.1.7 Strings

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

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

#### 3.4.1.2 Functions and Classes

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

### 3.4.2 Variables

Generally, a variable is a named memory block containing a value of some data type. The syntax for defining one or more variables is as follows:

declaration:
type-specifier init-declaration-list
init-declaration-list:
init-declaration-list
init-declaration
init-declaration init-declaration-list
type identifier [= initial_value];

For instance, the following two statements define an integer variable and a double variable respectively:

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

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

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

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

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

### 3.4.3 Type System

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

# 3.5 Expressions

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

## 3.5.1 Primary Expressions

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

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

## 3.5.2 Postfix Expressions

Postfix expressions contain operators grouping from left to right.

```
postfix-expression:
    primary-expression
    postfix-expression : [ expression ]
    postfix-expression ( argument-expression-listopt )
    postfix-expression . identifier
    postfix-expression :. identifier
    postfix-expression @ identifier
    postfix-expression -> identifier
argument-expression-list:
    argument-expression
    argument-expression-list , argument-expression
```

## 3.5.3 Unary Operators

Expressions with unary operators group from right to left.

```
unary-expression:
    unary-operator cast-expression
unary-operator: one of
    + - !
```

## 3.5.4 Casts Expression

Cast expressions are used to convert data of one type to another.

```
cast-expression:
    unary-expression
    ( type-specifier ) cast-expression
```

### 3.5.5 Algorithmic Operators

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

```
multiplicative-expression:
    multiplicative-expression * cast-expression
    multiplicative-expression / cast-expression
    multiplicative-expression % cast-expression
additive-expression:
    multiplicative-expression
    additive-expression + multiplicative-expression
    additive-expression - multiplicative-expression
```

### 3.5.6 Relational Operators

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

### 3.5.7 Equality Operators

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

### 3.5.8 Logical Operators

```
logical-AND-expression:
    equality-expression
    logical-AND-expression && equality-expression
logical-OR-expression:
    equality-expression
    logical-OR-expression || equality-expression
```

### 3.5.9 Assignment Expression

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

#### 3.5.9.1 List Operations

##### 3.5.9.1.1 List References

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

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

A list reference subscript has the form

postfix-expression:
    postfix-expression [ expression ]

Some examples of getting list elements are given below:

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

### 3.5.10 Miscellaneous Operators

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

##### 3.5.10.1.1 Dot operator

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

expression:
    expression . identifier

For example:

someObject.someMember;

accesses the member someMember of object someObject.

##### 4.12.1.1 LDot operator

A LDot operator (:.) is used to access the embedded functions of a list.

expression:
    expression :. identifier

For example:

someList :. size();

### 3.5.11 Precedence and Associativity

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

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

## 3.6 Functions

### 3.6.1 Defining a Function

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

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

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

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

which specifies a list of parameters followed by the return type of the function separated by a colon, and then a compound statement which will be executed when the function is invoked.
To make programmers of C family languages more comfortable, we introduced a syntax sugar for function definitions as illustrated below:

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

which is equivalent to the previous style of function definitions.

In conclusion, function definitions have the form:

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

where:

type-name:
  …
  function-type
literal:
  …
  function-literal
parameter-list:
  parameter **,** parameter-list
  parameter
function-type:
  **(** parameter-type-list **) :** type-name
function-literal:
  **(** parameter-list **) :** type-name compound-statement
parameter:
  type-name **identifier**
parameter-type-list:
  type-name, parameter-type-list
  parameter, parameter-type-list
  type-name
  parameter

### 3.6.2 Invoking a Function

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

statement:
    expression **(** argument-list **)**

where:

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

An example of function invocation is shown below:

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

#### 3.6.2.1 By-value vs. By-Reference

When passing basic type arguments to a function, what the function access is just a copy of the arguments passed in. Thus, modification to the arguments inside the function does not affect the original data.
When passing an argument which is a list, a function or a class, however, what is actually passed is the copy of the reference to the original data. Thus the code inside a function is able to modify the value of such an argument, but it cannot modify the original reference itself, since what it accesses is merely a copy of that reference.

### 3.6.3 Functions as First Class Objects

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

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

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

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

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

## 3.7 Classes

In MASL, a class is an entity that encapsulates a set of attributes, behaviors and states, and relates them together. A class may have different behaviors under different states, and these behaviors may in turn access the attributes of that class or change its state. In essence, a class in MASL is a Definitive State Automaton (DFA). Moreover, a class must be defined in the global scope.

### 3.7.1 Class Definition

An class in MASL can be defined like this:

```
class className = {
…
}
```

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

### 3.7.2 Member Variables

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

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

To access the variable member in class A, we write:

```
A.number
```

Or

```
A.number = 1
```

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

### 3.7.3 Member Functions

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

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

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

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

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

### 3.7.4 States

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

```
class Warrior {

      state watch { if(enemyInSight()) -> attack; }
      state attack { fight(); if(!enemyInSight()) -> watch; }
      boolean enemyInSight() { … }
      void fight() { … }
}
```

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

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

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

```
class Warrior {

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

The class warrior will go into watch state after creation.

### 3.7.5 Access Control

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

#### 3.7.5.1 List Functions

A list has some members that provide useful information or operations on itself. Suppose list is a list whose elements are of type T. Then:
list.length is a member property that stores the number of elements in the list. Setting it will have no effect.
list:.filter(f) returns a sublist of list that only contains elements which meet a criterion defined using the parameter f. f is a function of type (T):boolean. The element being judged will be passed to it as an argument and it returns true when that element meets the criterion, otherwise returns false.

## 3.8 Statements

### 3.8.1 Types of Statements

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

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

#### 3.8.1.1 Declaration Statement

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

```
declaration-statement:
    basictype-declaration;
    function-declaration;
    class-declaration;
basictype-declaration:
    basictype-specifier init-declarator-list;
init-declarator-list:
    init-declarator;
```

```
        init-declarator, init-declarator-list
    init-declarator:
        declarator
        declarator = expression
    function-declaration:
        function identifier = type-specifier { statement };
    type-specifier:
        basic-type specifier
        function-type-specifier
        class
    class-declaration:
        class identifier = class-literal;
```

#### 3.8.1.2 Expression Statement

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

```
    expression-statement:
        expression;
        ;
```

#### 3.8.1.3 Compound Statement

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

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

#### 3.8.1.4 Control Flow Statements

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

```
    control-flow-statement:
        if ( expression ) statement
        if ( expression ) statement else statement
        for ( expressionopt; expressionopt; expressionopt ) statement
        for ( identifier : expression ) statement
        while ( expression ) statement
        do statement while ( expression )
```

Specifically, in the statement:

for ( expressionopt-init; expressionopt-condition; expressionopt-update ) statement
Any or all of the three expressions may be omitted. And if expressionopt-condition is omitted, the condition for iteration will be forever true, creating an infinite loop.
In the other form of for-loop:

for ( identifier : expression ) statement

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

#### 3.8.1.5 Jump Statement

There are 3 statements in MASL that can transfer the control flow instantly to another location.
To go on to the next iteration of the inner-most loop without executing the subsequent code of the loop body, use the continue statement, which may only appear within the body of a for or while loop.
To get out of the current loop immediately, use break. To return from anywhere in a function, use return, followed by an optional return value if the type of the function indicates one. These 3 kinds of statements all fall under the roof of jump statements.
jump-statement:

**continue**;
**break**;
**return** expressionopt;

A continue statement may appear only within a loop. It causes control to pass to the loop-continuation portion of the smallest enclosing loop.
A break statement may appear only in a loop, and it will terminate the execution of the smallest enclosing loop statement. Control will pass to the statement following the terminated loop statement.
A function returns to its caller by a return statement. When return is followed by an expression, the value is returned to the caller of the function. The expression is converted, as by assignment, to the type of the return value of the function it appears in.

### 3.8.2 Structure of a MASL Source File

In the top level of a MASL source file, where there are no embracing parentheses, the declaration statements, the expression statements, the control flow statements and compound statements may appear in any order, as long as the variables in a statement

are still in its scope (see Section 7.3). These statements will be executed in sequence at runtime.

### 3.8.3 Scope

#### 3.8.3.1 Lexical Scoping

MASL supports lexical scoping for variable bindings. Thus, the scope of a variable is effective from the end of its declaration statement till the end of the current block i.e. the component statement it is defined in.
If a variable is defined in the head of a block, such as a variable defined in the loop-continuation portion of a loop, or the parameter of a function, then the variable is accessible in the entire block.
Besides, code in a block is able to access the variables defined in an outer enclosing block. It is not true in reverse, however. This holds true for a state versus a class definition, a function and the top level of a MASL source file, etc.

#### 3.8.3.2 Class Member Accessibility

While conforming to lexical scoping rules, class member accessibility is also determined by the way it is created. In Section 6.5 we said that when a class is built based on another, all the members of the latter one will be copied into the former one. Thus, when trying to access the member of a class, MASL will first check all the members defined specifically in that class as well as the base class it is copied from. There is no inheritance in MASL, and one member name in a class must correspond to at most one member. If an attempt is made to write to a member with the same name as one from the base class, that member is simply overwritten instead of being hidden.

# 4 Project Plan

## 4.1 Project process

### 4.1.1 Planning and Specification

After finishing the Language Reference Manual, we had a team meeting on Saturday, 29 Sep. We made an elaborate plan for the whole development process. Every Saturday evening was chosen as the regular team meeting and we also finalized the architectural design and devised the project development rough timeline.

### 4.1.2 Development

With the project plan well-defined and specified, we initiated the project immediately. We established a repository on Github for version control and swift communication. The whole project was divided into several modules and every teammate had definite responsibilities. Thanks to parallel developing, the whole project progressed in a rapid pace. And due to the great assistance from Github, every member in the team was able to communicate and collaborate during the whole development process.

### 4.1.3 Testing

Several days before the presentation, all the modules are almost done and the synthesis of the whole compiler was started. Although every member in our team paid great attention to unit test during individual module developing, we ran into a lot of trouble when synthesizing the whole project. It was at that moment that we all realized that testing will never be overemphasized for a big group project development. We learned that if the interfaces between modules were defined better from the beginning and better unit tests were performed frequently and thoroughly, synthesis and final testing would be performed more smoothly. Fortunately, with automated testing, we tested our whole project thoroughly in the end.

## 4.2 Team Responsibilities

The following table indicates the responsibilities of each team member during the development process:

| Developer | Responsibilities |
|---|---|
| Dale Zhao | Overall framework, scanner, parser, AST Generation |
| Chong Zhang | Semantic |
| Wei Wang | Translate |
| Jiatian Li | Semantic |

## 4.3 Project Timeline

| Date | Milestone |
|---|---|
| 09-26-2012 | Project proposal completed |
| 10-31-2012 | LRM completed with language grammar specified |
| 11-20-2012 | Scanner and parser completed |
| 12-05-2012 | Semantic analysis completed |
| 12-10-2012 | Code generation completed |
| 12-13-2012 | Testing completed, Compiler fully completed |

| 12-18-2012 | Final report completed |
|---|---|

## 4.4 Software Development Environment

MSAL is written in Objective Caml (OCaml) and Java. The whole project is developed under Linux-based OS. Ocaml is used to develop the scanner, parser and translator. The source code will be translated into Java source code. The simulation engine and libraries are also written in Java.

To compile MSAL, we need the OCaml toolchain available at website:
http://caml.inria.fr/download and Java 1.6.
For version control, GitHub repository was used.

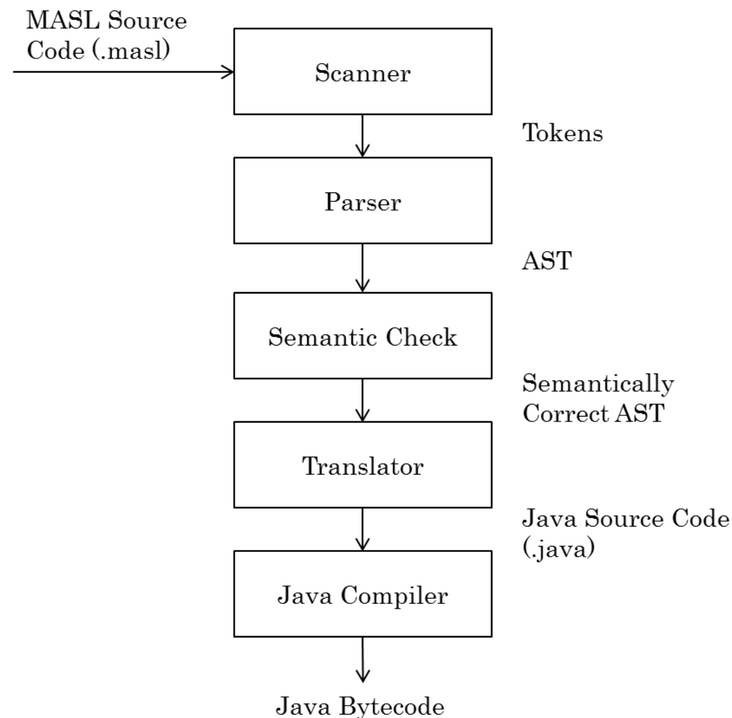
## 4.5 Project Log

The following table lists actual dates of significant project milestones.

| Date | Milestone/Feature |
|---|---|
| 09-20-2012 | Project initiated |
| 09-28-2012 | Language proposal completed |
| 10-25-2012 | Language Reference Manual draft completed |
| 10-28-2012 | Language convention and grammar finalized |
| 10-29-2012 | Language Reference Manual completed |
| 10-29-2012 | Architecture designed and teammate responsibilities specified |
| 12-03-2012 | Interfaces between modules defined |
| 12-10-2012 | Scanner and parser completed |
| 12-18-2012 | Semantic analysis completed |
| 12-18-2012 | Code generation completed |
| 12-18-2012 | Testing completed, Compiler fully completed |
| 12-19-2012 | Final report completed |

# 5 Architecture Design

## 5.1 Component of the MASL Compiler




The compiler has 5 blocks. They are scanner, parser, abstract syntax tree, semantic checker and translator.

**Scanner**
The scanner transforms the MASL source code into tokens. At this stage, comments are omitted. The scanner can also catch illegal characters. This part is written in scanner.mll file.

**Parser**
The parser checks syntax correctness using tokens generated by scanner. In addition, with these tokens the parser generates an abstract syntax tree. This part is written in parser.mly file.

**AST**

AST (Abstract Syntax Tree. It is a kind of hierarchy structure whose nodes are related to productions. It is the interface between the scanner/parser and semantic checker.

**Semantic Checker**
With the AST, the semantic checker checks the semantic correctness of the program. It recursively walks over the AST and checks each node. Here are some examples of what semantic check does. It checks whether a function returns a correct type, whether the variables are defined before used, whether the both sides of a binary operator is legal, and so on.

**Translator**
After passing the semantic check, it is guaranteed that the program is semantically correct. The translator walks over the AST again and translates the each node into into corresponding Java source code. After all nodes are visited, the translator translate the MASL program into a Java source.

## 5.2 Work of Each Member

The scanner, parser and AST parts are written by Dale Zhao. Chong Zhang and Jiatian Li work on the semantic check part. Wei Wang works on the translator.

# 6 Test Plan

1. Representative source language programs:

(1) Greatest Common Divisor

a. MASL Source Code:

```
int gcd(int a, int b) {
while (a != b) {
  if (a > b) {
            a = a - b;
      } else {
            b = b - a;
      }
}
      return a;
}
```

```
printInt(gcd(2,14));
printChar('\n');
printInt(gcd(3,15));
printChar('\n');
printInt(gcd(99,121));
printChar('\n');
```

b. Java Source Code after Translation:

```
public class gcd1 extends MaslSimulation {
      MaslFunction<Integer> gcd=new MaslFunction<Integer>() {
            @Override
            public Integer invoke(Object... args) {
                  int a = (Integer) args[0];
                  int b = (Integer) args[1];
                  {
                        while(a!=b) {
                              {
                                    if(a>b) {
                                          {
                                                a=a-b;
                                          }
                                    } else {
                                          {
                                                b=b-a;
                                          }
                                    }
                              }
                        }
                        return a;
                  }
            }
      }
      ;
      public void init() {
            printInt.invoke(gcd.invoke(2,14));
            printChar.invoke('\n');
            printInt.invoke(gcd.invoke(3,15));
            printChar.invoke('\n');
            printInt.invoke(gcd.invoke(99,121));
            printChar.invoke('\n');
      }
      public static void main(String[] args) {
            gcd1 sim = new gcd1();
            sim.init();}
}
```

(2) Game of Life - Glider Gun

a. MASL Source Code:

```
// Define the cell in the game as a class.
class Cell {


      // If the cell is alive, it will only live on with 2 or 3 live
neighbors.
      state Live {
            r = 0.0;
            g = 0.0;
            b = 0.0;
            lastLive = live;
            live = true;
            int liveNeighborsNum = countLiveNeighbors(neighbors);
            if(liveNeighborsNum < 2 || liveNeighborsNum > 3) {
                  r = 1.0;
                  g = 1.0;
                  b = 1.0;
                  this->Dead;
            }
      }

      // If the cell is dead, only when exactly 3 live neighbors will render
it alive.
      state Dead {
            r = 1.0;
            g = 1.0;
            b = 1.0;
            lastLive = live;
            live = false;
            if(countLiveNeighbors(neighbors) == 3) {
                  r = 0.0;
                  g = 0.0;
                  b = 0.0;
                  this->Live;
            }
      }

      // All neighbors of this cell
      [class Cell] neighbors;

      // State in last round
      bool lastLive = false;
      bool live = false;

      // A function to used to inspect whether a cell is alive or not
      bool isLive(class Cell c) {
            if (c.isUpdated) {
                  return c.lastLive;
            } else {
                  return c.live;
            }
      }
```

```
	// A routine counting the number of alive neighbors.
	int countLiveNeighbors([class Cell] n) {
		return n:.count(isLive);
	}

}

nx = 100;
ny = 100;
cellSize = 10;
interval = 100;

// Container of all cells
[class Cell] container;

// Matrix representation of all cells above
[[class Cell]] matrix;

// Initialize cells
for(int i = 0; i < nx; i = i + 1) {
	[class Cell] row;
	for (int j = 0; j < ny; j = j + 1) {
		class Cell c = class Cell();
		c.x = i;
		c.y = j;
		c->Dead;
		container:.append(c);
		row:.append(c);
	}
	matrix:.append(row);
}

// Initialize neighbors of each cell
for(class Cell c : container) {
	int x = c.x;
	int y = c.y;

	// Index of all neighbors
	[[int]] neighborIdx = [[int]]{
	[int]{x - 1, y - 1}, [int]{x - 1, y}, [int]{x - 1, y + 1},
	[int]{x, y - 1}, [int]{x, y + 1},
	[int]{x + 1, y - 1}, [int]{x + 1, y}, [int]{x + 1, y + 1}
	};

	// Handle index overflow and underflow, and
	// put the corresponding cell into current cell's neighbor list
	for([int] list : neighborIdx) {
		if(list:[0] < 0) {
			list:.set(0, list:[0] + nx);
		} else if (list:[0] > nx - 1) {
			list:.set(0, nx - list:[0]);
		}
		if(list:[1] < 0) {
			list:.set(1, list:[1] + ny);
```

```
            } else if (list:[1] > ny - 1) {
                  list:.set(1, ny - list:[1]);
            }
            (c.neighbors):.append(matrix:[list:[0]]:[list:[1]]);
      }
}

// Create initial pattern
[[int]] liveIdx = [[int]]{[int]{1, 5},[int]{2, 5},[int]{1, 6},[int]{2,
6},[int]{11, 5},[int]{11, 6},[int]{11, 7},[int]{12, 4},[int]{13, 3},[int]{14,
3},[int]{12, 8},[int]{13, 9},[int]{14, 9},[int]{15, 6},[int]{17, 6},[int]{17,
5},[int]{17, 7},[int]{16, 4},[int]{16, 8},[int]{18, 6},[int]{21, 5},[int]{21,
4},[int]{21, 3},[int]{22, 3},[int]{22, 4},[int]{22, 5},[int]{23, 6},[int]{23,
2},[int]{25, 2},[int]{25, 1},[int]{25, 6},[int]{25, 7},[int]{35, 3},[int]{35,
4},[int]{36, 3},[int]{36, 4}};

for([int] l : liveIdx) {
      matrix:[l:[0]]:[l:[1]]->Live;
      matrix:[l:[0]]:[l:[1]].live = true;
      matrix:[l:[0]]:[l:[1]].lastLive = true;
}

run(container);
```

b. Java Code after Translation:

```
public class gliderGun extends MaslSimulation {
      public class Cell extends MaslClass {
            public void __update() {
                  if("Live".equals(__curState)) {
                        Live();
                  } else if("Dead".equals(__curState)) {
                        Dead();
                  } else if(true){}
                  isUpdated = true;
            }
            private void Live() {
                  {
                        r=0.;
                        g=0.;
                        b=0.;
                        lastLive=live;
                        live=true;
                        int
liveNeighborsNum=countLiveNeighbors.invoke(neighbors);
                        if(liveNeighborsNum<2||liveNeighborsNum>3) {
                              {
                                    r=1.;
                                    g=1.;
                                    b=1.;
```

```
                        this.__curState = "Dead";
                    }
                } else {
                }
            }
        }
        private void Dead() {
            {
                r=1.;
                g=1.;
                b=1.;
                lastLive=live;
                live=false;
                if(countLiveNeighbors.invoke(neighbors)==3) {
                    {
                        r=0.;
                        g=0.;
                        b=0.;
                        this.__curState = "Live";
                    }
                } else {
                }
            }
        }

        public   MaslList<Cell> neighbors=new MaslList<Cell>();;

        public   boolean lastLive=false;

        public   boolean live=false;

        public MaslFunction<Boolean> isLive=new MaslFunction<Boolean>() {
            @Override
            public Boolean invoke(Object... args) {
                Cell c = (Cell) args[0];
                {
                    if(c.isUpdated) {
                        {
                            return c.lastLive;
                        }
                    } else {
                        {
                            return c.live;
                        }
                    }
                }
            }
        }
        ;
```

```
            public MaslFunction<Integer> countLiveNeighbors=new
MaslFunction<Integer>() {
                  @Override
                  public Integer invoke(Object... args) {
                        MaslList<Cell> n = (MaslList<Cell>) args[0];
                        {
                              return n.count(isLive);
                        }
                  }
            }
            ;

            public String __curState = null;
            public String toString() {
                  return "Cell{x:" + x + ",y:" + y + ",r:" + r + ",g:" + g +
",b:" + b + "}@" + __curState;
            }
      }
      public void init() {
            nx=100;
            ny=100;
            cellSize=10;
            interval=100;
            MaslList<Cell> container=new MaslList<Cell>();;
            MaslList<MaslList<Cell>> matrix=new MaslList<MaslList<Cell>>();;
            for(int i=0;
                        i<nx;i=i+1) {
                  {
                        MaslList<Cell> row=new MaslList<Cell>();;
                        for(int j=0;
                                    j<ny;j=j+1) {
                              {
                                    Cell c=new Cell();
                                    c.x=i;
                                    c.y=j;
                                    c.__curState = "Dead";
                                    container.append(c);
                                    row.append(c);
                              }
                                    }
                        matrix.append(row);
                  }
                        }
            for(Cell c:container) {
                  {
                        int x=c.x;
                        int y=c.y;
                        MaslList<MaslList<Integer>> neighborIdx=new
```

```
MaslList<MaslList<Integer>>(new MaslList<Integer>(x-1,y-1),new
MaslList<Integer>(x-1,y),new MaslList<Integer>(x-1,y+1),new
MaslList<Integer>(x,y-1),new MaslList<Integer>(x,y+1),new
MaslList<Integer>(x+1,y-1),new MaslList<Integer>(x+1,y),new
MaslList<Integer>(x+1,y+1));
                        for(MaslList<Integer> list:neighborIdx) {
                              {
                                    if(list.get(0)<0) {
                                          {
                                                list.set(0,list.get(0)+nx);
                                          }
                                    } else {
                                          if(list.get(0)>nx-1) {
                                                {
                                                      list.set(0,nx-
list.get(0));
                                                }
                                          } else {
                                          }
                                    }
                                    if(list.get(1)<0) {
                                          {
                                                list.set(1,list.get(1)+ny);
                                          }
                                    } else {
                                          if(list.get(1)>ny-1) {
                                                {
                                                      list.set(1,ny-
list.get(1));
                                                }
                                          } else {
                                          }
                                    }

      c.neighbors.append(matrix.get(list.get(0)).get(list.get(1)));
                              }
                        }
                  }
            }
            MaslList<MaslList<Integer>> liveIdx=new
MaslList<MaslList<Integer>>(new MaslList<Integer>(1,5),new
MaslList<Integer>(2,5),new MaslList<Integer>(1,6),new
MaslList<Integer>(2,6),new MaslList<Integer>(11,5),new
MaslList<Integer>(11,6),new MaslList<Integer>(11,7),new
MaslList<Integer>(12,4),new MaslList<Integer>(13,3),new
MaslList<Integer>(14,3),new MaslList<Integer>(12,8),new
MaslList<Integer>(13,9),new MaslList<Integer>(14,9),new
MaslList<Integer>(15,6),new MaslList<Integer>(17,6),new
MaslList<Integer>(17,5),new MaslList<Integer>(17,7),new
```

```
MaslList<Integer>(16,4),new MaslList<Integer>(16,8),new
MaslList<Integer>(18,6),new MaslList<Integer>(21,5),new
MaslList<Integer>(21,4),new MaslList<Integer>(21,3),new
MaslList<Integer>(22,3),new MaslList<Integer>(22,4),new
MaslList<Integer>(22,5),new MaslList<Integer>(23,6),new
MaslList<Integer>(23,2),new MaslList<Integer>(25,2),new
MaslList<Integer>(25,1),new MaslList<Integer>(25,6),new
MaslList<Integer>(25,7),new MaslList<Integer>(35,3),new
MaslList<Integer>(35,4),new MaslList<Integer>(36,3),new
MaslList<Integer>(36,4));
            for(MaslList<Integer> l:liveIdx) {
                  {
                        matrix.get(l.get(0)).get(l.get(1)).__curState =
"Live";
                        matrix.get(l.get(0)).get(l.get(1)).live=true;
                        matrix.get(l.get(0)).get(l.get(1)).lastLive=true;
                  }
            }
            run.invoke(container);
      }
      public static void main(String[] args) {
            gliderGun sim = new gliderGun();
            sim.init();}
}
```

2. Test Suites

(1) Unit Testing

To limit the scope of a bug and make it easy to debug, we carried out intensive unit testing. We have following test suites and they cover every basic element of our language:

(All unit test suites can be found in tests directory)

```
block.masl
class1.masl
class2.masl
dowhile.masl
expr.masl
for1.masl
for2.masl
for3.masl
foreach1.masl
```

```
foreach2.masl
fun.masl
gcd1.masl
if1.masl
list1.masl
while1.masl
while2.masl
```

(2) Integration Testing

We have several large demos written in our language to perform a comprehensive testing. The two integration test suites are gamOfLife.masl and gliderGun.masl.

3. Test Suites Justification

We iterated all nodes in our AST, identified basic elements of our language and tried to create test cases that cover all basic structures of our language. With the set of test suites, we can make sure that every basic element in our language is tested. With an automation testing script, we can easily perform regression testing after we add or modify some features in our project. With regression testing based on a set of unit test suites, we can easily locate a bug if we get one, because the scope of a bug is limited in a very small module.

4. Automation Testing

A simple shell script test.sh (in root directory of submission) is written to automate the test suites above. It iterates each test case, checks translation and compilation of the source code, runs the program and compares the program output to expected output. If some steps of some test cases result in errors, the automation script will print the errors to console.

5. Who did What

Wei Wang (ww2315): wrote gameOfLife.masl, gliderGun.masl, test.sh and part of unit test suites.

Jiatian Li (jl3930): wrote part of unit test suites.

# 7 Lessons Learned

**Chong Zhang**

I have learned a lot from the project and get a better understanding of the knowledge I learned in the class. In addition, I get more familiar with OCaml.

However, it is miserable during the implementation. We could have started early. Actually, we just put everything to the end of the term. It is really a nightmare. To make things even worse, the features of MASL are too complicated. My suggestion to the future students is that design an easier and more specific language.

**Dale Zhao**

Developing a working compiler of our own devise is a great way to relate theories taught in class to coding practice. By doing this final project, I have gained a more comprehensive insight on various data structures and algorithms used in the compiling process, as well as how different components of a compiler support each other and work as a whole.

As the group leader, I also learned how to better dispatch the work for different parts of the project to individual teammates. A set of well designed interfaces and a solid code framework in the early stage can greatly expedite this process. We also decided to use GitHub to establish our shared code repository, which greatly facilitate team collaboration.

As a word for future students doing this project, starting early and simple is always the best strategy to make room for unexpected difficulties during the project cycle. Also, sufficient unit tests can save you a lot of time from debugging in times of integration.

**Wei Wang**

The most important lesson I learned is that designing a language with beautiful and unambiguous grammar is not easy. We came up with a design of grammar that we thought was beautiful. However, when implementing the abstract syntax tree based on the syntax, we got numerous shift/reduce and reduce/reduce errors. We tried a lot of alternatives and most of them failed. At last, we gave up some features to make the grammar unambiguous. It made the language not as beautiful as we expected. Designing beautiful and unambiguous language is like making a blade which is sharp and safe to use. We should carefully keep balance between the two factors.

Advice for Future Teams:

1) Using version control system is really beneficial when you are working in a team.
2) Start early and enjoy the process. Leaving all the work in final week leads to a disaster.
3) Make sure all members in your team have consistent development environment. Devise a programming style guide before you start.

**Jiatian Li**

Language implementation is a huge challenge. Having lived through it, I learned a lot. First, start early and better planning is very important. Language development is not a simple process. One should never expect to finish this job over night. Second, the feedback from the TA is extremely valuable. After evaluating our proposal and LRM, the TA warned us several time that there are only several weeks to implement the whole project and it is almost impossible to finish all the ideas in such a short time. In retrospect, if we followed the TA's suggestion and started simple, we would not have to suffer the nightmare in the last few days. What's more, we should well define the interfaces between modules from the beginning and perform unit tests frequently and thoroughly. If we do so, we would not run into trouble when we synthesize every module in the end. Finally, the Microc should be deemed as a ticket to Noah's Ark throughout the whole project. Because we didn't convert the raw AST to SAST, we ran into lots of trouble and inconvenience when implementing the semantic check and code generation part.

# 8 Appendix

## 8.1 Scanner.mll

```
(* Primary Author: Dale zhao (dz2242) *)
(* Scanner for MASL. *)

(* Header section. *)
{
      open Parser
      (* TODO Keep track of character position. *)
      let inc_lnum lexbuf =
            let pos = lexbuf.Lexing.lex_curr_p in
            lexbuf.Lexing.lex_curr_p <- {pos with
                  Lexing.pos_lnum = pos.Lexing.pos_lnum + 1;
                  Lexing.pos_bol = pos.Lexing.pos_cnum;
            }
      ;;

      let explode s =
            let rec exp i l =
                  if i < 0 then l else exp (i - 1) (s.[i]::l) in
            exp (String.length s - 1) [];;
}

(* Definition section *)

let whitespace = [' ' '\t' '\r' '\n']
let nonwhitespace = _#whitespace
let digit = ['0'-'9']
let letter = ['a'-'z' 'A'-'Z']
let input_char = _#['\r' '\n']
let common_escape_sequence = "\\t" | "\\r" | "\\n" | "\\0"
let single_char = input_char#['\'' '\\']
let escape_sequence_char = common_escape_sequence | "\\'"
let single_char_string = input_char#['"' '\\']
let escape_sequence_string = common_escape_sequence | "\\\""

(* Rule section. *)

rule single_comment_parser = parse
      '\n'  { inc_lnum lexbuf; token_parser lexbuf }
      | _#['\n']  { single_comment_parser lexbuf }

and multiline_comment_parser = parse
      "*/"  { token_parser lexbuf }
      | _   as char { if char == '\n' then inc_lnum lexbuf;
multiline_comment_parser lexbuf }

and token_parser = parse
      (* Whitespaces. *)
      whitespace as char      { if char == '\n' then inc_lnum lexbuf;
token_parser lexbuf }
      (* Comments. *)
      | "//"      { single_comment_parser lexbuf }
      | "/*"      { multiline_comment_parser lexbuf }
      (* Separators. *)
      | ':' { COLON }
      | ';' { SEMICOLON }
      | ',' { COMMA }
      | '(' { LPAREN }
```

```
      | ')' { RPAREN }
      | '{' { LBRACE }
      | '}' { RBRACE }
      | '[' { LSQBRA }
      | ']' { RSQBRA }
      (* Keywords. *)
      | "int"     { INT }
      | "double"  {  DOUBLE }
      | "char"    { CHAR }
      | "bool"    { BOOL }
      | "class"   { CLASS }
      | "object"  { OBJECT }
      | "fun" { FUN }
      | "void"    { VOID }
      | "if"      { IF }
      | "else"    { ELSE }
      | "for"     { FOR }
      | "while"   { WHILE }
      | "do"      { DO }
      | "continue"      { CONTINUE }
      | "break"   { BREAK }
      | "return"  { RETURN }
      | "this"    { THIS }
      | "state"   { STATE }
      (* Operators. *)
      | '+' { PLUS }
      | '-' { MINUS }
      | '*' { MULT }
      | '/' { DIV }
      | '%' { MOD }
      | "&&"      { AND }
      | "||"      { OR }
      | '!' { NOT }
      | '>' { GT }
      | ">="      { GE }
      | "=="      { EQ }
      | "!="      { NEQ }
      | "<="      { LE }
      | '<' { LT }
      | '=' { ASSIGN }
      | '.' { DOT }
      | '@' { AT }
      | "->"      { TRANS }
      (* Literals (for basic data types). *)
      | digit+ as lxm   { INT_LITERAL(int_of_string lxm) }
      | digit+ ('.' digit*)? (['e' 'E'] ['+' '-']? digit+)
      | digit+ ('.' digit*) (['e' 'E'] ['+' '-']? digit+)?
      | digit* '.' digit+ (['e' 'E'] ['+' '-']? digit+)?    as lxm
{ DOUBLE_LITERAL(float_of_string lxm) }
      | '\'' single_char '\'' as lxm      { CHAR_LITERAL(lxm.[1]) }
      | '\''  escape_sequence_char '\'' as lxm { CHAR_LITERAL(Scanf.sscanf
("\"" ^ lxm ^ "\"") "%S%!" (fun u -> u.[1])) }
      | "true" | "false" as lxm     { BOOL_LITERAL(bool_of_string lxm) }
      | '"' (single_char_string | escape_sequence_string)* '"'    as lxm
{ STRING_LITERAL(explode (Scanf.sscanf ("\"" ^ (String.sub lxm 1
(String.length lxm - 2)) ^ "\"") "%S%!" (fun u -> u))) }
            (* Identifiers. *)
```

```
      | (letter | '_') (letter | '_' | digit)* as lxm { ID(lxm) }
      | _ as lxm { raise (Failure("illegal token" ^ (Char.escaped lxm))); }
      | eof { EOF }
      (* TODO Handle EOF and invalid input. *)

(* Trailer section. *)
{
}
```

## 8.2 parser.mly

```
/* Primary Author: Dale zhao (dz2242) */
/* Parser for MASL. */

/* Header section. */
%{
      open Ast;;
%}

/* Declaration section. */

/* Declaring tokens. */

/* Separators. */
%token COLON SEMICOLON COMMA LPAREN RPAREN LBRACE RBRACE LSQBRA RSQBRA EOF

/* Keywords. */
/* Type specifiers. */
%token INT DOUBLE CHAR BOOL CLASS OBJECT FUN VOID
/* Control flow. */
%token IF ELSE FOR WHILE DO CONTINUE BREAK RETURN
/* Object definitions. */
%token STATE
%token THIS

/* Operators. */
/* Arithmetic operators. */
%token PLUS MINUS MULT DIV MOD
/* Logic operators. */
%token AND OR NOT
/* Relational operators. */
%token GT GE EQ NEQ LE LT
/* Assignment operators. */
%token ASSIGN
/* Object manipulations. */
%token DOT AT TRANS

/* Identifiers. */
%token <string> ID

/* Literals (for basic data types). */
%token <int> INT_LITERAL
%token <float> DOUBLE_LITERAL
%token <char> CHAR_LITERAL
```

```
%token <bool> BOOL_LITERAL
%token <char list> STRING_LITERAL

/* Associativity and precedence of operators. */
%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left GT GE LE LT
%left PLUS MINUS
%left MULT DIV MOD
%left UPLUS UMINUS NOT
%left DOT
%left AT TRANS
%left LPAREN RPAREN
%left COLON
%left LSQBRA RSQBRA

%start program
%type <Ast.program> program

%%

/* Rule section. */

program:
        stmt_list { Program($1) }

stmt_list:
        /* Empty. */ { [] }
      | stmt_nonempty_list { List.rev $1 }

stmt_nonempty_list:
        stmt { [$1] }
      | stmt_nonempty_list stmt { $2 :: $1 }

stmt:
        SEMICOLON { NoStmt }
            /* Declaration statement. */
      | basic_type_decl SEMICOLON { $1 }
      | func_decl { $1 }
      | class_decl { $1 }
        /* Expression statement. */
      | expr SEMICOLON { Expr($1) }
        /* Compound statement. */
      | comp_stmt { $1 }
        /* Control flow statement. */
      | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, NoStmt) }
      | IF LPAREN expr RPAREN stmt ELSE stmt { If($3, $5, $7) }
      | FOR LPAREN stmt expr_opt SEMICOLON expr_opt RPAREN stmt { For($3, $4,
$6, $8) }
      | FOR LPAREN type_specifier ID COLON expr RPAREN stmt { ForEach($3, $4,
$6, $8) }
      | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
      | DO comp_stmt WHILE LPAREN expr RPAREN { DoWhile($2, $5) }
```

```
        /* Jump statement. */
      | CONTINUE SEMICOLON { Continue }
      | BREAK SEMICOLON { Break }
      | RETURN expr_opt SEMICOLON { Return($2) }

basic_type_decl:
        type_specifier basic_init_decl_list { BasicDecl($1, List.rev $2) }

type_specifier:
        INT { Int }
      | DOUBLE { Double }
      | CHAR { Char }
      | BOOL { Bool }
      | VOID { Void }
      | LPAREN param_type_list RPAREN COLON type_specifier { FuncType($5,
$2) }
      | CLASS ID { Class($2) }
      | LSQBRA type_specifier RSQBRA { ListType($2) }

basic_init_decl_list:
        basic_init_decl { [$1] }
      | basic_init_decl_list COMMA basic_init_decl { $3 :: $1 }

basic_init_decl:
        ID { BasicInitDefault($1) }
      | ID ASSIGN expr { BasicInitAssign($1, $3) }

func_decl:
        FUN LPAREN type_specifier RPAREN ID ASSIGN expr { FuncDecl($3, $5,
$7) }
      | type_specifier ID LPAREN param_list RPAREN comp_stmt
{ FuncDecl(FuncType($1, List.map fst $4), $2, FuncLit($1, $4, $6)) }

param_list:
        /* Empty. */ { [] }
      | nonempty_param_list { List.rev $1 }

nonempty_param_list:
        param { [$1] }
      | nonempty_param_list COMMA param { $3 :: $1 }

param:
        type_specifier ID { ($1, $2) }

param_type_list:
        /* Empty. */ { [] }
      | nonempty_param_list { List.rev (List.map fst $1) }
      | nonempty_unnamed_param_type_list { List.rev $1 }

nonempty_unnamed_param_type_list:
        type_specifier { [$1] }
      | nonempty_unnamed_param_type_list COMMA type_specifier { $3 :: $1 }

func_literal:
       FUN LPAREN param_list RPAREN COLON type_specifier comp_stmt { ($6, $3,
$7) }
```

```
class_decl:
        CLASS ID LBRACE state_decl_list stmt_list RBRACE { ClassDecl($2,
List.rev $4, $5) }

object_literal:
        type_specifier LPAREN RPAREN { ObjectLit($1) }

list_literal:
        LSQBRA type_specifier RSQBRA LBRACE list_elems RBRACE { ListLit($2,
$5) }
      | STRING_LITERAL { ListLit(Char, List.map (fun c ->
BasicLit(CharLit(c))) $1) }

list_elems:
        /* Empty */ { [] }
      | nonempty_list_elems { List.rev $1 }

nonempty_list_elems:
        expr { [$1] }
      | nonempty_list_elems COMMA expr { $3 :: $1 }

state_decl_list:
        /* Empty. */ { [] }
      | state_decl_list state_decl { $2:: $1 }

state_decl:
        STATE ID comp_stmt { ($2, $3) }

basic_literal:
        INT_LITERAL { IntLit($1) }
      | DOUBLE_LITERAL { DoubleLit($1) }
      | CHAR_LITERAL { CharLit($1) }
      | BOOL_LITERAL { BoolLit($1) }
      | object_literal { $1 }
      | list_literal { $1 }

expr:
      | ID { Id($1) }
      | basic_literal { BasicLit($1) }
      | func_literal { FuncLit($1) }
      | THIS { This }
      | LPAREN expr RPAREN { $2 }
      | expr LPAREN arg_list RPAREN { FuncCall($1, $3) }
      | PLUS expr %prec UPLUS { UnaryOp(Plus, $2) }
      | MINUS expr %prec UMINUS { UnaryOp(Minus, $2) }
      | expr MULT expr { BinaryOp($1, Mult, $3) }
      | expr DIV expr { BinaryOp($1, Div, $3) }
      | expr MOD expr { BinaryOp($1, Mod, $3) }
      | expr PLUS expr { BinaryOp($1, Plus, $3) }
      | expr MINUS expr { BinaryOp($1, Minus, $3) }
      | expr GT expr { BinaryOp($1, Gt, $3) }
      | expr GE expr { BinaryOp($1, Ge, $3) }
      | expr EQ expr { BinaryOp($1, Eq, $3) }
      | expr NEQ expr { BinaryOp($1, Neq, $3) }
      | expr LE expr { BinaryOp($1, Le, $3) }
      | expr LT expr { BinaryOp($1, Lt, $3) }
      | NOT expr { UnaryOp(Not, $2) }
```

```
      | expr AND expr { BinaryOp($1, And, $3) }
      | expr OR expr { BinaryOp($1, Or, $3) }
      | expr DOT expr { BinaryOp($1, Dot, $3) }
      | expr AT expr { BinaryOp($1, At, $3) }
      | expr TRANS expr { BinaryOp($1, Trans, $3) }
      | expr ASSIGN expr { BinaryOp($1, Assign, $3) }
      | expr COLON LSQBRA expr RSQBRA { BinaryOp($1, Index, $4) }
      | expr COLON DOT expr { BinaryOp($1, LDot, $4) }

expr_opt:
        /* Empty. */ { NoExpr }
      | expr { $1 }

arg_list:
        /* Empty */ { [] }
      | nonempty_arg_list { List.rev $1 }

nonempty_arg_list:
        expr { [$1] }
      | nonempty_arg_list COMMA expr { $3 :: $1 }

comp_stmt:
        LBRACE stmt_list RBRACE { CompStmt($2) }

%%

(* Trailer section. *)
```

# 8.3 ast.ml

```
(* Primary Author: Dale zhao (dz2242) *)

(*** Tokens. ***)




(* Operators. *)

type op =

        Plus | Minus | Mult | Div | Mod

      | And | Or | Not

      | Gt | Ge | Eq | Neq | Le | Lt

      | Assign

      | Dot | At | Trans | Index | LDot
```

```
(* Basic type literals. *)
and basic_literal =
        IntLit of int
      | DoubleLit of float
      | CharLit of char
      | BoolLit of bool
      | ObjectLit of object_literal
      | ListLit of type_spec * expr list

(*** Productions. ***)

(* Type specifiers. *)
and type_spec =
        Int
      | Double
      | Char
      | Bool
      | Void
      | Class of string
      | FuncType of type_spec * type_spec list
      | ListType of type_spec

(* Expressions. *)
and expr =
            Id of string
      | BasicLit of basic_literal
      | FuncLit of func_literal
      | This
      | UnaryOp of op * expr
```

```
      | BinaryOp of expr * op * expr
      | FuncCall of expr * expr list
      | NoExpr

(* Statements. *)

(* Basic type declaration. *)
and basic_init_decl =
        BasicInitDefault of string
      | BasicInitAssign of string * expr

(* Function declaration. *)
and param = type_spec * string
and func_literal = type_spec * param list * stmt

(* Object declaration. *)
and state = string * stmt
and object_literal = type_spec

(* Overall structures of statements. *)
and stmt =
(* Declaration statements. *)
      | BasicDecl of type_spec * basic_init_decl list
      | FuncDecl of type_spec * string * expr
      | ClassDecl of string * state list * stmt list
(* Expression statement. *)
      | Expr of expr
(* Compound statement. *)
```

```
      | CompStmt of stmt list

(* Control flow statement. *)

      | If of expr * stmt * stmt

      | For of stmt * expr * expr * stmt

      | ForEach of  type_spec * string * expr * stmt

      | While of expr * stmt

      | DoWhile of stmt * expr

(* Jump statement. *)

      | Continue

      | Break

      | Return of expr

      | NoStmt


(* input *)

and program = Program of stmt list;;
```

# 8.4 semantic.ml

```
(* Primary Author: Chong Zhang (cz2276), Jiatian Li (jl3930) *)
(*Semantic Check*)
open Ast;;

module NameMap = Map.Make(struct
      type t = string
      let compare x y = Pervasives.compare x y
end)

(*check program*)

(*Environment Details:*)
(*0: Outer Environment*)
(*1: class*)
(*2: while Loop*)
(*3: compond statement*)
(*4: state*)
(*5: if*)
(*6: function*)
(*7: for loop*)

let rec check_semantic program =
```

```
	match program with
	| Program(stmt_list) -> ignore(List.fold_left (check_stmt 0 [(0,"")])
	(
	List.fold_right2
	(fun id t -> NameMap.add id t)
	["printInt"; "printDouble"; "printChar"; "printBool";
"printStr";"nx";"ny";"cellSize";"interval"]
	[
	(FuncType(Void,[Int]), [(0,"")]); (FuncType(Void,[Double]), [(0,"")]);
(FuncType(Void,[Char]), [(0,"")]); (FuncType(Void,[Bool]), [(0,"")]);
(FuncType(Void,[ListType(Char)]), [(0,"")]);
	(Int, [(0,"")]); (Int, [(0,"")]); (Int, [(0,"")]); (Int, [(0,"")])
	]
	NameMap.empty,
	NameMap.empty, NameMap.empty
	)
	stmt_list);true

(*check statements*)

and check_stmt env level (v_table, c_table, s_table) stmt =
	match stmt with (*match all types of statements*)
	| BasicDecl(type_spec, basic_init_decl) ->
		let rec check_basic_init_decl v_table list = (*recursively check
the basic declaration list*)
			match list with
			| [] -> (v_table, c_table, s_table)
			| head::tail ->
			match head with
			| BasicInitDefault(id) ->
					check_basic_init_decl (check_redefine id
type_spec level v_table c_table env) tail
			| BasicInitAssign(id, expr) ->
				if (type_compatable type_spec (check_expr v_table
c_table s_table env level expr)) then
					check_basic_init_decl (check_redefine id
type_spec level v_table c_table env) tail
				else
					raise (Failure("Basic Assignment Check Fails"))
		in check_basic_init_decl v_table basic_init_decl
	| FuncDecl(type_spec, id, expr) ->
		ignore(
			if id = "run" && env = 0 then
				raise(Failure("Cannot Define Run Function"))
			);
		if (check_expr (check_redefine id type_spec level v_table c_table
env) c_table s_table env ((6, id)::level) expr) = type_spec then
			(check_redefine id type_spec level v_table c_table env,
c_table, s_table)
		else
			raise(Failure("Function Type Mismatch"))
	| ClassDecl(id, state_list, stmt_list) ->
		ignore(check_redefine id Void level v_table c_table env);
		if env = 0 then
			match add_s_c_table
			(List.fold_right2 (fun x y -> NameMap.add x y)
["x";"y";"r";"g";"b";"isUpdated"]
```

```
[(Int,(1,id)::level);(Int,(1,id)::level);(Double,(1,id)::level);(Double,(1,id
)::level);(Double, (1,id)::level);(Bool, (1,id)::level)] v_table)
                  (NameMap.add id [("x", Int);("y",
Int);("r",Double);("g",Double);("b",Double);("isUpdated", Bool)] c_table)
(NameMap.add id [] s_table) id state_list stmt_list ((1, id)::level) with
                  | (c_table', s_table') -> (v_table, c_table', s_table')
            else
                  raise (Failure("Cannot Define Class"))

      | Expr(expr) ->
            ignore (check_expr v_table c_table s_table env level expr);
            (v_table, c_table, s_table)
      | CompStmt(stmt_list) ->
            let rec check_comp_stmt v_table' list =
                  match list with
                  | [] -> (v_table, c_table, s_table)
                  | head::tail ->
                        match check_stmt 3
                        (
                              match level with
                              | (6, _)::_
                              | (7, _)::_ -> level
                              | _ -> (3, "")::level
                        )
                              (v_table', c_table, s_table) head with
                        | (v_table'', c_table, s_table) -> check_comp_stmt
v_table'' tail
            in check_comp_stmt v_table stmt_list
      | If(expr, stmt1, stmt2) ->
            ignore (check_stmt env ((5, "")::level) (v_table, c_table,
s_table) stmt1);
            ignore (check_stmt env ((5, "")::level) (v_table, c_table,
s_table) stmt2);
            if (check_expr v_table c_table s_table env level expr) = Bool
then
                  (v_table, c_table, s_table)
            else
                  raise (Failure("If Statement Error"))
      | For(stmt1, expr1, expr2, stmt2) ->
            (
            match stmt1 with
            | BasicDecl(_, _) | Expr(_) | NoStmt ->
                  (
                  match check_stmt env ((7, ""):: level) (v_table, c_table,
s_table) stmt1 with
                  | (v_table, c_table, s_table) ->
                        if (check_expr v_table c_table s_table env level
expr1) = Bool then
                              match check_expr v_table c_table s_table env
level expr1 with
                              | _ -> check_stmt env ((7, ""):: level)
(v_table, c_table, s_table) stmt2
                        else
                              raise(Failure("Expect a Bool Expr in For"))
                  )
            | _ -> raise(Failure("Cannot Define Such Stmt in For"))
            )
```

```
	| ForEach(type_spec, iterator, expr, stmt) ->
		(
		match (check_expr v_table c_table s_table env level expr) with
		| ListType(t) ->
			if t = type_spec then
				check_stmt env ((7, ""):: level) ((check_redefine
iterator type_spec level v_table c_table env), c_table, s_table) stmt
			else
				raise(Failure("Iterator Type Mismatch"))
		)
	| While(expr, stmt1) ->
		ignore (check_stmt env ((2, "")::level) (v_table, c_table,
s_table) stmt1);
		if (check_expr v_table c_table s_table env level expr) = Bool
then
			(v_table, c_table, s_table)
		else
			raise (Failure("Expect a Bool Expr in While"))
	| DoWhile(stmt, expr) ->
		ignore (check_stmt env ((2, "")::level) (v_table, c_table,
s_table) stmt);
		if (check_expr v_table c_table s_table env level expr) = Bool
then
			(v_table, c_table, s_table)
		else
			raise (Failure("Expect a Bool Expr in Dowhile"))
	| Continue ->
		let rec helper list =
			match list with
			| [] -> raise(Failure("Continue Must Be in Loop"))
			| (env, _)::tail ->
				if env = 2 || env = 7 then
					(v_table, c_table, s_table)
				else
					helper tail
		in helper level
	| Break ->
		let rec helper list =
			match list with
			| [] -> raise(Failure("Continue Must Be in Loop"))
			| (env, _)::tail ->
				if env = 2 || env = 7 then
					(v_table, c_table, s_table)
				else
					helper tail
			in helper level
	| Return(expr) ->
		(
		let rec find_func_type list =
			match list with
			| [] -> raise(Failure("Unkown Error"))
			| head::tail ->
				match head with
				| (environment, name) ->
					if environment = 6 then
						match NameMap.find name v_table with
						| (FuncType(func_type,_),_) ->
```

```
                                          if func_type = Void then
                                                raise(Failure("Function Has
No Return"))
                                          else
                                                if (check_expr v_table
c_table s_table env level expr) = func_type then
                                                      (v_table, c_table,
s_table)
                                                else
                                                      raise(Failure("Function
Return Type Mismatch"))
                                    |     _ -> raise(Failure(name^" is not a
Function Type"))
                              else
                                    find_func_type tail
            in find_func_type level
            )
      | NoStmt -> (v_table, c_table, s_table)
      | _ -> raise (Failure("Not Finished"))

(*check expression*)

and check_expr v_table c_table s_table env level expr =
      match expr with (*match all types of expressions*)
      | Id(id) ->
            if NameMap.mem id v_table then
                  match NameMap.find id v_table with
                  | (some_type, _) -> some_type
            else
                  raise (Failure("Cannot Find Identifier " ^ id))
      | BasicLit(basic_literal) ->
            (
            match basic_literal with
            | IntLit(t) -> Int
            | DoubleLit(t) -> Double
            | CharLit(t) ->  Char
            | BoolLit(t) -> Bool
            | ObjectLit(object_literal) -> object_literal
            | ListLit(type_spec, expr_list) ->
                  let rec check_expr_list list =
                        match list with
                        | [] -> ListType(type_spec)
                        | head::tail ->
                              if type_compatable type_spec (check_expr
v_table c_table s_table env level head) then
                                    check_expr_list tail
                              else
                                    raise(Failure("List Element Type
Mismatch"))
                  in check_expr_list expr_list
            )
      | FuncLit(type_spec, param_list, stmt) ->
            let rec get_func_param type_spec_list list v_table' =
                  match list with
                  | [] ->
                        ignore(check_stmt env level (v_table', c_table,
s_table) stmt);
```

```
                        FuncType(type_spec, List.rev type_spec_list)
                  | head::tail ->
                        match head with
                        | (t, name) ->  get_func_param (t::type_spec_list)
tail (check_redefine name t level v_table' c_table env)
            in get_func_param [] param_list v_table
      | This ->
            let rec helper list flg=
                  (
                  match list with
                  | [] -> raise(Failure("Cannot Use This Operator Here"))
                  | head::tail ->
                        (
                        match head with
                        | (1, id) ->
                              if flg = 1 then Class(id) else
raise(Failure("Cannot Use This Operator Here"))
                        | (4, _) -> helper tail 1
                        | _ -> helper tail flg
                        )
                  )
            in helper level 0
      | UnaryOp(op, expr) ->
            (
            match op with
            | Plus
            | Minus ->
                  (
                  let helper t =
                        judge_alg_type t t
                  in helper (check_expr v_table c_table s_table env level
expr)
                  )
            | Not ->
                  match check_expr v_table c_table s_table env level expr
with
                  | Bool -> Bool
                  | _ -> raise(Failure("Type Mismatch"))
            )
      | BinaryOp(e1, op, e2) ->
            (
            match op with
            | Plus | Minus | Mult
            | Div -> judge_alg_type (check_expr v_table c_table s_table env
level e1) (check_expr v_table c_table s_table env level e2)
            | Mod ->
                  (
                  match ((check_expr v_table c_table s_table env level e1),
(check_expr v_table c_table s_table env level e2)) with
                  | (Int, Int)
                  | (Int, Char)
                  | (Char, Int) -> Int
                  | _ -> raise(Failure("Type Mismatch"))
                  )
            | And
            | Or ->
                  (
```

```
                  match ((check_expr v_table c_table s_table env level e1),
(check_expr v_table c_table s_table env level e2)) with
                  | (Bool, Bool) -> Bool
                  | _ -> raise(Failure("Type Mismatch"))
                  )
            | Gt | Ge | Eq | Neq | Le
            | Lt ->
                  judge_logic_type (check_expr v_table c_table s_table env
level e1) (check_expr v_table c_table s_table env level e2)
            | Assign ->
                  let check_left_type el =
                        (*only when the left is an identifier, A.B or A:[B]
can the assignment success*)
                        match el with
                        | Id(id) ->
                              if (NameMap.mem id v_table) then
                                    match (NameMap.find id v_table) with
                                    | (Class(name), _) ->
                                          if (check_expr v_table c_table
s_table env level e2) = Class(name) then
                                                Class(name)
                                          else
                                                raise(Failure("Class
Assignment Fails"))
                                    | (FuncType(arg1, arg2), _) ->
                                          if (check_expr v_table c_table
s_table env level e2) = FuncType(arg1, arg2) then
                                                FuncType(arg1, arg2)
                                          else
                                                 raise(Failure("Function
Assignment Fails"))
                                    | (ListType(arg), _) ->
                                          if (check_expr v_table c_table
s_table env level e2) = ListType(arg) then
                                                ListType(arg)
                                          else
                                                raise(Failure("List
Assignment Fails"))
                                    | (type_spec, _) ->
                                          match (type_spec, (check_expr
v_table c_table s_table env level e2)) with
                                          | (Double, Int) | (Double, Char) |
(Double, Double) -> Double
                                          | (Int, Char) | (Int, Int) -> Int
                                          | (Char, Char) -> Char
                                          | (Bool, Bool) -> Bool
                                          | _ -> raise(Failure("Basic
Assignment Fails"))
                              else
                                    raise (Failure("Cannot Find Identifier
"^id))
                        | BinaryOp(e1', op', e2') ->
                              (
                              match op' with
                              | Dot ->
                                    (
```

```
                                    match ((check_expr v_table c_table
s_table env level e1'), e2') with
                                    | (Class(c_name), Id(id2)) ->
                                    if (find_cls_mem c_table c_name id2) =
check_expr v_table c_table s_table env level e2 then
                                          (find_cls_mem c_table c_name id2)
                                    else
                                          raise(Failure("Cannot Find Class
Member"))
                                    | _ -> raise(Failure("Dot Operation Error
1"))
                                    )
                              | Index ->
                                    (
                                    match ((check_expr v_table c_table
s_table env level e1'), e2') with
                                    | (ListType(t), _) ->
                                          if (check_expr v_table c_table
s_table env level e2) = Int then
                                                t
                                          else
                                                raise(Failure("List Type
Mismatch"))
                                    | _ -> raise (Failure("Index Operation
Error 1"))
                                    )
                              | _ -> raise(Failure("Assignment Fails"))
                              )
                        | _ -> raise(Failure("Assignment Error"))
                  in check_left_type e1
            | Index ->
                  (
                  match ((check_expr v_table c_table s_table env level e1),
e2) with
                  | (ListType(t), _) ->
                              if (check_expr v_table c_table s_table env
level e2) = Int then
                                    t
                              else
                                    raise(Failure("List Type Mismatch"))
                  | _ -> raise(Failure("Index Operation Error 2"))
                  )
            | Trans ->
                  (
                  match(e1, e2) with
                  | (e1, Id(id2)) ->
                        let c_type = check_expr v_table c_table s_table env
level e1 in
                        (
                        match c_type with
                        | Class(c_name) ->
                        if find_cls_state s_table c_name id2 then
                              (*
                              match NameMap.find id1 v_table with
                              | (type_spec, _) -> type_spec
                              *)
                              Void
```

```
                        else
                              raise(Failure("Cannot Find State "^id2))
                        )
                  | _ -> raise(Failure("Trans Operation Error"))
                  )
            | At ->
                  (
                  match(e1, e2) with
                  | (e1, Id(id2)) ->
                        let c_type = check_expr v_table c_table s_table env
level e1 in
                        (
                        match c_type with
                        | Class(c_name) ->
                        if find_cls_state s_table c_name id2 then
                              (*
                              match NameMap.find id1 v_table with
                              | (type_spec, _) -> type_spec
                              *)
                              Bool
                        else
                              raise(Failure("Cannot Find State "^id2))
                        )
                  | _ -> raise(Failure("At Operation Error"))
                  )
            | Index ->
                  (
                  match check_expr v_table c_table s_table env level e1 with
                  | ListType(t) ->
                        if (check_expr v_table c_table s_table env level e2)
= Int then
                              ListType(t)
                        else
                              raise(Failure("Index Number Must Be an
Integer"))
                  | _ -> raise(Failure("Index Operation Error 3"))
                  )
            | Dot ->
                  (
                  match ((check_expr v_table c_table s_table env level e1),
e2) with
                  | (Class(class_name), Id(id2)) -> find_cls_mem c_table
class_name id2
                  | (Class(class_name), FuncCall(f_id, e_list)) ->
                        (
            match f_id with
            | Id(id) ->
                              ignore(find_cls_mem c_table class_name id);
                  (
                              let rec helper list =
                                    (
                                    match list with
                                    | [] -> raise(Failure("Cannot Find
Function in Class"))
                                    | head::tail ->
                                          match head with
                                          | (f_name, f_type) ->
```

```
                                                if f_name = id then
                                                     (
                                                     match f_type with
                                                     | FuncType(type_spec,
type_list) ->
                                                          let rec
check_param type_list expr_list =
                                                               match
(type_list, expr_list) with
                                                               | ([], [])
-> type_spec
                                                              |  (t::tail1,
e::tail2) ->
                                                                    if
(check_expr v_table c_table s_table env level e) = t then
     check_param tail1 tail2
                                                                    else
     raise(Failure("Function Parameter Type Mismatch"))
                                                               | _ ->
raise(Failure("Function Parameter Mismatch"))
                                                          in check_param
type_list e_list
                                                     )
                                                else
                                                     helper tail
                                      )
                                 in helper (NameMap.find class_name c_table)
                                 )
                | _ -> raise(Failure("Function Call Format Error"))
                )
                     | _ -> raise(Failure("Dot Operation Error 2"))
                     )
                | LDot ->
                     (
                     match ((check_expr v_table c_table s_table env level e1),
e2) with
                     | (ListType(list_type), FuncCall(name, e_list)) ->
                          (
                          match (name, e_list) with
                          | (Id("insert"), [e1; e2]) | (Id("set"), [e1; e2]) ->
                               (
                     match ((check_expr v_table c_table s_table env level e1),
(check_expr v_table c_table s_table env level e2)) with
                     | (Int, t) -> if t = list_type then Void else
raise(Failure("Function Argument Type Mismatch"))
                     | _ -> raise(Failure("Function Argument Type Mismatch"))
                               )
                          | (Id("append"), [e1]) ->
                               (
                               match (check_expr v_table c_table s_table env
level e1) with
                               | t -> if t = list_type then Void else
raise(Failure("Function Argument Type Mismatch"))
                               | _ -> raise(Failure("Function Argument Type
Mismatch"))
```

```
                              )
                        |     (Id("remove"), [e1]) | (Id("get"), [e1]) ->
                              (
                              match (check_expr v_table c_table s_table env
level e1) with
                              | Int -> list_type
                              | _ -> raise(Failure("Function Argument Type
Mismatch"))
                              )
                        | (Id("filter"), [e1]) ->
                              (
                              match (check_expr v_table c_table s_table env
level e1) with
                              | FuncType(return_type, arg_list) ->
                                    (
                                    match arg_list with
                                    | t::[] ->
                                          if t = list_type && Bool =
return_type then ListType(list_type) else raise(Failure("Function Argument
Type Mismatch"))
                                    | _ ->      raise(Failure("Function
Argument Mismatch"))
                                    )
                              | _ -> raise(Failure("Function Argument Type
Mismatch"))
                              )
                        | (Id("count"), [e1]) ->
                              (
                              match (check_expr v_table c_table s_table env
level e1) with
                              | FuncType(return_type, arg_list) ->
                                    (
                                    match arg_list with
                                    | t::[] ->
                                          if t = list_type && Bool =
return_type then Int else raise(Failure("Function Argument Type Mismatch"))
                                    | _ ->      raise(Failure("Function
Argument Mismatch"))
                                    )
                              | _ -> raise(Failure("Function Argument Type
Mismatch"))
                              )
                        | (Id("size"), []) -> Int
                        | _ -> raise(Failure("No Such Function"))
                        )
                  | _ -> raise(Failure("LDot Operation Error"))
                  )
            )
      | FuncCall(e1, expr_list) ->
            (
            match e1 with
            | Id(id) ->
                  if id = "run" && env = 0 then
                        match expr_list with
                        | hd::[] ->
                              (
```

```
                              match check_expr v_table c_table s_table env
level hd with
                              | ListType(t) -> (match t with | Class(t) ->
Void | _ -> raise(Failure("Run Function Error")))
                              | _ -> raise(Failure("Run Function Error"))
                              )
                        | _ -> raise(Failure("Run Function Error"))
                  else
                  if NameMap.mem id v_table then
                        match NameMap.find id v_table with
                        | (FuncType(type_spec, type_list),_) ->
                              let rec check_param type_list expr_list =
                                     match (type_list, expr_list) with
                                    | ([], []) -> type_spec
                                    | (t::tail1, e::tail2) ->
                                          if (check_expr v_table c_table
s_table env level e) = t then
                                                check_param tail1 tail2
                                          else
                                                raise(Failure("Function
Parameter Type Mismatch"))
                                    | _ -> raise(Failure("Function Parameter
Mismatch"))
                              in check_param type_list expr_list
                        | _ -> raise(Failure(id^" is not an Function"))
                  else
                        raise(Failure("Cannot Find Function "^id))
            | _ -> raise(Failure("Function Call Format Error"))
            )
      | NoExpr -> Void

(*find the state of the object in state table*)

and find_cls_state s_table id id' =
      if NameMap.mem id s_table then
            let rec find_state list id =
                  match list with
                  | [] -> raise(Failure("Cannot Find State "^id))
                  | head::tail ->
                        if head = id then
                              true
                        else
                              find_state tail id
            in find_state (NameMap.find id s_table) id'
      else
            raise(Failure("Cannot Find Class "^id))

and find_cls_state2 v_table s_table id id' =
      if NameMap.mem id v_table then
            let (c_type, _) = NameMap.find id v_table in
                  match c_type with
                  | Class(c_name) -> find_cls_state s_table c_name id'
                  | _ -> raise(Failure(id^" is not an object"))
      else
            raise(Failure("Cannot Find Object "^id))

(*find the member of the object in Class table*)
```

```
and find_cls_mem c_table id id' =
      if NameMap.mem id c_table then
            let rec find_mem list id =
                  match list with
                  | [] -> raise (Failure("Cannot Find Member "^id))
                  | head::tail ->
                        match head with
                        | (m_id, m_type) ->
                              if m_id = id then
                                    m_type
                              else
                                    find_mem tail id
            in find_mem (NameMap.find id c_table) id'
      else
            raise (Failure("Cannot Find Class "^id))

(*add states and stmts to the table*)

and add_s_c_table v_table c_table s_table id state_list stmt_list level =
      match add_c_table v_table c_table s_table id stmt_list level with
      | (v_table', c_table') ->
            match add_s_table v_table' c_table' s_table id state_list level
with
            | (s_table') ->
            if check_state v_table' c_table' s_table' id state_list level
then
                  (c_table', s_table')
            else
                  raise(Failure("States Error"))

(*check states of the class*)

and check_state v_table c_table s_table id state_list level =
      let rec check_each_state list =
            match list with
            | [] -> true
            | head::tail ->
                  match head with
                  | (s_id, c_stmt) ->
                        match c_stmt with
                        | CompStmt(t) ->
                              (
                              match check_stmt 4 ((4, "")::level) (v_table,
c_table, s_table) c_stmt with
                              | (_,_,_) -> check_each_state tail
                              )
                        | _ -> raise (Failure("Need a Compound Stmt"))
      in check_each_state state_list

(*add states to the state table*)

and add_s_table v_table c_table s_table id state_list level =
      let rec add_state s_table' list =
            match list with
            | [] ->
                  if NameMap.mem id s_table' then
```

```
                        s_table'
                  else
                        NameMap.add id [] s_table'
            | head::tail ->
                  match head with
                  | (s_id, c_stmt) ->
                        match c_stmt with
                        | CompStmt(t) ->
                              add_state
                              (
                              if NameMap.mem id s_table' then
                                    NameMap.add id (s_id::(NameMap.find id
s_table')) s_table'
                              else
                                    NameMap.add id [s_id] s_table'
                              )
                              tail
                        | _ -> raise (Failure("Need a Compound Stmt"))
      in add_state s_table state_list

(*add class to the class table*)

and add_c_table v_table c_table s_table id stmt_list level =
      let rec add_class v_table c_table list =
            match list with
            | [] ->
                  (v_table,
                  if NameMap.mem id c_table then
                        c_table
                  else
                        NameMap.add id [] c_table
                  )
            | head::tail ->
                  match head with
                  | BasicDecl(_, _) | FuncDecl(_, _, _) ->
                        (
                       match check_stmt 1 (level) (v_table, c_table, s_table)
head with
                        | (v_table',_,_) ->
                              let rec add_to_table c_table list1 list2 =
                                    match (list1, list2) with
                                    | ([] ,[]) -> add_class v_table' c_table
tail
                                    | ([], (id2, (t2, l2))::tl2) ->
                                          add_to_table
                                          (
                                          if NameMap.mem id c_table then
                                                NameMap.add id ((id2,
t2)::(NameMap.find id c_table)) c_table
                                          else
                                                NameMap.add id [(id2, t2)]
c_table
                                          )
                                          list1 tl2
                                    | (hd1::tl1, hd2::tl2) ->
                                          if hd1 = hd2 then
                                                add_to_table c_table tl1 tl2
```

```
                                                else
                                                     (
                                                     match (hd1, hd2) with
                                                    |  ((id1,  (t1,  l1)),  (id2,  (t2,
l2))) ->
                                                           if id1 = id2 then
                                                                 add_to_table
                                                                 (
                                                                 if NameMap.mem id
c_table then
                                                                       NameMap.add
id ((id2, t2)::(NameMap.find id c_table)) c_table
                                                                 else
                                                                       NameMap.add
id [(id2, t2)] c_table
                                                                 )
                                                                 tl1 tl2
                                                           else
                                                                 add_to_table
                                                                 (
                                                                 if NameMap.mem id
c_table then
                                                                       NameMap.add
id ((id2, t2)::(NameMap.find id c_table)) c_table
                                                                 else
                                                                       NameMap.add
id [(id2, t2)] c_table
                                                                 )
                                                                 list1 tl2
                                                     )
                                          | _ -> raise(Failure("Unknown Error"))

                              in add_to_table c_table (NameMap.bindings
v_table) (NameMap.bindings v_table')
                        )
                  | _ -> add_class v_table c_table tail
      in add_class v_table c_table stmt_list

(*judge what type should be returned*)

and judge_alg_type t1 t2 =
      match (t1, t2) with
      | (Int, Int) -> Int
      | (Double, Double) -> Double
      | (Char, Char) -> Char
      | (Int, Double) -> Double
      | (Int, Char) -> Int
      | (Double, Int) -> Double
      | (Double, Char) -> Double
      | (Char, Int) -> Char
      | (Char, Double) -> Double
      | _ -> raise(Failure("Type Mismatch"))

and judge_logic_type t1 t2 =
      match (t1, t2) with
      | (Int, Int)
      | (Double, Double)
```

```
      | (Char, Char)
      | (Int, Double)
      | (Int, Char)
      | (Double, Int)
      | (Double, Char)
      | (Char, Int)
      | (Char, Double) -> Bool
      | _ -> raise(Failure("Type Mismatch"))

and type_compatable left right =
      match (left, right) with
      | (Char, Char)
      | (Int, Char) | (Int, Int)
      | (Double, Char) | (Double, Int) | (Double, Double) -> true
      | (a, b) -> a = b

(*check whether there is a name conflict*)

and check_redefine id type_spec level v_table c_table env =
      if NameMap.mem id v_table then
            match NameMap.find id v_table with
            | (_, level') ->
                  if level = level' then
                        raise(Failure("Name Conflict"))
                  else
                        if NameMap.mem id c_table && env = 0 then
                              raise(Failure("Name Conflict"))
                        else
                              NameMap.add id (type_spec, level) v_table
      else
            if NameMap.mem id c_table && env = 0 then
                  raise(Failure("Name Conflict"))
            else
                  NameMap.add id (type_spec, level) v_table
;;
```

# 8.5 translate.ml

```
(* Author: Wei Wang (ww2315), Dale Zhao (dz2242)*)
(* Create translation environment: *)
(* create_env parent_env -> child_env *)
(* Check if current position is contained *)
(* in some loop. *)
(* Linked symbol table. *)
(* *)

open Ast;;
open Str;;

(* translate: string -> Ast.program -> (string * string) list *)

let rec translate sim_name node =
      match node with
      | Program(stmts) ->
```

```
                        (* A MASL program is actually translated into *)
                        (* a subclass of MaslSimulation in Java. *)
                        "public class " ^ sim_name ^ " extends MaslSimulation
{\n" ^
                        (* Make the definition of all MASL classes as nested
Java class,*)
                        (* and make all MASL functions as Java class method.
*)
                        (List.fold_left
                              (fun acc stmt -> acc ^ (translate_stmt "  "
stmt)) ""
                              (List.filter
                                          (fun stmt -> match stmt with
                                    | ClassDecl(_, _, _) -> true
                                                | FuncDecl(_, _, _) -> true
                                    | _ -> false) stmts)) ^
                        "  public void init() {\n" ^
                       (* Everything other than class definition goes into *)
                        (* the method init(). *)
                        (List.fold_left
                              (fun acc stmt -> acc ^ (translate_stmt "    "
stmt)) ""
                              (List.filter
                                          (fun stmt -> match stmt with
                                    | ClassDecl(_, _, _) -> false
                                                | FuncDecl(_, _, _) -> false
                                    | _ -> true) stmts)) ^
                        "  }\npublic static void main(String[] args) {\n" ^
            sim_name ^ " sim = new " ^ sim_name ^ "();\nsim.init();}\n" ^
                        "}\n"
and translate_type_spec_obj node = match node with
      | Int -> "Integer"
      | Double -> "Double"
      | Bool -> "Boolean"
      | Char -> "Character"
      | FuncType(return_type, param_types) -> "MaslFunction<" ^
    (translate_type_spec_obj return_type) ^ ">"
      (*| Class(id) -> "class " ^ id*)
      | Class(id) -> id
      (*| Object -> "object"*)
  | ListType(type_spec) -> "MaslList<" ^ (translate_type_spec_obj type_spec)
^ ">"
      | Void -> "Void"
and translate_type_spec node = match node with
      | Int -> "int"
      | Double -> "double"
      | Bool -> "boolean"
      | Char -> "char"
      | FuncType(return_type, param_types) -> "MaslFunction<" ^
    (translate_type_spec_obj return_type) ^ ">"
      (*| Class(id) -> "class " ^ id*)
      | Class(id) -> id
      (*| Object -> "object"*)
  | ListType(type_spec) -> "MaslList<" ^ (translate_type_spec_obj type_spec)
^ ">"
      | Void -> "void"
and translate_stmt indent node = match node with
```

```
      (* decl_stmt *)
  | BasicDecl(type_spec, decl_list) ->
            let str =
                  (List.fold_left
                       (fun acc decl -> acc ^ (translate_decl type_spec decl)
^ ",")
                        "" decl_list) in
            let decls = String.sub str 0 (String.length str - 1) in
            indent ^ translate_type_spec type_spec ^ " " ^ decls ^
            ";\n"
  | FuncDecl(type_spec, id, expr) -> let return_type =
    begin
      match type_spec with
      | FuncType(rt,_) -> rt
      | _ -> Void (*Impossible, used to suppress warning*)
    end
    in "MaslFunction<" ^ (translate_type_spec_obj return_type) ^ "> " ^
    id ^ "=" ^ (translate_expr expr) ^ ";\n"
      | ClassDecl(id, states, stmts) -> indent ^ "public class " ^
    id ^ " extends MaslClass {\n" ^ (generate_state_update states) ^
    (translate_states states) ^
    (* Add "public" before each statement*)
    (List.fold_left
        (fun acc stmt -> acc ^ "\npublic " ^ (translate_stmt "  " stmt))
        "" stmts
    ) ^ "\npublic String __curState = null;\n" ^
    "public String toString() {\nreturn \"" ^ id ^ "{x:\" + x + \",y:\" + y +
\",r:\" + r + \",g:\" + g + \",b:\" + b + \"}@\" + __curState;\n}\n}\n"
  (*| ObjectDecl(id, expr) -> indent ^ "objectdecl\n"*)
      (* expr_stmt *)
  | Expr(expr) -> indent ^ translate_expr expr ^ ";\n"
      (* comp_stmt *)
  | CompStmt(stmts) ->
            indent ^ "{\n" ^
            (List.fold_left
                       (fun acc stmt -> acc ^ (translate_stmt ("  " ^ indent)
stmt))
                        "" stmts) ^
            indent ^ "}\n"
      (* control_flow_stmt *)
  | If(pred, then_body, else_body) ->
            indent ^ "if(" ^ translate_expr pred ^ ") {\n" ^
            translate_stmt ("  " ^ indent) then_body ^
            indent ^ "} else {\n" ^
            translate_stmt ("  " ^ indent) else_body ^
            indent ^ "}\n"
  | For(init, pred, update, body) ->
            indent ^ "for(" ^ translate_stmt "" init ^
            translate_expr pred ^ ";" ^
            translate_expr update ^ ") {\n" ^
            translate_stmt ("  " ^ indent) body ^
            indent ^ "}\n"
  | ForEach(type_spec, iter, container, body) -> indent ^ "for(" ^
    (translate_type_spec type_spec) ^ " " ^ iter ^ ":" ^
    (translate_expr container) ^ ") {\n" ^ (translate_stmt ("  " ^ indent)
body) ^ "}\n"
  | While(pred, body) -> indent ^ "while(" ^
```

```
            translate_expr pred ^ ") {\n" ^
            translate_stmt ("  " ^ indent) body ^ "}\n"
  | DoWhile(body, pred) -> indent ^ "do {" ^ translate_stmt ("  " ^ indent)
body ^ "}while(" ^ translate_expr pred ^ ");\n"
      (* jump_stmt *)
  | Continue -> indent ^ "continue;\n"
  | Break -> indent ^ "break;\n"
  | Return(expr) -> indent ^ "return " ^ translate_expr expr ^ ";\n"
  | NoStmt -> indent ^ ""
and translate_expr node =
      match node with
      |     Id(id) -> id
      | BasicLit(lit) ->
            begin
            match lit with
            | IntLit(lit) -> string_of_int lit
            | DoubleLit(lit) -> string_of_float lit
            | CharLit(lit) -> "'" ^ (Char.escaped lit) ^ "'"
            | BoolLit(lit) -> string_of_bool lit
            | ObjectLit(lit) -> "new " ^ (translate_type_spec lit) ^ "()"
            | ListLit(type_spec, exprs) -> "new MaslList<" ^
(translate_type_spec_obj type_spec) ^
      ">(" ^ (translate_arg_list exprs) ^ ")"
            end
      | FuncLit(type_spec, param_list, comp_stmt) -> "new MaslFunction<" ^
(translate_type_spec_obj type_spec) ^
        ">() {\n@Override\npublic " ^ (translate_type_spec_obj type_spec) ^
        " invoke(Object... args) {\n" ^
        let idxs = List.rev (List.fold_left (fun acc para -> (List.length
acc)::acc) [] param_list) in
        (List.fold_left2
          (fun acc param idx -> acc ^ (translate_type_spec (fst param)) ^ " "
^ (snd param) ^
          " = (" ^ (translate_type_spec_obj (fst param) ^ ") args[" ^
(string_of_int idx) ^ "];\n")
          )
          "" param_list idxs
        ) ^
        (translate_stmt "  " comp_stmt) ^ (match type_spec with Void ->
"return null;" | _ -> "") ^ "}\n}\n"
      (*| ObjectLit(lit) -> "ObjectLit"*)
      | This -> "this"
      | UnaryOp(op, expr) ->
            (*"(" ^*)
            begin
                  match op with
                  | Plus -> "+"
                  | Minus -> "-"
                  | Not -> "!"
                  | _ -> ""
            end ^
            translate_expr expr (*^ ")"*)
      | BinaryOp(expr1, op, expr2) -> (*"(" ^*)
    begin
      match op with
      | At -> (translate_expr expr1) ^ ".__curState.equals(\"" ^
(translate_expr expr2) ^ "\")"
```

```
      | Trans -> (translate_expr expr1) ^ ".__curState = \"" ^
(translate_expr expr2) ^ "\""
      | Index -> (translate_expr expr1) ^ ".get(" ^ (translate_expr expr2) ^
")"
                  | LDot -> (translate_expr expr1) ^ "." ^ (
        begin
          match expr2 with
          | FuncCall(func, args) -> (translate_expr func) ^ "(" ^
(translate_arg_list args) ^ ")"
          | _ -> "#Impossible#" (*Impossible, used to suppress warning*)
        end)
      | _ ->
                  (translate_expr expr1) ^
                  begin
                        match op with
                        | Plus -> "+"
                        | Minus -> "-"
                        | Mult -> "*"
                        | Div -> "/"
                        | Mod -> "%"
                        | And -> "&&"
                        | Or -> "||"
                        | Gt -> ">"
                        | Ge -> ">="
                        | Eq -> "=="
                        | Neq -> "!="
                        | Le -> "<="
                        | Lt -> "<"
                        | Assign -> "="
                        | Dot -> "."
                        | _ -> ""
                  end ^
                  (translate_expr expr2)
    end (*^ ")"*)
      | FuncCall(func, args) -> (translate_expr func) ^ ".invoke(" ^
(translate_arg_list args) ^ ")"
      | NoExpr -> ""
and translate_decl type_spec decl = match decl with
        BasicInitDefault(id) ->
            id ^ "=" ^
            begin
            match type_spec with
      | Int -> "0"
      | Double -> "0.0"
      | Bool -> "false"
      | Char -> "'\\0'"
      | FuncType(return_type, param_types) -> "null"
      | Class(id) -> "new " ^ id ^ "()"
      | ListType(type_spec) -> "new MaslList<" ^ (translate_type_spec_obj
type_spec) ^ ">();"
      (*| Object -> "#ObjectDefaultValue#"*)
      | Void -> "void"
            end
  | BasicInitAssign(id, expr) ->
            id ^ "=" ^ translate_expr expr
and translate_states states =
  (List.fold_left
```

```
                  (fun acc state -> acc ^ "private void " ^ (fst state) ^ "()
{\n" ^
      translate_stmt "  " (snd state) ^ "}\n")
                        "" states
      )

(*Generate __update function*)
and generate_state_update states = "public void __update() {\n if(" ^
            (List.fold_left
                  (fun acc state -> acc ^ "\"" ^ (fst state) ^
"\".equals(__curState)) {\n" ^
      (fst state) ^ "();\n} else if(")
                        "" states
    ) ^ "true){}\nisUpdated = true;\n}\n"
and translate_arg_list arg_list = let str =
  (List.fold_left
    (fun acc arg -> acc ^ "," ^ (translate_expr arg))
    "" arg_list
  ) in replace_first (regexp ",") "" str
;;
```

## 8.6 toplevel.ml

```
(* Primary Author: Dale zhao (dz2242) *)

type action = Ast | Translate | Compile | Version



let print_ast ast = Astutils.print_program ast;;

let translate_ast translate_name ast = Translate.translate translate_name
ast;;

let compile_ast ast = ();;

let print_version () =

      print_string (

            "\n**** Multi-Agent Simulation Language Compiler v 0.0.1 ****
\n\n" ^

            "Usage:\n\n" ^

            "  masl -a | -t | -c | -v source_file\n\n" ^

            "  -a                 Print the AST of a program.\n\n" ^

            "  -t                 Translate a MASL source program into a
Java\n" ^

            "                     source file (.java).\n\n" ^
```

```
            "  -c               Compile a MASL source program into a Java\n"
^
            "                   class (.java).\n\n" ^
            "  -v               Display the version and usage of this
program.\n\n" ^
            "  source_file      The MASL source file.\n\n");;

print_string "\n";;

let main =
      let action =
            try
                  if Array.length Sys.argv > 1 then
                  List.assoc
                        Sys.argv.(1)
                        [("-a", Ast); ("-t", Translate); ("-c", Compile); ("-
v", Version)]
                  else Version
            with Not_found -> Version in
      let src_name =
            if Array.length Sys.argv > 2 then Sys.argv.(2)
            else "stdin" in
      match action with
      | Ast ->
            print_string ("Printing AST of program " ^ src_name ^ " ...\n");
            let lexbuf =
            if Array.length Sys.argv > 2 then
                  Lexing.from_channel (open_in Sys.argv.(2))
            else
                  Lexing.from_channel stdin in
```

```
            let ast = Parser.program Scanner.token_parser lexbuf in
            print_ast ast;
      | Compile
      | Translate as flag ->
            print_string ("Translating program " ^ src_name ^ " ...\n");
            let lexbuf =
            if Array.length Sys.argv > 2 then
                  Lexing.from_channel (open_in Sys.argv.(2))
            else
                  Lexing.from_channel stdin in
            let ast = Parser.program Scanner.token_parser lexbuf in
            let translate_name =
                  String.sub Sys.argv.(2) 0 (String.index Sys.argv.(2) '.')
in
            let java_src =
                  begin
                  try
                  Semantic.check_semantic ast;
            with Failure msg -> print_string msg; exit 0;
                  end;
                  translate_ast translate_name ast in
            let translate_chn = open_out (translate_name ^ ".java") in
            output_string translate_chn java_src;
            flush translate_chn;
            print_string ("Written to " ^ translate_name ^ ".java.\n");
            if flag == Compile then
                  begin
            print_string ("Compiling program " ^ src_name ^ " ...\n");
                        flush stdout;
```

```
            Unix.system ("javac " ^ translate_name ^ ".java; rm -rf " ^ 
translate_name ^ ".java"); ()

                  end

            else ()

      | Version -> print_version ();;
```

# 8.7 astutils.ml

```
(* Primary Author: Dale zhao (dz2242) *)

open Ast;;


(* A utility that prints out the AST. *)


let rec print_program node = match node with

      | Program(stmts) ->

            print_string "Program {\n";

            List.iter (print_stmt "  ") stmts;

            print_string "}\n"

and print_type_spec indent node = match node with

      | Int -> print_string (indent ^ "Int\n")

      | Double -> print_string (indent ^ "Double\n")

      | Bool -> print_string (indent ^ "Bool\n")

      | Char -> print_string (indent ^ "Char\n")

      | FuncType(return_type, param_types) ->

            print_string (indent ^ "FuncType {\n");

            print_type_spec ("  " ^ indent) return_type;

            List.iter

                        (print_type_spec ("  " ^ indent)) param_types;

            print_string (indent ^ "}\n")

      | Class(id) -> print_string (indent ^ "Class { " ^ id ^ " }\n")
```

```
      | ListType(elem_type) ->
            print_string (indent ^ "ListType {\n");
            print_type_spec ("  " ^ indent) elem_type;
            print_string (indent ^ "}\n")
      | Void -> print_string (indent ^ "Void\n")
and print_stmt indent node = match node with
  | BasicDecl(type_spec, decl_list) ->
      print_string (indent ^ "BasicDecl {\n");
                  print_type_spec ("  " ^ indent) type_spec;
      List.iter (print_basic_init_decl ("  " ^ indent)) decl_list;
                  print_string (indent ^ "}\n")
  | FuncDecl(type_spec, id, expr) ->
            print_string (indent ^ "FuncDecl {\n");
            print_type_spec ("  " ^ indent) type_spec;
            print_string ("  " ^ indent ^ "_Id { " ^ id ^ " }\n");
            print_expr ("  " ^ indent) expr;
            print_string (indent ^ "}\n")
      | ClassDecl(id, states, stmts) ->
            print_string (indent ^ "ClassDecl {\n");
            print_string ("  " ^ indent ^ "_Id { " ^ id ^ " }\n");
            List.iter (print_state ("  " ^ indent)) states;
            List.iter (print_stmt ("  " ^ indent)) stmts;
            print_string (indent ^ "}\n")
  | Expr(expr) ->
            print_string (indent ^ "Expr {\n");
            print_expr ("  " ^ indent) expr;
            print_string (indent ^ "}\n")
  | CompStmt(stmts) ->
```

```
        print_string (indent ^ "CompStmt {\n");
        List.iter (print_stmt ("  " ^ indent)) stmts;
        print_string (indent ^ "}\n")
| If(pred, then_body, else_body) ->
        print_string (indent ^ "If {\n");
        print_expr ("  " ^ indent) pred;
        print_stmt ("  " ^ indent) then_body;
        print_stmt ("  " ^ indent) else_body;
        print_string (indent ^ "}\n")
| For(init, pred, update, body) ->
        print_string (indent ^ "For {\n");
        print_stmt ("  " ^ indent) init;
        print_expr ("  " ^ indent) pred;
        print_expr ("  " ^ indent) update;
        print_stmt ("  " ^ indent) body;
        print_string (indent ^ "}\n")
| ForEach(type_spec, iter, container, body) ->
        print_string (indent ^ "ForEach {\n");
        print_type_spec ("  " ^ indent) type_spec;
        print_string ("  " ^ indent ^ iter);
        print_expr ("  " ^ indent) container;
        print_stmt ("  " ^ indent) body;
        print_string (indent ^ "}\n")
| While(pred, body) ->
        print_string (indent ^ "While {\n");
        print_expr ("  " ^ indent) pred;
        print_stmt ("  " ^ indent) body;
        print_string (indent ^ "}\n")
| DoWhile(body, pred) ->
```

```
            print_string (indent ^ "DoWhile {\n");
            print_expr ("  " ^ indent) pred;
            print_stmt ("  " ^ indent) body;
            print_string (indent ^ "}\n")
  | Continue -> print_string (indent ^ "Continue\n")
  | Break -> print_string (indent ^ "Break\n")
  | Return(expr) ->
            print_string (indent ^ "Return {\n");
            print_expr ("  " ^ indent) expr;
            print_string (indent ^ "}\n")
  | NoStmt -> print_string (indent ^ "NoStmt\n")
and print_expr indent node = match node with
      |     Id(id) -> print_string (indent ^ "Id {" ^ id ^ "}\n")
      | BasicLit(lit) ->
            print_string (indent ^ "BasicLit {\n");
            begin
            match lit with
            | IntLit(lit) ->
                        print_string ("  " ^ indent ^ "IntLit { " ^
(string_of_int lit) ^ " }\n")
            | DoubleLit(lit) -> print_string ("  " ^ indent ^ "DoubleLit { "
^ (string_of_float lit) ^ " }\n")
            | CharLit(lit) -> print_string ("  " ^ indent ^ "CharLit { " ^
(Char.escaped lit) ^ " }\n")
            | BoolLit(lit) -> print_string ("  " ^ indent ^ "BoolLit { " ^
(string_of_bool lit) ^ " }\n")
                  | ObjectLit(lit) ->
                        print_string ("  " ^ indent ^ "ObjectLit {\n");
            print_object_lit ("    " ^ indent) lit;
            print_string ("  " ^ indent ^ "}\n")
```

```
          | ListLit(elem_type, elems) ->
                print_string ("  " ^ indent ^ "ListLit {\n");
                print_string ("    " ^ indent ^ "_ElemType {\n");
    print_type_spec ("      " ^ indent) elem_type;
                print_string ("    " ^ indent ^ "}\n");
    List.iter
                      (print_expr ("    " ^ indent)) elems;
                print_string ("  " ^ indent ^ "}\n")
    end;
    print_string (indent ^ "}\n");
| FuncLit(lit) ->
    print_string (indent ^ "FuncLit {\n");
    print_func_lit ("  " ^ indent) lit;
    print_string (indent ^ "}\n")
| This -> print_string (indent ^ "This\n")
| UnaryOp(op, expr) ->
    begin
          match op with
          | Plus -> print_string (indent ^ "Plus {\n");
          | Minus -> print_string (indent ^ "Minus {\n");
          | Not -> print_string (indent ^ "Not {\n");
          | _ -> ()
    end;
    print_expr ("  " ^ indent) expr;
    print_string (indent ^ "}\n")
| BinaryOp(expr1, op, expr2) ->
    begin
          match op with
          | Plus -> print_string (indent ^ "Plus {\n")
```

```
                  | Minus -> print_string (indent ^ "Minus {\n")
                  | Mult -> print_string (indent ^ "Mult {\n")
                  | Div -> print_string (indent ^ "Div {\n")
                  | Mod -> print_string (indent ^ "Mod {\n")
                  | And -> print_string (indent ^ "And {\n")
                  | Or -> print_string (indent ^ "Or {\n")
                  | Gt -> print_string (indent ^ "Gt {\n")
                  | Ge -> print_string (indent ^ "Ge {\n")
                  | Eq -> print_string (indent ^ "Eq {\n")
                  | Neq -> print_string (indent ^ "Neq {\n")
                  | Le -> print_string (indent ^ "Le {\n")
                  | Lt -> print_string (indent ^ "Lt {\n")
                  | Assign -> print_string (indent ^ "Assign {\n")
                  | Dot -> print_string (indent ^ "Dot {\n")
                  | At -> print_string (indent ^ "At {\n")
                  | Trans -> print_string (indent ^ "Trans {\n")
                  | _ -> ()
            end;
            print_expr ("  " ^ indent) expr1;
            print_expr ("  " ^ indent) expr2;
            print_string (indent ^ "}\n")
      | FuncCall(func, args) ->
            print_string (indent ^ "FuncCall {\n");
            print_expr ("  " ^ indent) func;
            List.iter (print_expr ("  " ^ indent)) args;
            print_string (indent ^ "}\n")
      | NoExpr -> print_string (indent ^ "NoExpr");
and print_basic_init_decl indent node = match node with
```

```
  | BasicInitDefault(id) ->
          print_string (indent ^ "BasicInitDefault {\n");
          print_string ("  " ^ indent ^ id ^ "\n");
          print_string (indent ^ "}\n")
  | BasicInitAssign(id, expr) ->
          print_string (indent ^ "BasicInitAssign {\n");
          print_string ("  " ^ indent ^ "_Id { " ^ id ^ " }\n");
          print_expr ("  " ^ indent) expr;
          print_string (indent ^ "}\n")
and print_func_lit indent node = match node with
      | (return_type, params, body) ->
          print_string (indent ^ "_ReturnType {\n");
          print_type_spec ("  " ^ indent) return_type;
          print_string (indent ^ "}\n");
          List.iter
                (fun (param_type, param_id) ->
                      print_string (indent ^ "_Param {\n");
                      print_type_spec ("  " ^ indent) param_type;
                      print_string ("  " ^ indent ^ param_id ^ "\n");
                      print_string (indent ^ "}\n"))
                params;
          print_stmt indent body
and print_state indent node =
      print_string (indent ^ "_State {\n");
      begin
      match node with
      | (state_id, body) ->
          print_string ("  " ^ indent ^ "_Id { " ^ state_id ^ " }\n");
          print_stmt ("  " ^ indent) body
```

```
      end;
      print_string (indent ^ "}\n")
and print_object_lit indent class_id =
            print_type_spec ("  " ^ indent) class_id;
;;
```