Columbia University

Computer Science Department

COMS W4115 Programming Languages and Translators

CAL: Cellular Automaton Language

Authors:
Calvin Hu (ch2880)
Nathan Keane (nak2126)
Eugene Kim (esk2152)

Supervisor:
Prof. Stephen A. Edwards
Teaching Assistant: Qiuzi `Jessie' Shangguan

August 16, 2013

1 INTRODUCTION

1.1 Background

Cellular automata are discrete, abstract computational systems that provide useful models of non-linear dynamics in various scientific fields. Cellular automata are composed of discrete, homogenous units called cells and each cell possesses one of a finite number of states, which changes at discrete time steps simultaneously with the states of the other cells. State change of a cell is governed by the states of its immediately surrounding cells. Cellular automata can display complex, evolving behavior of small, homogenous units following set rules and are used to study computation, algorithmic problems, pattern formation, abstract complexity science and theoretical biology.

1.2 Project Overview

Our language, Cellular Automata Language (CAL), is intended for programmers to quickly and easily design cellular automata suited for their use. It should be easy for the programmers to designate the set of initial states and set rules associated with their own cellular automata. They should be able to see the outcome after a specific number of steps in both textual and graphical formats.

CAL makes use of a limited number of primitives to allow easy instantiation of rules and states. State of an entire cellular automaton will be encapsulated in a primitive called *grid*. Initial state of cellular automata will be set either manually in code or via input files using system function such as "def grid create_grid(int width, int height)." We have a primitive called *direction*, which is a property associated one of the eight neighboring cells as well as the cell itself (north, south, east, west, northwest, southwest, northeast, southeast and center). A key datatype for CAL is called *actor_type* - a data type that describes the internal variables and rules for how an actor should act. An actor is the property of a cell at a given time that has a distinct property of what its next move will be depending on the current states of its neighboring cells. CAL syntax allows programmers to declare a rule succinctly through this *actor_type*.

In addition, we keep many of the basic data types like int, char, bool and string used in popular languages such as C, C++ and Java to allow flexibility and ease of learning for programmers. Arithmetic and boolean operators used in popular languages are mostly retained and have identical properties as programmers are used to. CAL also implements *if else* conditional statements and *while* loops, but our hope is that programmers will rarely have to use these features in programming cellular automata due to our new powerful data types.

As part of our project, our team built a scanner, parser, ast file, semantic analyzer and code generator, which altogether will scan, parse, error-check and generate correct C code that can be turned into corresponding executable files through a C compiler.

1.3 Language Features

- CAL can build a grid of designated type and set its initial stage through easy system functions.
- CAL can set rules for each actor_type through its succinct syntax.
- CAL can also set initial variables associated with each actor_type.
- CAL can display the built cellular automaton with a designated time interval.
- CAL can change the grid and cell size as well as the screen size.

2 LANGUAGE TUTORIAL

2.1 Getting Started with the Compiler

Before running our compiler, configure your environment first by installing proper OCaml package for your system by visiting: http://caml.inria.fr/download.en.html. CAL compiler also require C SDL package, which can be obtained from http://www.libsdl.org/ . Finally, your system should have gcc installed in your system.

2.2 Installing and Compiling the Compiler

After the procedure above, please place our project folder named 'CAL.tar.gz' into your system, which contains source files. You need to first compile our CAL compiler by running the following command inside the project folder.

make clean
make

After doing so, it will produce the compiler binary. Run the following to compile your CAL source file.

./cal.out <source_file>

This will produce an executable with a ".out" extension, as well as an intermediate C source file. The executable can be run with:

./<output_file>

2.3 A first example of the CAL program

2.3.1 CAL Program: brians_brain.cal

```
1   actor_type Off = |
2           init:
3
4   rules:
5                   neighborhood(On) == 2 => {
6                           assign_type(center, On);
7                   }
8                   default => { }
9
10  |
11
12  actor_type On = |
13          init:
14
15  rules:
16                  default => { assign_type(center, Dying); }
17
18  |
19
20  actor_type Dying = |
21          init:
22
23          rules:
24                  default => { assign_type(center, Off); }
25
26  |
27
28  def void setup(){
29          grid_size(300, 300);
30          set_chronon(10);
31          set_cell_size(2);
32  }
```

This is a CAL program to produce a cellular automaton called "Brian's Brain", which consists of a two-dimensional grid of cells and each cell may be in one of three states: *On*, *Dying* or *Off*. In each time step, a cell turns on if it was off but had exactly two neighbors that were on. All cells that were *On* go into the *Dying* state, which is not counted as an *On* cell in the neighbor count, and prevents any cell from being born there. Cells that were in *Dying* state go into the *Off* state.

Line 1-10: Declaration of actor_type *Off*.
Line 2-3: Declaration and initialization of local variables (none used in this program).
Line 4-9: Declaration of rules for actor_type *Off*.

Line 5-6: Rule declared that if a cell currently subject to an actor_type *Off* has exactly two neighboring cells of type of *On*, then the cell (center) will be assigned an actor_type *On* in the next time step.
Line 8: Default rule (none declared here) if none of the rules applies.
Line 12-19: Declaration of rules for actor_type *On*.
Line 16: Default rule that a cell subject to an actor_type *On* will always be subject to actor_type *Off* in the next time step. This default rule will always apply because there is no other rule declared for actor_type *On*.
Line 20-26: Declaration of rules for actor_type *Dying*.
Line 24: Default rule that a cell subject to an actor_type *Dying* will always be subject to actor_type *Off* in the next time step. This default rule will always apply because there is no other rule declared for actor_type *Dying*.
Line 29: Set grid size to 300 by 300 cells.
Line 30: Set time interval to minimum 10 milliseconds.
Line 31: Set each cell size to 2 pixels.

2.3.2 Compile the Program

To compile the program, use the following command:

./cal brians_brains.cal

2.3.3 Running the Program

Use the following command to execute the program:

./brians_brains.out

2.3.4 Result

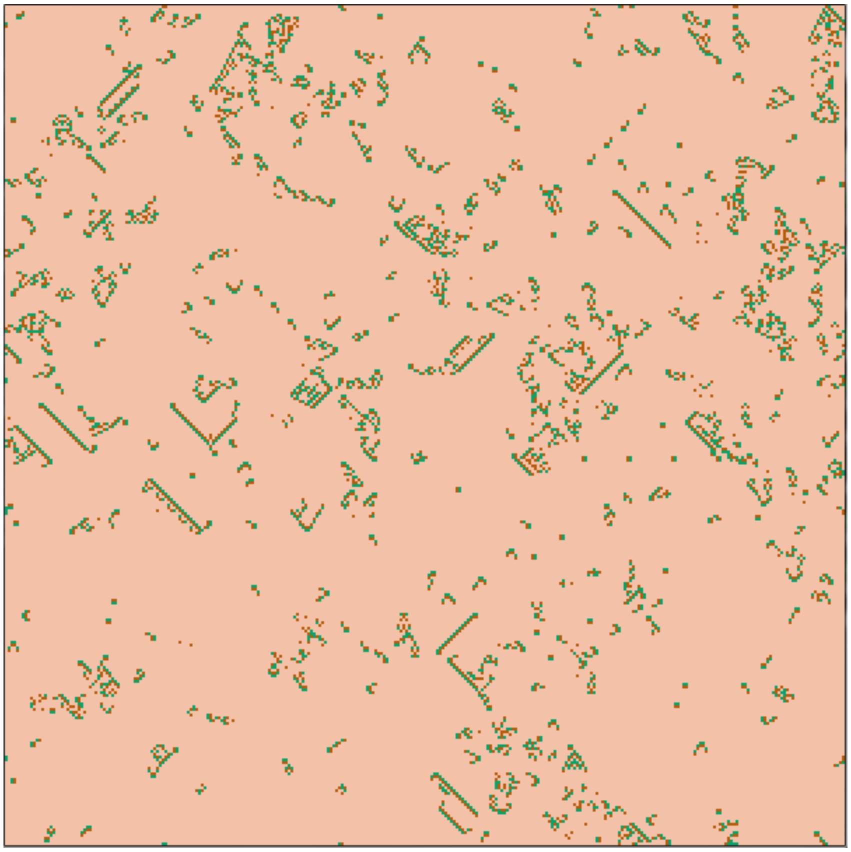
The graphical output (a snapshot taken) of the above program is as below in Figure 1. Colors of different actor_types are randomly chosen.

2.4 Additional Examples

2.4.1 Langton's Ant

Langton's ant is a two-dimensional cellular automaton with a very simple set of rules but complicated emergent behavior. Cells are initially assigned Black or White randomly and one cell is designated as the ant. The ant can travel in any of the four directions (N, S, W, E) at each time step. The ant moves according to the rules below:

- At a white square, turn 90° right, flip the color of the cell, move forward one unit
- At a black square, turn 90° left, flip the color of the cell, move forward one unit



<Figure 1>

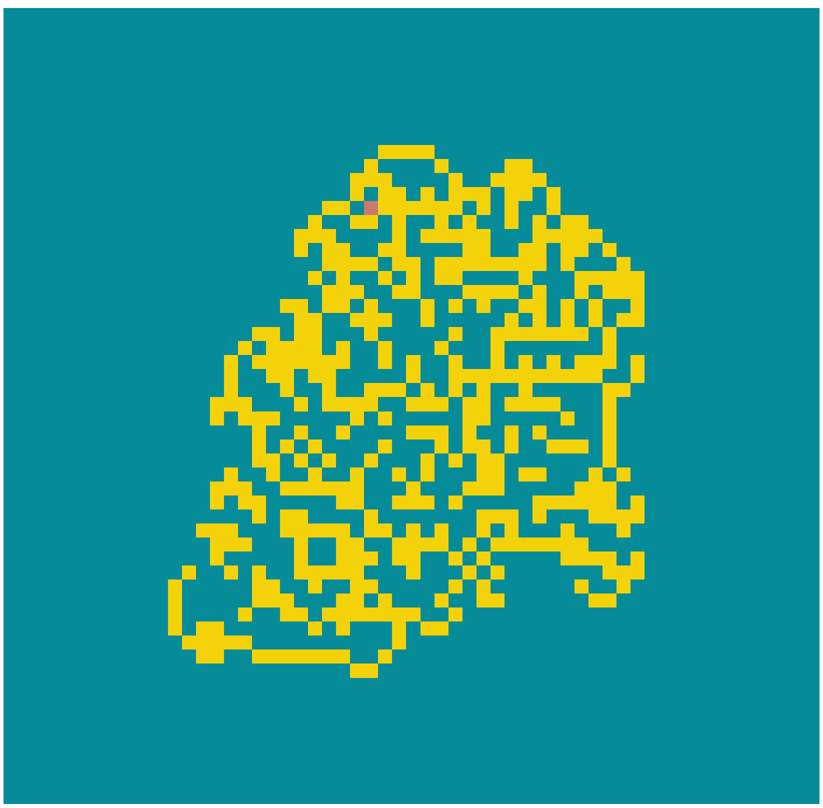
The following is the program that implements the above Langton’s Ant cellular automaton.

```
actor_type White = |
        init:

rules:
                default => { }

|

actor_type Black = |
        init:

rules:
```

```
            default => { }
|



actor_type Ant = |
    init:
            actor_type atype = White;
            direction ant_dir = north;

      rules:
            (cellat(ant_dir) == White) => {
                  if(ant_dir == north){
                        ant_dir = east;
                        move(north, atype);
                  }else if(ant_dir == east){
                        ant_dir = south;
                        move(east, atype);
                  }else if(ant_dir == south){
                        ant_dir = west;
                        move(south, atype);
                  }else{
                        ant_dir = north;
                        move(west, atype);
                  }
                  atype = Black;
                  printf("Direction: %d, Atype: %c\n", ant_dir, atype);
            }
            default => {
                  if(ant_dir == north){
                        ant_dir = west;
                        move(north, atype);
                  }else if(ant_dir == west){
                        ant_dir = south;
                        move(west, atype);
                  }else if(ant_dir == south){
                        ant_dir = east;
                        move(south, atype);
                  }else{
                        ant_dir = north;
                        move(east, atype);
                  }
                  atype = White;
```

```
                              printf("Direction: %d, Atype: %d\n", ant_dir, atype);
                    }

|


def void setup(){
          grid_size(200, 200);
          set_chronon(1);
          set_cell_size(4);
          set_grid_pattern(3, Black, White, 200, 200, 0, 0);
          set_actor(75, 75, Ant);
}
```

The following <Figure 2> is a snapshot of the graphical output of the above program:



<Figure 2>

2.4.2 Rule 90

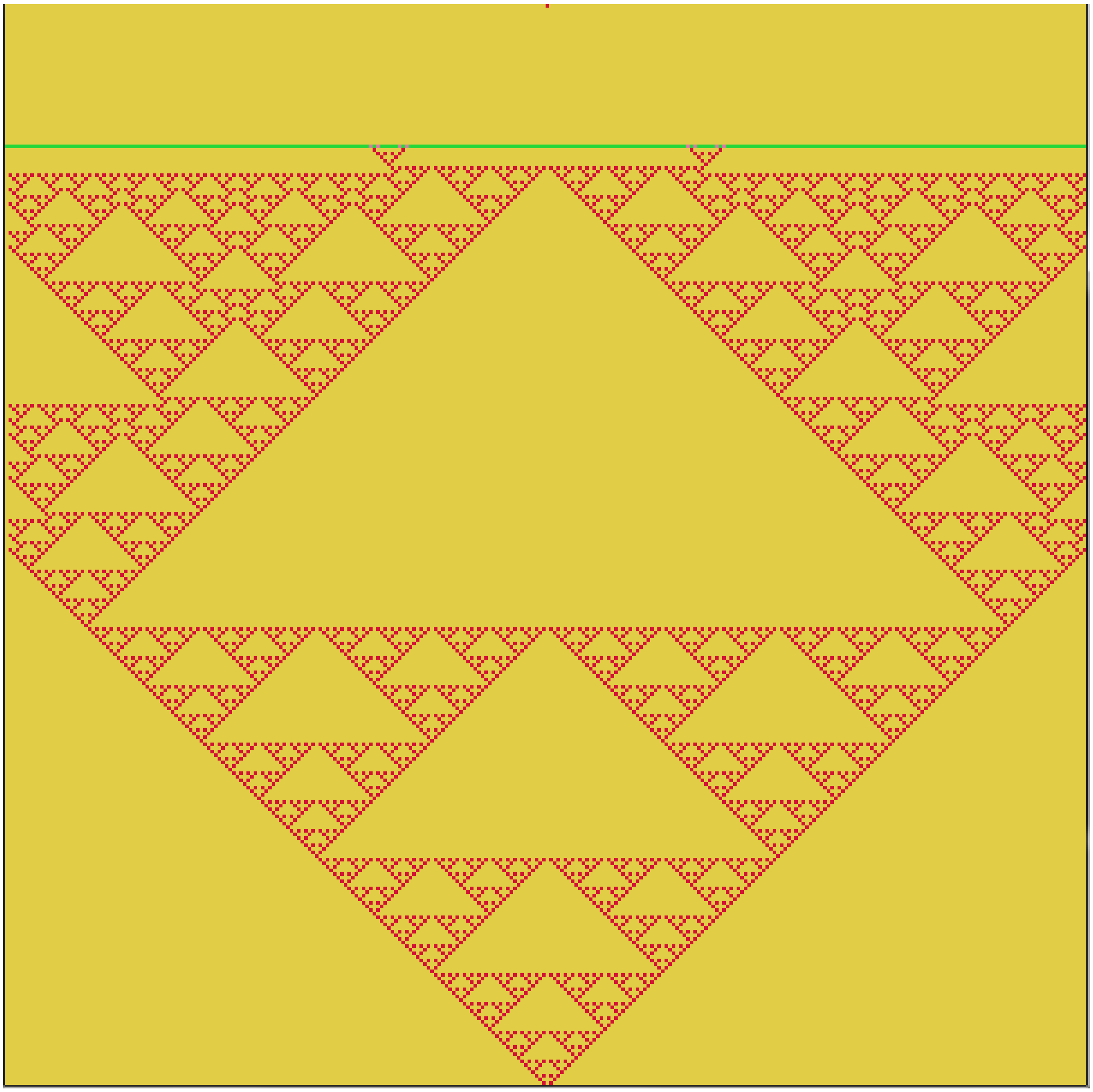
Rule 90 is a one-dimensional cellular automaton based on the exclusive or function. Each cell can hold either a 0 or a 1 value and at each time step all values are simultaneously replaced by the exclusive or of the two neighboring values.

The following is the program that implements the above Rule 90 cellular automaton.

```
actor_type On = |
        init:

        rules:
           cellat(west) == On  && cellat(east) == On => {
                          assign_type(center, SetOn);
                 assign_type(south, Off);
                 }
           cellat(west) == On  && cellat(east) == Off => {
                          assign_type(center, SetOn);
                 assign_type(south, On);
                 }
           cellat(west) == Off  && cellat(east) == On => {
                          assign_type(center, SetOn);
                 assign_type(south, On);
                 }
           cellat(west) == Off  && cellat(east) == Off => {
                          assign_type(center, SetOn);
                 assign_type(south, Off);
           }
        default => { }
|

actor_type Off = |
     init:

        rules:
           cellat(west) == On && cellat(east) == On => {
                          assign_type(center, SetOff);
                 assign_type(south, Off);
                 }
           cellat(west) == On && cellat(east) == Off => {
                          assign_type(center, SetOff);
                 assign_type(south, On);
                 }
           cellat(west) == Off && cellat(east) == On => {
                          assign_type(center, SetOff);
                 assign_type(south, On);
```

```
                }
            cellat(west) == Off && cellat(east) == Off => {
                        assign_type(center, SetOff);
                assign_type(south, Off);
                }
            default => { }
|

actor_type SetOn = |
        init:

        rules:
                default => { }
|

actor_type SetOff = |
        init:

        rules:
                default => { }
|

def void setup(){
    grid_size(300, 300);
    set_cell_size(2);
    set_grid_pattern(3, SetOff, On, 300, 300, 0, 0);
    set_grid_pattern(3, Off, On, 300, 1, 0, 0);
    set_actor(150, 0, On);
    set_chronon(20);
}
```

The graphical output (a snapshot taken) of the above program is as below in Figure 3. Colors of different actor_types are randomly chosen.



<Figure 3>

2.4.3 Game of Life

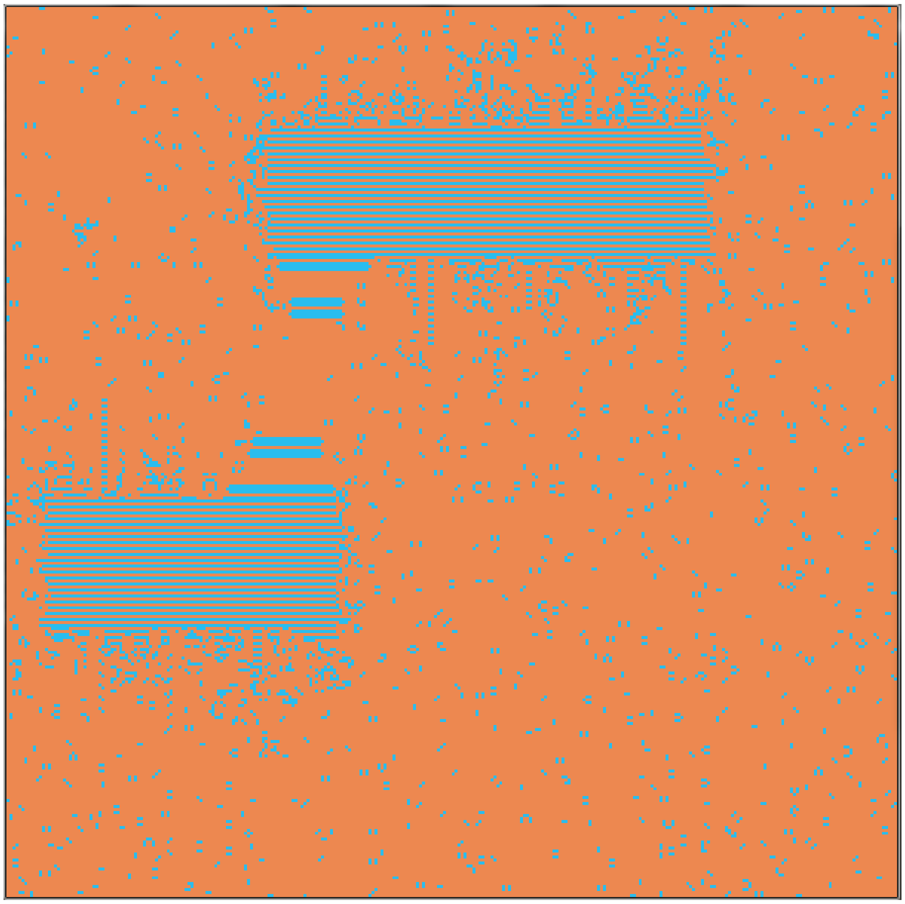
Game of Life cellular automaton consists of a grid of cells which can either be Live or Dead. At each time step, one of the following transition logic will apply:

- Any live cell with fewer than two live neighbours dies, as if caused by under-population.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overcrowding.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The following is the program that implements the Game of Life cellular automaton.

```
actor_type Live = |
	init:

	rules:
		neighborhood(Live) < 2 => {assign_type(center, Dead);}
		neighborhood(Live) == 2 || neighborhood(Live) == 3 => {assign_type(center, Live);}
		neighborhood(Live) > 3 => {assign_type(center, Dead);}

		default => { }
|

actor_type Dead = |
	init:

	rules:
		neighborhood(Live) == 3 => {assign_type(center, Live);}

		default => { }
|

def void setup(){
	grid_size(300, 300);
	set_chronon(20);
	set_cell_size(2);
	set_actor(50, 50, Dead);
	set_grid_pattern(1, Dead, Live, 125, 125, 75, 0);
	set_grid_pattern(1, Dead, Live, 125, 125, 125, 0);
	set_grid_pattern(1, Dead, Live, 125, 125, 0, 125);
}
```

The graphical output (a snapshot taken) of the above program is as below in Figure 4. Colors of different actor_types are randomly chosen.



<Figure 4>

3 LANGUAGE REFERENCE MANUAL

3.1 Introduction

Cellular automata are discrete, abstract computational systems that provide useful models of non-linear dynamics in various scientific fields. Cellular Automata Language (CAL) is intended for programmers to quickly and easily design cellular automata suited for their use. Programmers can easily designate the set of initial states and set rules associated with their own cellular automata and see the outcome after a specific number of steps in both textual and graphical formats. State of an entire cellular automaton will be encapsulated in a primitive called Grid. CAL allows programmers to declare a rule succinctly and efficiently by using CAL's unique syntax.

3.2 Lexical Conventions

3.2.1 Comments

A double slash - "//" comments out text to the right on the same line.

```
//This is a comment.
bool b = true;  //this is also a comment.
```

3.2.2 Identifiers (Names)

An identifier is a sequence of letters and digits. The first character must be alphabetic. The underscore counts as alphabetic. Names are case sensitive. Only the first 30 characters are guaranteed to be significant.

3.2.3 Keywords

The following identifiers are reserved for use as keywords:

```
if
else
true
false
while
bool
char
int
string
grid
direction
north
south
east
west
northwest
southwest
northeast
southeast
center
rules
return
def
init
this
actor_type
```

3.2.4 Constants and Literals

Our language will provide functionality for literals of type bool, char, int, string, rule and grid. If any of these literals are assigned to a variable, the variable's declared type must match the type of the literal.

3.2.4.1 Boolean constants

The two boolean literals are the usual true and false. Examples:

```
true;
false;
bool x = true;
if (x) {
        //code
}
```

3.2.4.2 Character constants

Character literals in CAL are standard ASCII- they are nested in between single quotes. Example:

```
‘a’;
‘_’;
char ch = ‘a’; // ‘a’ is a literal;
```

3.2.4.3 Integer constants

An integer constant consists of a sequence of digits. Examples:

```
45;
-1;
int a = 2;  // 2 is the literal
```

Cal only provides support for decimal representation of integers.

3.2.4.4 Actor_type constants

Actor_types are types of actors in the cellular automaton. They are used to create and configure actors in the grid. The init block allocates variables as well as initializes them in the configuration appropriate when the actor is created. The rules block takes a series of conditions and resulting transition logic. Loops are not allowed within transition blocks.

```
actor_type a1 =|
        init:
                <datatype> <variable_name>;
                <datatype> <variable_name2>;
        rules:
                <condition> => {
                        <transition_block>
                 }
                default =>{
                        <transition_block>
```

}

|

3.2.4.5 Direction constants

The direction datatype has 9 possible values: this, north, south, east, west, northeast, northwest, southeast, southwest

example:
direction d = north;

3.3 Data Types

CAL will support the following primitive data types:

int - an integer
char - a character
bool - a boolean
string - a character string
grid - a n by m array of characters that represents a cellular automata grid
direction - a direction (N,S,E,W,NW,NE,SW,SE) associated with a neighboring cell type.
actor_type - a data type that describes the internal variables and rules for how an actor should act.

3.4 Expressions and Operators

3.4.1 Unary Operators - group right-to-left

- <expression>
    Operand must be a char or int. Returns negative of that value.

3.4.2 Boolean Operators - group right-to-left

<expression> && <expression>
    Both of the operands must be of type bool. Returns true if both are true, false otherwise.

<expression> || <expression>
    Both of the operands must be of type bool. Returns true if one of the operands is true

<expression> == <expression>
    Both of the operands must be of the same type, either bool,char,int, direction or string. Returns true if both operands are bit-wise equal, false otherwise.

<expression> != <expression>

Both of the operands must be of the same type, either bool, char, int, direction or string. Returns false if both operands are bit-wise equal, true otherwise.

3.4.3 Additive Operators - group left-to-right

<expression> + <expression>

Both of the operands must be of the same type, either an int or string. Returns the sum of the two operands if it is an int, and the concatenation if is two strings.

<expression> - <expression>

Both of the operands must be of the same type, int. Returns subtraction of left from right.

3.4.4 Multiplicative Operators - group left-to-right:

<expression> / <expression>

Both of the operands must be of type int. Returns integer division.

<expression> * <expression>

Both of the operands must be of type int. Returns integer multiplication.

<expression> % <expression>

Both of the operands must be of type int. Returns the remainder from the division of the first by the second.

3.4.5 Relational Operators - group left-to-right:

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

Both of the operands must be of the same type int, char, string. Compares bit-wise relations.

3.4.6 Assignment Operator

lvalue = <expression>

The value of the expression replaces that of the object referred to by the lvalue with a deep copy.

3.5 Statements

3.5.1 Expression Statement

An expression statement is any expression consisting of variables, constants, operators and functions followed by a semicolon.

3.5.2 If Statement

The *if* statement is executed conditionally based on the boolean value of a test expression. The test expression has to be of bool type. When the test expression evaluates to true, then the statement following keyword *if* will be executed. Otherwise, the statement following keyword *else* will be executed. Else clause is not optional in *if* construct. You can use a series of *if/else if/else* statements to test for multiple conditions. But only the first statement whose test condition evaluates to true will be executed. The following are two general forms of the *if* statement:

(1)

```
if (expression)
        {statement}
else
        {statement}
```

(2)

```
if (expression)
        {statement}
else if (expression)
        {statement}
else if (expression)
...
else
        {statement}
```

3.5.3 While Loops

The *while* statement allows multiple execution of a statement as long as the test expression evaluates to true. The test expression has to be of bool type. The following is the general form of the while statement.

```
while (expression)
        {statement}
```

3.5.4 Return Statement

The *return* statement is used by a function to return program control and a value to the function that called it. The following is a general form of the *return* statement.

return value;

3.6 Scope Rules

Scope defines the region of a program in which an identifier is visible. It is illegal to refer to identifiers unless they have been declared. Identifiers declared at the top-level of a file is visible to the entire file. Declarations made inside functions are only visible within those functions.

3.7 Declarations

3.7.1 Variable Declarations

Variables must be declared and initialized before they can be used. Variable declarations are in the following form where type can be one of the following types: bool, char, int, string, grid and actor_type.

type identifier = initialization expression

3.7.2 Function Declarations

Functions must be declared and initialized before they can be used. A functions is declared with the keyword def followed by a return type, function identifier and a list of arguments each preceded by its type and separated by commas in a parenthesis. The general form of function declarations is as follows.

def type identifier (type argument1, type argument2, …)

3.8 System Functions

def void move(direction d, actor_type a) - Can only be used in a transition block. Moves actor to neighboring cell location in the given direction parameter and leaves an actor of type a in its previous location.

def void assign_type(direction d, actor_type a) - Can only be used in a transition block. Assigns the actor at the cell in direction d to actor_type a.

def void set_actor(int x, int y, actor_type a) - Assigns the actor_type at the cell at location (i,j).

def actor_type cellat(direction d) - Can only be used in a transition block. Returns the actor_type in the neighboring cell location in the given direction parameter.

def int neighborhood(actor_type a) - Can only be used in a transition block. Returns the number of actors of type a in the neighboring cell locations.

def actor_type cellat(direction) - Can only be used in a transition block. Given a direction relative to the actor calling the function, returns the actor in the cell location at the position.

def direction randomof(actor_type a) - Can only be used in a transition block. Returns a direction of a random cell in the neighborhood that contains an actor of the type given.

def int random(int upper) - Returns a random integer from the range 0 to upper exclusive.

def grid_size(int width, height) - Creates a grid with the given height and width and fills it randomly with the declared actor_type.

def void set_chronos(int milliseconds) - set each chronos step at x milliseconds.

def void set_cell_size(int size) - set size each cell.

def void set_grid_pattern(int pattern_type, actor_type actor_a, actor_type, actor_b, int width, int height, int x, int y) - Sets the pattern at location (x,y) in the grid with int x and int y.

Patterns:
- 0 - Checkerboard with actor_a and actor_b
- 1 - Alternating rows with actor_a and actor_b
- 2 - Alternating columns with actor_a and actor_b
- 3 - Fill with actor_a

3.9 Example

Wa-Tor

```
grid g = create_grid(100, 100);
int i = 0;
int j = 0;
int type = 0;

actor_type Free = |
	init:

	rules:
		default => { }

|

actor_type Fish = |
```

```
	init:
		int counter = 0;

	rules:
		counter <= 10 && neighborhood(Free) > 0 => {
			move(randomof(Free), Free);
		}
		counter > 10  && neighborhood(Free) > 0 => {
			assign_type(randomof(Free), Fish);
			counter = 0; }
		default => { counter = counter + 1; }

|

actor_type Shark = |
	init:
		int counter = 0;
		int energy = 10;

	rules:
		neighborhood(Fish > 0) => {
			move(randomof(Fish), Free);
			energy = energy + 1;
		}
		neighborhood(Fish <= 0) && neighborhood(Free >0) && counter <= 10 =>{
			move(randomof(Free), Free);
			energy = energy - 1;
		}
		neighborhood(Fish <= 0) && counter > 10  && neighborhood(Free > 0) => {
			assign_type(randomof(Free), Shark);
			energy = energy - 1;
			counter = 0;
		}
		energy = 0 =>{
			assign_type (this, Free);
		}
		default => {
			counter = counter + 1;
			energy = energy - 1;
		}

|
```

```
while(i < 100){
	while(j < 100){
		type = random(3);
		if(type == 0){
			assign_type(g,i,j,Shark);
		}
		else if{
			assign_type(g,i,j,Fish);
		}
		else{
			assign_type(g,i,j,Free);
		}

	}
}

run(g, 100);
display(g);
```

4 PROJECT PLAN

4.1 Process

We formed our group during the first class of the PLT course based on our availability, which was an important factor in hindsight. Due to the quick timing of the summer semester, we had to set a relatively strict timeline for the progress of our project and ensure that we adhere it throughout the semester. A regular meeting was important. We met every week, mostly two or three times a week at a computer lab on campus to ensure that we could work next to each other. Proximity of working near each other allowed us to code in parallel, debug collectively and make sure we were all accomplishing our deliverables. We found that working together increased our productivity by multiples. Most of the planning, specification, development and testing were done in person together, allowing smooth communication and quick feedback.

In order to efficiently collaborate, we established a github for the project and made sure we git add/commit/push/pull regularly and checked each other's progress and code. Some of us were not familiar with github initially but we found it extremely useful and powerful as the project progressed.

4.2 Programming Style

In general, we followed the programming style that was used in MicroC example provided in class.

4.2.1 Names

Function names are lower-cased and underscores are used to separate words.

4.2.2 Function Definitions

The first line of a function declaration should end with the word function if it spans multiple lines. Comments describing the function precedes the function definitions.

4.2.3 Indentation

In order to increase readability, we indented the body of large blocks according to the nested structure. If a block is multi-nested, it was indented multiple tabs according to the depth of the nested structure.

4.3 Project Timeline

2013/7/8	Team formed
2013/7/12	Language defined
2013/7/15	Language proposal submitted
2013/7/18	TA feedback on proposal received
2013/7/19	Proposal modified per TA's feedback
2013/7/24	Language reference manual submitted
2013/7/29	Scanner completed
2013/8/8	Parser and AST completed
2013/8/10	Scanner, Parser and AST tested
2013/8/12	C code generation completed
2013/8/13	Write sample programs
2013/8/15	Semantic analyzer completed
2013/8/15	Testing suite completed
2013/8/16	Final report/presentation

4.4 Responsibility

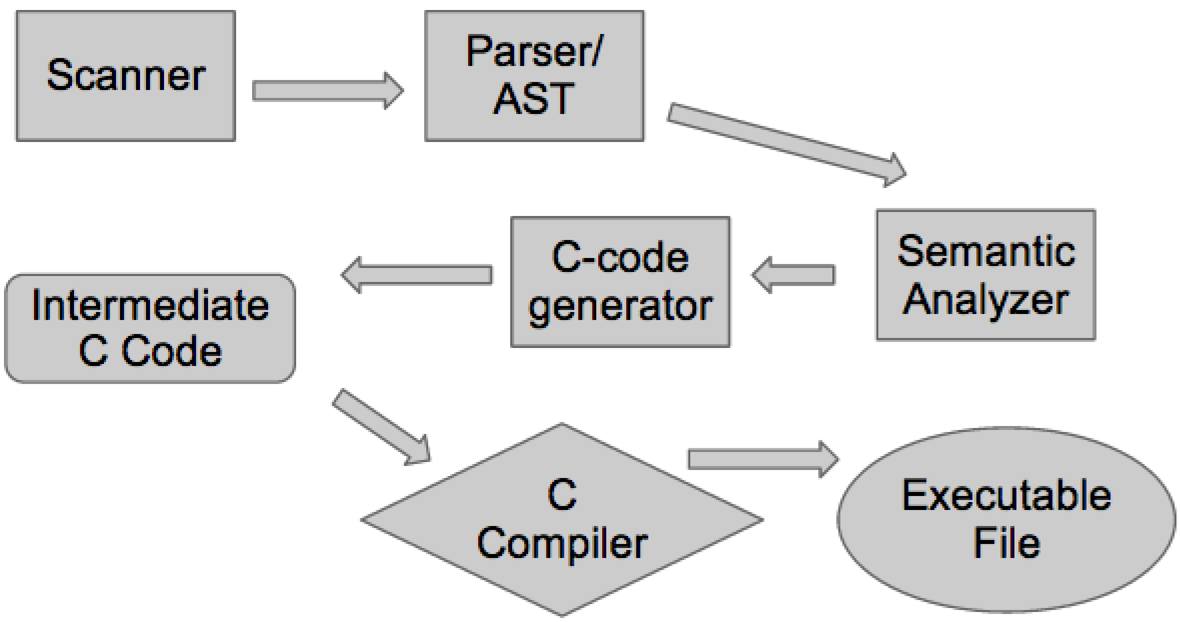
As discussed above, we collaborated on most aspects of the project with each person taking a lead on different components of the project. Eugene Kim took the lead on the scanner, CAL sample programs, presentation and final report. Calvin took the lead on C code generation, compilation and graphical aspects of the project in addition to coming up with the project idea. Nathan Keane oversaw the parser, AST and semantic analyzer as well as testing. All of us collaborated on the proposal and language reference manual.

4.5 Software Development Environment

Project was developed on Windows and Mac machines using the following components:
OCaml v4.00.1, Ocamllex, Ocamlyacc, C, C SDL library, GitHub, Makefile.

5 ARCHITECTURAL DESIGN

5.1 CAL Architectural Design Diagram



5.2 CAL Components

5.2.1 Scanner

The scanner component reads a CAL program and turns it into a stream of tokens (ignoring white spaces and comments), which is then passed onto the parser. It will reject any CAL program that does not conform to the syntax of our language.

5.2.2 Parser and AST

The stream of tokens passed from the scanner will be parsed into an abstract syntax tree according to the properties of each parsed object. Parser will produce error if it cannot produce a meaningful, grammatical abstract syntax tree with the given tokens.

5.2.3 Semantic Analyzer

Semantic analyzer will check for duplicate variables, duplicate functions, validity of expressions and validity of statements as well as format of function declarations. If there is an error, semantic analyzer will throw an error statement, indicating what type of error has occurred.

5.2.4 C Code Generator

Code generator will take the ast object passed from parser and convert it to a valid C program, which will then be compiled into an executable C file via a C compiler.

6 TEST

6.1 Testing of AST

This was to test semantic errors. A python script was written that piped the standard error of each .cal test file with a specific semantic error to an appropriately named result file. The error was then checked to see if it was matched with what we expected. For example the file "test_case1.cal" :

```
actor_type Live = |
      init:

      rules:
            neighborhood(Live) < 2 => {assign_type(center, Dead);}
            neighborhood(Live) == 2 || neighborhood(Live) == 3 => {assign_type(center, Live);}
            neighborhood(Live) > 3 => {assign_type(center, Dead);}

            default => { }
|

actor_type Dead = |
      init:

      rules:
            neighborhood(Live) == 3 => {assign_type(center, Live);}

                  default => { }
```

|

```
def void setup(){
	int x = 2;
	int x = 3;

	grid_size(200, 200);
	set_chronon(200);
	//set_actor(50, 50, Dead);
	//set_grid_pattern(0, Dead, Live);
	set_grid_random();
}
```

Has a specific semantic error where it declares int x twice. Other than that the file is semantically correct. Thus the predicted error would be:
'Fatal error: exception Failure("Duplicate variable names!")' ,
which corresponded to the error message in our AST.

6.2 Testing of CAL files

This was testing for after the compile phase. We compared this to expected C code output.

Tests of Cellular Automata were done primarily by visual inspection. We compared the wiki Gifs to our own graphical outputs. This confirmed that we correctly coded Brians Brain:
http://en.wikipedia.org/wiki/File:Brian%27s_brain.gif

7 LESSONS LEARNED

7.1 Calvin Hu

Communicate well with your group members, miscommunication can lead to redundancy and unnecessary work.

OCaml is actually pretty nice.

Have someone else test your code after you, you'll find new ways to break things.

7.2 Nate Keane

Start small. Our whole project was less than 2000 lines of code which is very small considering the amount of time we put into it. By the end of the project I realized that the most effective way to code was to do it one line at a time and compile everything. This greatly diminished debugging time.

Segregate work appropriately. By the end of the project we each were specialists in a separate piece of code and knew how to interface in an efficient way. This was a large improvement from having to each learn what every line of code did exactly.

7.3 Eugene Kim

Putting much time and thought into the language proposal and language reference manual paid off for our group. Because we did that, we did not end up deviating much from our original syntax and structure, which made group work easy. We all followed the mutually agreed language features and syntax when we were implementing our parts.

I learned that working together in person makes collaboration so much more easy and efficient. It made easy to check in on each others' progress, ask questions on issues we were not familiar with, and get help on debugging.

Finally, it was important to keep a positive attitude and believe that we could get our project done even when there seemed to be many obstacles ahead of us. Congratulating each other and believing in each other also helped the process and team spirit.

On a more substantive front, I believe I learned alot from this course, including functional programming, lexical analysis, parsing, language translating, language design, github and group coding. Looking back, I am glad that we put so much time and effort into this class. We were all so proud when our code actually compiled and produced beautiful graphical output.

8 APPENDIX: SOURCE CODES

```
(* Scanner *)
(* Scanner for CAL *)

{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf }      (* Whitespace *)
| "//"         { comment lexbuf }          (* Comments *)

(* Punctuations *)
| '('          { LPAREN }
| ')'          { RPAREN }
| '{'          { LBRACE }
| '}'          { RBRACE }
```

```
| '['        { LBRACK }
| ']'        { RBRACK }
| '|'        { BAR }
| ';'        { SEMI }
| ':'        { COLON }
| ','        { COMMA }

(* Arithmetic Operators *)
| '+'        { PLUS }
| '-'        { MINUS }
| '*'        { TIMES }
| '/'        { DIVIDE }
| '%'        { MOD }
| '='        { ASSIGN }

(* Relational Operators *)
| "=="       { EQ }
| "!="       { NEQ }
| "&&"       { AND }
| "||"       { OR }
| '<'        { LT }
| "<="       { LEQ }
| ">"        { GT }
| ">="       { GEQ }
| "=>"       { ARROW }
| '!'        { NOT }

(* Key Words *)
| "if"       { IF }
| "else"     { ELSE }
| "true"     { TRUE }
| "false"    { FALSE }
| "while"    { WHILE }
| "bool"     { BOOL }
| "char"     { CHAR }
| "int"      { INT }
| "void"     { VOID }
| "grid"     { GRID }
| "direction"  { DIRECTION }
| "north"    { NORTH }
| "south"    { SOUTH }
| "east"     { EAST }
| "west"     { WEST }
```

```
| "northwest"   { NORTHWEST }
| "southwest"   { SOUTHWEST }
| "northeast"   { NORTHEAST }
| "southeast"   { SOUTHEAST }
| "center"      { CENTER }
| "rules"       { RULES }
| "return"      { RETURN }
| "def"         { DEF }
| "init"        { INIT }
| "default"     { DEFAULT }
| "actor_type"  { ACTOR_TYPE }

(* Literals *)
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| '\"'[^'\"']*'\"' as lxm { STR_LITERAL(lxm) }
| ['a'-'z' 'A'-'Z' '_']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof { EOF }

(* Special Character Process*)
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
  '\n'      { token lexbuf }
| _         { comment lexbuf }

(* parsey.mly *)

%{
open Ast
let parse_error s =
  print_endline s;
  flush stdout

%}

%token LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK BAR SEMI COLON
COMMA
%token PLUS MINUS TIMES DIVIDE MOD ASSIGN
%token EQ NEQ AND OR LT LEQ GT GEQ ARROW NOT
%token IF ELSE TRUE FALSE WHILE
%token BOOL CHAR INT GRID VOID
%token DIRECTION NORTH SOUTH EAST WEST NORTHWEST SOUTHWEST
NORTHEAST SOUTHEAST CENTER
```

```
%token RULES RETURN DEF INIT DEFAULT ACTOR_TYPE

%token <int> LITERAL
%token <string> STR_LITERAL
%token <string> ID

%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%left MOD


%start program
%type <Ast.program> program


%%


program:
        /* nothing */ { [] }
  | program fdecl { ($2 :: $1) }
  | program actor_type { ($2 :: $1) }

fdecl:
     DEF datatype id LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
          {
               CFunc({
                              dtype = $2;
                              fname = $3;
                              formals = $5;
                              locals = List.rev $8;
                              body= List.rev $9
                    })
```

```
                    }


formals_opt:
     /* nothing */ { [] }
    | formal_list   { List.rev $1 }

formal_list:
     datatype id                  { [FParam($1, $2)] }
    | formal_list COMMA datatype id { FParam($3, $4) :: $1 }


id:
      ID     { $1 }

grid:
    LBRACK grid_matrix RBRACK { $2 }

grid_row:
    ID                    { [$1] }
    | grid_row COMMA ID { $3 :: $1 }

grid_matrix:
    grid_row SEMI          { [$1] }
    | grid_matrix grid_row SEMI { $2 :: $1 }


datatype:
     BOOL          { BoolType }
     | CHAR        { CharType }
     | INT         { IntType }
     | VOID        { VoidType }
     | GRID        { GridType }
     | DIRECTION   { DirectionType }
     | ACTOR_TYPE  { Actor_TypeType }


rule:
      expr ARROW LBRACE stmt_list RBRACE { Rule($1, $4) }

rule_list:
       /* nothin */ { [] }
       | rule_list rule { $2 :: $1 }
```

```
default_rule:
        DEFAULT ARROW LBRACE stmt_list RBRACE  { $4 }

vdecl_list:
        /* nothing */    { [] }
       | vdecl_list vdecl { $2 :: $1 }

vdecl:
       datatype id ASSIGN expr SEMI { VDecl($1,$2,string_of_expr $4) }


actor_type:
       ACTOR_TYPE id ASSIGN BAR INIT COLON vdecl_list RULES COLON rule_list
default_rule BAR
                    {
                          ActorType({
                                         aname = $2;
                                          alocals = $7;
                                          arules = $10;
                                          adefault = $11;})

                    }



stmt_list:
        /* No empty block allowed */ { [] }
       | stmt_list stmt             { $2 :: $1 }




stmt:
     expr SEMI                                  { Expr($1) }
     | RETURN expr SEMI                         { Return($2) }
     | LBRACE stmt_list RBRACE                  {Block(List.rev $2) }
     | IF LPAREN expr RPAREN stmt %prec NOELSE        { If($3, $5, Block([])) }
     | IF LPAREN expr RPAREN stmt ELSE stmt           { If($3, $5, $7) }
     | WHILE LPAREN expr RPAREN stmt                  { While($3, $5 ) }

expr:
          LITERAL                       { Literal($1) }
       | STR_LITERAL                    { String($1) }
```

```
    | id                                  { Id($1) }
    | grid                    { Grid($1) }
    | expr PLUS expr                      { Binop($1, Add, $3 ) }
    | expr MINUS  expr                    { Binop($1, Sub,   $3) }
  | expr TIMES  expr                      { Binop($1, Mult,  $3) }
  | expr MOD expr          { Binop($1, Div,   $3) }
    /*Directions*/
    | NORTH                        { Direction(North)     }
  | SOUTH                        { Direction(South)     }
  | WEST                        { Direction(West)      }
  | EAST                        { Direction(East)      }
  | NORTHEAST                        { Direction(NorthEast) }
  | NORTHWEST                        { Direction(NorthWest) }
  | SOUTHEAST                        { Direction(SouthEast) }
  | SOUTHWEST                        { Direction(SouthWest) }
  | CENTER                        { Direction(Center) }
    /* bools */
  | TRUE                        { BVal(True) }
  | FALSE                        { BVal(False) }
    | expr EQ expr                  { EExpr($1, BEqual, $3) }
  | expr NEQ expr                 { EExpr($1, BNeq, $3) }
  | expr GT expr                 { RExpr($1, BGreater, $3) }
  | expr GEQ expr                 { RExpr($1, BGeq, $3) }
  | expr LT expr                 { RExpr($1, BLess, $3) }
  | expr LEQ expr                 { RExpr($1, BLeq, $3) }
  | expr AND expr                 { BExpr($1, And, $3) }
  | expr OR expr                 { BExpr($1, Or, $3) }
    | id ASSIGN expr                 { Assign($1, $3) }
  | id LPAREN actuals_opt RPAREN  { Call($1, $3) }
  | LPAREN expr RPAREN           { Bracket($2) }


actuals_opt:
        /* nothin */ { [] }
        | actual_list {List.rev $1}

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


(* ast.ml *)
(* AST *)
```

```
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq | Mod
type direc = North | South | East | West | NorthEast | NorthWest | SouthEast | SouthWest | Center

type bv = True | False

type bop = And | Or
type eop = BEqual | BNeq
type rop = BLess | BLeq | BGreater | BGeq

type mop = MTimes | MDivide | MMod (*multiplicative expr ops*)
type aop = AAdd | ASub (*additve expr ops*)

type vop = VAdd | VSub | VMult | VDiv

type dt = BoolType | CharType | IntType | VoidType | GridType | DirectionType | Actor_TypeType

type fparam = FParam of dt * string

type vdecl = VDecl of dt * string * string

type expr =
         Literal of int
        | Boolean of bool
        | String of string
        | Id of string
        | Grid of string list list
        | Direction of direc
        | Bracket of expr
        | Binop of expr * op * expr
        | Assign of string * expr
        | Call of string * expr list
        | Noexpr
        | BVal of bv
        | RExpr of expr * rop * expr
        | EExpr of expr * eop * expr
        | BExpr of expr * bop * expr

type stmt =
        Expr of expr
        | Return of expr
        | Block of stmt list
        | If of expr * stmt * stmt
```

```
        | While of expr * stmt


type rule = Rule of expr * stmt list

type arithexpr =
        | ALiteral of int
        | AId of string

type varexpr =
        | VLiteral of int
        | VId of string
        | VStringLit of string
        | VBoolLit of bool
        | VBinop of varexpr *  vop * varexpr

type func_decl = {
        dtype : dt;
        fname : string;
        formals : fparam list;
        locals : vdecl list;
        body : stmt list;
}

type actor_type = {
          aname : string;
          alocals : vdecl list;
          arules : rule list;
          adefault: stmt list;

}

type func =
        | CFunc of func_decl
        | ActorType of actor_type


type program = func list


let string_of_var_dec (a,b,c) = a ^ b ^ c

let string_of_vop  = function
```

```
        | VAdd -> "+"
        | VSub -> "-"
        | VMult-> "*"
        | VDiv -> "/"

let string_of_arithexpr  = function
        | ALiteral(i) -> string_of_int i
        | AId(s) -> s


let rec string_of_varexpr = function
        | VLiteral(i) -> string_of_int i
        | VId(s) -> s
        | VStringLit(s) -> s
        | VBoolLit(b) -> string_of_bool b
        | VBinop(v1,op,v2) -> string_of_varexpr v1 ^ " " ^ string_of_vop op ^ " " ^ string_of_varexpr
v2


let string_of_dt = function
        BoolType -> "bool"
        | CharType -> "char"
        | IntType -> "int"
        | VoidType -> "void"
        | GridType -> "grid"
        | DirectionType -> "direction"
        | Actor_TypeType -> "actor_type"


let string_of_bop = function
        | And -> "&&"
        | Or -> "||"

let string_of_rop = function
        | BLess -> "<"
        | BLeq -> "<="
        | BGreater -> ">"
        | BGeq -> ">="

let string_of_eop = function
        | BEqual -> "=="
        | BNeq -> "!="
```

```
let string_of_bv = function
        | True -> "true"
        | False -> "false"

let string_of_op = function
          Add -> "+"
        | Sub -> "-"
        | Mult -> "*"
        | Div -> "/"
        | Equal -> "=="
        | Neq -> "!="
        | Less -> "<"
        | Leq -> "<="
        | Greater -> ">"
        | Geq -> ">="
        | Mod -> "%"


let rec string_of_listlist = function
        [] -> ""
        | a::b -> (String.concat "," a) ^ ";" ^ ( string_of_listlist b)


let string_of_grid = function
        g -> "[" ^ (string_of_listlist g) ^ "]"

let rec string_of_expr = function
          Literal(l) -> string_of_int l
        | Boolean(b) -> string_of_bool b
        | String(s) -> s
        | Id(s) -> s
        | Grid(g) -> string_of_grid g
        | Direction(d) ->
                    begin
                            match d with
                                North -> "NORTH" | South -> "SOUTH" | East -> "EAST" | West ->
"WEST"  | Center -> "CENTER"
                              | NorthEast -> "NORTHEAST" | SouthEast -> "SOUTHEAST" | NorthWest
-> "NORTHWEST" | SouthWest -> "SOUTHWEST"
                            end
            | Binop(e1, o, e2) ->
                    begin
                            match o with
```

```
                        | _ ->
                        string_of_expr e1 ^ " " ^ (match o with
                                  Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
                                | Equal -> "==" | Neq -> "!="
                                | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">=" | Mod ->
"%")
                            ^ " " ^ string_of_expr e2
                end
      | Assign(v, e) -> v ^ " = " ^ string_of_expr e
      | Call(f, el) -> (match Str.string_match (Str.regexp "move") f  0 with
          true -> f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ", read_grid, write_grid, i, j)"
         |_ -> match Str.string_match (Str.regexp "assign_type") f  0 with
             true -> f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ", write_grid, i, j)"
            |_ -> match Str.string_match (Str.regexp "neighborhood") f  0 with
                true -> f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ", read_grid, i, j)"
               |_ -> match Str.string_match (Str.regexp "randomof") f  0 with
                   true -> f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ", read_grid , i, j)"
                  |_ -> match Str.string_match (Str.regexp "cellat") f  0 with
                      true -> f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ", read_grid , i, j)"
                     |_ -> f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")")
      | Noexpr -> ""
      | BVal(v) -> string_of_bv v
      | RExpr(e1,o,e2) -> string_of_expr e1 ^ " " ^ string_of_rop o ^ " " ^ string_of_expr e2
      | EExpr(e1,o,e2) -> string_of_expr e1 ^ " " ^ string_of_eop o ^ " " ^ string_of_expr e2
      | BExpr(e1,o,e2) -> string_of_expr e1 ^ " " ^ string_of_bop o ^ " " ^ string_of_expr e2
      | Bracket(e1) -> " ( " ^ string_of_expr e1 ^ " ) "

let rec string_of_stmt = function
        Block(stmts) -> "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
      | Expr(expr) -> string_of_expr expr ^ ";\n";
      | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
      | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
      | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s1 ^ "else\n" ^ string_of_stmt
s2
      | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s

let rec string_of_expr_at = function
        Literal(l) -> string_of_int l
      | Boolean(b) -> string_of_bool b
      | String(s) -> s
      | Id(s) -> s
      | Grid(g) -> string_of_grid g
      | Direction(d) ->
```

```
                begin
                        match d with
                            North -> "NORTH" | South -> "SOUTH" | East -> "EAST" | West ->
"WEST" | Center -> "CENTER"
                          | NorthEast -> "NORTHEAST" | SouthEast -> "SOUTHEAST" | NorthWest
-> "NORTHWEST" | SouthWest -> "SOUTHWEST"
                end
      | Binop(e1, o, e2) ->
                begin
                        match o with
                          | _ ->
                                        string_of_expr_at e1 ^ " " ^ (match o with
                                                    Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
                                                  | Equal -> "==" | Neq -> "!="
                                                  | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">=" |
Mod -> "%")
                                            ^ " " ^ string_of_expr_at e2
                  end
      | Assign(v, e) -> "!addwritegrid!" ^ v ^ " = " ^ string_of_expr_at e
      | Call(f, el) -> (match Str.string_match (Str.regexp "move") f  0 with
          true -> f ^ "(" ^ String.concat ", " (List.map string_of_expr_at el) ^ ", read_grid, write_grid, i,
j, width, height)"
         |_ -> match Str.string_match (Str.regexp "assign_type") f  0 with
             true -> f ^ "(" ^ String.concat ", " (List.map string_of_expr_at el) ^ ", write_grid, i, j,
width, height)"
            |_ -> match Str.string_match (Str.regexp "neighborhood") f  0 with
                true -> f ^ "(" ^ String.concat ", " (List.map string_of_expr_at el) ^ ", read_grid, i, j,
width, height)"
               |_ -> match Str.string_match (Str.regexp "randomof") f  0 with
                   true -> f ^ "(" ^ String.concat ", " (List.map string_of_expr_at el) ^ ", read_grid , i, j,
width, height)"
                  |_ -> match Str.string_match (Str.regexp "cellat") f  0 with
                      true -> f ^ "(" ^ String.concat ", " (List.map string_of_expr_at el) ^ ", read_grid , i,
j, width, height)"
                     |_ -> f ^ "(" ^ String.concat ", " (List.map string_of_expr_at el) ^ ")")
      | Noexpr -> ""
      | BVal(v) -> string_of_bv v
      | RExpr(e1,o,e2) -> string_of_expr_at e1 ^ " " ^ string_of_rop o ^ " " ^ string_of_expr_at e2
      | EExpr(e1,o,e2) -> string_of_expr_at e1 ^ " " ^ string_of_eop o ^ " " ^ string_of_expr_at e2
      | BExpr(e1,o,e2) -> string_of_expr_at e1 ^ " " ^ string_of_bop o ^ " " ^ string_of_expr_at e2
      | Bracket(e1) -> " ( " ^ string_of_expr_at e1 ^ " ) "
```

```
let rec string_of_stmt_at = function
          Block(stmts) -> "{\n" ^ String.concat "" (List.map string_of_stmt_at stmts) ^ "}\n"
        | Expr(expr) -> string_of_expr_at expr ^ ";\n";
        | Return(expr) -> "return " ^ string_of_expr_at expr ^ ";\n";
        | If(e, s, Block([])) -> "if (" ^ string_of_expr_at e ^ ")\n" ^ string_of_stmt_at s
        | If(e, s1, s2) ->  "if (" ^ string_of_expr_at e ^ ")\n" ^ string_of_stmt_at s1 ^ "else\n" ^
string_of_stmt_at s2
        | While(e, s) -> "while (" ^ string_of_expr_at e ^ ") " ^ string_of_stmt_at s

let rec string_of_expr_setup = function
          Literal(l) -> string_of_int l
        | Boolean(b) -> string_of_bool b
        | String(s) -> s
        | Id(s) -> s
        | Grid(g) -> string_of_grid g
        | Direction(d) ->
                    begin
                        match d with
                           North -> "NORTH" | South -> "SOUTH" | East -> "EAST" | West ->
"WEST" | Center -> "CENTER"
                          | NorthEast -> "NORTHEAST" | SouthEast -> "SOUTHEAST" | NorthWest
-> "NORTHWEST" | SouthWest -> "SOUTHWEST"
                    end
        | Binop(e1, o, e2) ->
                     begin
                        match o with
                          | _ ->
                                              string_of_expr_setup e1 ^ " " ^ (match o with
                                                      Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
                                                    | Equal -> "==" | Neq -> "!="
                                                    | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">=" |
Mod -> "%")
                                              ^ " " ^ string_of_expr_setup e2
                     end
        | Assign(v, e) -> "!addwritegrid!" ^ v ^ " = " ^ string_of_expr_setup e
        | Call(f, el) -> (match Str.string_match (Str.regexp "move") f  0 with
            true -> f ^ "(" ^ String.concat ", " (List.map string_of_expr_setup el) ^ ", read_grid,
write_grid, i, j, width, height)"
           |_ -> match Str.string_match (Str.regexp "assign_type") f  0 with
               true -> f ^ "(" ^ String.concat ", " (List.map string_of_expr_setup el) ^ ", write_grid, i, j,
width, height)"
               |_ -> match Str.string_match (Str.regexp "neighborhood") f  0 with
```

```
                    true -> f ^ "(" ^ String.concat ", " (List.map string_of_expr_setup el) ^ ", read_grid, i, j,
width, height)"
                    |_ -> match Str.string_match (Str.regexp "randomof") f  0 with
                        true -> f ^ "(" ^ String.concat ", " (List.map string_of_expr_setup el) ^ ", read_grid , i,
j, width, height)"
                        |_ -> match Str.string_match (Str.regexp "cellat") f  0 with
                            true -> f ^ "(" ^ String.concat ", " (List.map string_of_expr_setup el) ^ ", read_grid
, i, j, width, height)"
                            |_ -> match Str.string_match (Str.regexp "grid_size") f  0 with
                                true -> f ^ "(" ^ String.concat ", " (List.map string_of_expr_setup el) ^ ",
&grid1, &grid2, &MAX_X, &MAX_Y)"
                                |_ -> match Str.string_match (Str.regexp "set_actor") f  0 with
                                    true -> f ^ "(" ^ String.concat ", " (List.map string_of_expr_setup el) ^ ",
grid1 , grid2)"
                                    |_ -> match Str.string_match (Str.regexp "cellat") f  0 with
                                        true -> f ^ "(" ^ String.concat ", " (List.map string_of_expr_setup el) ^ ",
read_grid , i, j)"
                                        |_ -> match Str.string_match (Str.regexp "set_chronon") f  0 with
                                            true -> f ^ "(" ^ String.concat ", " (List.map string_of_expr_setup el) ^ ",
&STEP_TIME)"
                                            |_ -> match Str.string_match (Str.regexp "set_grid_pattern") f  0 with
                                                true -> f ^ "(" ^ String.concat ", " (List.map string_of_expr_setup el) ^
", grid1, grid2)"
                                                |_ -> match Str.string_match (Str.regexp "set_cell_size") f  0 with
                                                    true -> f ^ "(" ^ String.concat ", " (List.map string_of_expr_setup
el) ^ ", &CELL_SIZE)"
                                                    |_ -> match Str.string_match (Str.regexp "set_grid_random") f  0
with
                                                        true -> f ^ "(" ^ String.concat ", " (List.map
string_of_expr_setup el) ^ "grid1, grid2, MAX_X, MAX_Y)"
                                                        |_ -> match Str.string_match (Str.regexp "read_grid") f  0 with
                                                            true -> f ^ "(" ^ String.concat ", " (List.map
string_of_expr_setup el) ^ ", grid1, grid2, MAX_X, MAX_Y)"
                                                            |_ -> f ^ "(" ^ String.concat ", " (List.map
string_of_expr_setup el) ^ ")")
        | Noexpr -> ""
        | BVal(v) -> string_of_bv v
        | RExpr(e1,o,e2) -> string_of_expr_at e1 ^ " " ^ string_of_rop o ^ " " ^ string_of_expr_at e2
        | EExpr(e1,o,e2) -> string_of_expr_at e1 ^ " " ^ string_of_eop o ^ " " ^ string_of_expr_at e2
        | BExpr(e1,o,e2) -> string_of_expr_at e1 ^ " " ^ string_of_bop o ^ " " ^ string_of_expr_at e2
        | Bracket(e1) -> " ( " ^ string_of_expr_at e1 ^ " ) "
```

```
let rec string_of_stmt_setup = function
     Block(stmts) -> "{\n" ^ String.concat "" (List.map string_of_stmt_setup stmts) ^ "}\n"
    | Expr(expr) -> string_of_expr_setup expr ^ ";\n";
    | Return(expr) -> "return " ^ string_of_expr_setup expr ^ ";\n";
    | If(e, s, Block([])) -> "if (" ^ string_of_expr_setup e ^ ")\n" ^ string_of_stmt_setup s
    | If(e, s1, s2) ->  "if (" ^ string_of_expr_setup e ^ ")\n" ^ string_of_stmt_setup s1 ^ "else\n" ^
string_of_stmt_setup s2
    | While(e, s) -> "while (" ^ string_of_expr_setup e ^ ") " ^ string_of_stmt_setup s


let string_of_vdecl = function
    VDecl(dtt, nm, v) ->  string_of_dt dtt ^ " " ^ nm ^ " = " ^ v ^ ";\n"


let string_of_fparam = function
    FParam(dt,s) -> string_of_dt dt ^ " " ^ s


let string_of_fdecl  = function
    | CFunc(fdecl) -> if (String.compare fdecl.fname "setup") !=0 then
            "\n" ^ string_of_dt fdecl.dtype ^ " " ^ fdecl.fname ^ "(" ^ String.concat ", " (List.map
string_of_fparam fdecl.formals) ^ ") {\n" ^
            String.concat "" (List.map string_of_vdecl fdecl.locals) ^
            String.concat "" (List.map string_of_stmt fdecl.body) ^
            "}\n"
          else ""
    |_ -> ""


let string_of_program (funcs) = String.concat "\n" (List.map string_of_fdecl funcs)


(* compiler.ml *)
open Ast
open Str
open Cal_std

module StringMap = Map.Make(String);;

let rec concat_list l =
  match l with
    [] -> ""
```

```
  | hd :: tl -> hd ^ (concat_list tl)

let add_actortype_initdec fdecl =
	let vdecls = fdecl.alocals in
	let aname = fdecl.aname in
	let string_of_vinit vd = match vd with
		VDecl(dt, name, _) -> "    " ^ Ast.string_of_dt dt ^ " " ^ name ^ ";\n" in
			"struct " ^ aname ^ "_actortype{\n" ^  concat_list (List.map string_of_vinit
vdecls) ^ "     } " ^ aname ^ "_actor;\n"


let add_colors = function ActorType(fdecl) ->
		(let aname = fdecl.aname in
		let red = Random.int 256 in
		let green = Random.int 256 in
		let blue = Random.int 256 in
		"						case " ^ String.uppercase aname ^
"_ACTORTYPE:
						color = SDL_MapRGB(screen->format, "^
string_of_int red ^","^ string_of_int green ^"," ^ string_of_int blue ^");
						break;\n")
		| _ -> ""

let rec replace_actortypes block at_list = match at_list with
	[] -> block
	| hd :: tl ->
		(match hd with ActorType(hd) -> replace_actortypes (global_replace (Str.regexp
hd.aname) ((String.uppercase hd.aname) ^ "_ACTORTYPE") block) tl | _ -> block)

let add_actortype_initinit fdecl func_list =
	let vdecls = fdecl.alocals in
	let aname = fdecl.aname in
	let string_of_initinit vd = match vd with
		VDecl(_, name, value) -> "				(*actor).actors." ^ aname ^ "_actor." ^
name ^ " = " ^ (replace_actortypes value func_list) ^ ";\n" in
			"			case " ^ String.uppercase aname ^ "_ACTORTYPE:\n" ^
concat_list (List.map string_of_initinit vdecls) ^ "				break;\n"

let add_actortype_deftype fdecl type_number =
	let aname = fdecl.aname in
	"#define " ^ String.uppercase aname ^ "_ACTORTYPE " ^ string_of_int type_number ^ "\n"

let add_actortype_updateblock fdecl func_list =
```

```
	let aname = fdecl.aname in
	let rules = fdecl.arules in
	let default = fdecl.adefault in
	let locals = fdecl.alocals in
	let concat_tabs s = "                                              " ^ s in
	let concat_tabs2 s = "                                     " ^ s in
	let string_of_rule rl = match rl with
		Rule(condition, statement_list) -> "if(" ^ string_of_expr_at condition ^ "){\n" ^
concat_list (List.map concat_tabs (List.map Ast.string_of_stmt_at statement_list)) ^ "
		}\n                                          else " in
	let string_of_default df rule_count = match rule_count with
		0 ->  concat_list (List.map  concat_tabs2 (List.map Ast.string_of_stmt_at df)) ^ "\n"
		|_  -> "{\n" ^ concat_list (List.map  concat_tabs (List.map Ast.string_of_stmt_at df)) ^
"                                            }\n" in
	let add_prefix_to_var s_block var = match var with
		VDecl(_, name, _) -> global_replace (Str.regexp "!addwritegrid!read_grid")
"write_grid" (global_replace (Str.regexp name) ("read_grid[i][j].actors."^aname^"_actor."^name)
s_block) in
	let rec add_prefix_to_vars stmt_block vars = match vars with
		[] -> stmt_block
		| hd :: tl -> add_prefix_to_vars (add_prefix_to_var stmt_block hd) tl in
	add_prefix_to_vars (replace_actortypes("                                       case " ^
String.uppercase aname ^ "_ACTORTYPE:\n                                       " ^ concat_list
(List.map string_of_rule rules) ^ string_of_default default (List.length rules) ^ "
		break;\n") func_list) locals


let get_actortype_initdecs = function ActorType(fdecl) -> add_actortype_initdec fdecl | _ -> ""

let rec get_actortype_initinits functions functions_for_count =
	let get_actortype_initinits_helper func func_list = match func with
		ActorType(fdecl) -> (add_actortype_initinit fdecl func_list)| _ -> "" in
	match functions_for_count with
		[] -> ""
		| hd :: tl -> ((get_actortype_initinits_helper hd functions) ^ (get_actortype_initinits
functions tl))

let rec get_actortype_count funcs count = match funcs with
	[] -> count
	| hd :: tl -> (match hd with
		ActorType(hd) -> get_actortype_count tl (count + 1)
		|_ -> get_actortype_count tl count)
```

```
let get_actortype_deftypes fdecl type_number = match fdecl with ActorType(fdecl) ->
add_actortype_deftype fdecl type_number | _ -> ""

let rec get_actortype_deftypes_rec functions type_number =
	match functions with
	  [] -> ""
	| hd :: tl -> get_actortype_deftypes hd type_number ^ get_actortype_deftypes_rec tl
(type_number + 1)

let rec get_actortype_updateblocks functions functions_for_count=
	let get_actortype_updateblocks_helper func func_list = match func with
		ActorType(func) -> (add_actortype_updateblock func func_list) | _ -> ""  in
	match functions_for_count with
		[] -> ""
		| hd :: tl ->((get_actortype_updateblocks_helper hd functions) ^
(get_actortype_updateblocks functions tl))

let rec get_setup_function functions func_count = match func_count with
	[] -> ""
	| hd :: tl -> (match hd with
		CFunc(hd) -> if (String.compare (string_of_dt hd.dtype) "void") == 0 &&
(List.length hd.formals) <= 0 && (String.compare hd.fname "setup") == 0 then replace_actortypes
(concat_list (List.map Ast.string_of_stmt_setup hd.body)) functions else get_setup_function
functions tl
		|_ -> get_setup_function functions tl)

let translate functions program_name =
	let ochannel = ignore(Random.self_init ()); open_out (program_name ^ ".c") in
		let sorted_functions = List.rev (List.sort compare functions) in
		let at_count = get_actortype_count functions 0 in
		let actor_initdecs = concat_list (List.map get_actortype_initdecs sorted_functions) in
		let actor_initinits = get_actortype_initinits sorted_functions sorted_functions in
		let actor_typedefs = get_actortype_deftypes_rec sorted_functions 0 in
		let actor_updateblocks =  get_actortype_updateblocks sorted_functions
sorted_functions in
		let actor_colors = concat_list (List.map add_colors sorted_functions) in
		let normal_funcs = concat_list (List.map string_of_fdecl sorted_functions) in
		let setup_func = get_setup_function sorted_functions sorted_functions in
		let program_string =
			global_replace (Str.regexp("PROGRAM_NAME"))
("\""^program_name^"\"")
			(Cal_std.std_template0 ^ string_of_int at_count ^ Cal_std.std_template1 ^
actor_typedefs ^ Cal_std.std_template2 ^
```

```
                              actor_initdecs ^ Cal_std.std_template3 ^ actor_initinits ^
                              Cal_std.std_template4 ^ actor_updateblocks ^ Cal_std.std_template5 ^
                              normal_funcs ^ Cal_std.std_template6 ^ setup_func ^ Cal_std.std_template7 ^
actor_colors^
                              Cal_std.std_template8)

                  in
                              let exit_code = ignore(Printf.fprintf ochannel "%s" program_string);
                                    close_out ochannel;
                                    Sys.command (Printf.sprintf "gcc -o %s.out %s.c -lmingw32
-lSDLmain -lSDL
" program_name program_name) in
                                          match exit_code with
                                                0 -> "\nCompilation was a success.\n"
                                                |_ -> "\nCompilation failed.\n"

(* cal_std.ml *)
let std_template0 = "#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include \"SDL/SDL.h\"

#define SOUTHWEST 0
#define WEST 1
#define NORTHWEST 2
#define SOUTH 3
#define CENTER 4
#define NORTH 5
#define SOUTHEAST 6
#define EAST 7
#define NORTHEAST 8

#define TOTAL_TYPES "

let std_template1 = "

typedef int direction;
typedef char actor_type;

"

let std_template2 = "
```

```
typedef struct actor{
			union actor_types{"

let std_template3 = "
;
			} actors;
			char type;
		} ACTOR;

void init(char type, ACTOR* actor){
		(*actor).type = type;
		switch(type){"

let std_template4 = "	}
}



void set_grid_random(ACTOR** grid1, ACTOR** grid2, int width, int height){
		int i,j;
		for(i = 0; i < width; i++){
			for(j = 0; j < height; j++){
				init(rand() % TOTAL_TYPES, &grid1[i][j]);
				grid2[i][j] = grid1[i][j];
			}
		}
}

void grid_size(int w, int h, ACTOR*** grid1, ACTOR*** grid2, int *width, int *height){
		int i;
		for(i = 0; i < *width; i++){
			free((*grid1)[i]);
			free((*grid2)[i]);
		}
		free(*grid1);
		free(*grid2);
		*width = w;
		*height = h;
		*grid1 = malloc(w * sizeof(ACTOR*));
		*grid2 = malloc(w * sizeof(ACTOR*));
		for(i = 0; i < w; i++){
			(*grid1)[i] = malloc(h * sizeof(ACTOR));
			(*grid2)[i] = malloc(h * sizeof(ACTOR));
```

```
	}
	set_grid_random(*grid1, *grid2, *width, *height);
}

void set_actor(int x, int y, char type, ACTOR** grid1, ACTOR** grid2){
	init(type, &grid1[x][y]);
	init(type, &grid2[x][y]);
}

void set_chronon(int millis, int *time){
	*time = millis;
}

void set_cell_size(int size, int *cellsize){
	*cellsize = size;
}

void set_grid_pattern(int pattern_type, char atype_1, char atype_2, int w_s, int h_s, int startx, int
starty, ACTOR** grid1, ACTOR** grid2){
	int i,j;
	int width = startx + w_s;
	int height = starty + h_s;
	switch(pattern_type){
		//CHECKERBOARD
		case 0:
			printf(\"inloop\\n\");
			for(i = startx; i < width; i+=2){
				for(j = starty; j < height; j+=2){
					init(atype_1, &grid1[i][j]);
					grid2[i][j] = grid1[i][j];
				}
			}
			for(i = startx; i < width; i+=2){
				for(j = starty + 1; j < height; j+=2){
					init(atype_2, &grid1[i][j]);
					grid2[i][j] = grid1[i][j];
				}
			}
			for(i = startx + 1; i < width; i+=2){
				for(j = starty + 1; j < height; j+=2){
					init(atype_1, &grid1[i][j]);
					grid2[i][j] = grid1[i][j];
				}
```

```
		}
		for(i = startx + 1; i < width; i+=2){
			for(j = starty; j < height; j+=2){
				init(atype_2, &grid1[i][j]);
				grid2[i][j] = grid1[i][j];
			}
		}
		break;
	//ROWS
	case 1:
		for(i = startx; i < width; i++){
			for(j = starty; j < height; j+=2){
				init(atype_1, &grid1[i][j]);
				grid2[i][j] = grid1[i][j];
			}
		}
		for(i = startx; i < width; i++){
			for(j = starty+1; j < height; j+=2){
				init(atype_2, &grid1[i][j]);
				grid2[i][j] = grid1[i][j];
			}
		}
		break;
	//COLUMNS
	case 2:
		for(i = startx; i < width; i+=2){
			for(j = starty; j < height; j++){
				init(atype_1, &grid1[i][j]);
				grid2[i][j] = grid1[i][j];
			}
		}
		for(i = starty+1; i < width; i+=2){
			for(j = starty; j < height; j++){
				init(atype_2, &grid1[i][j]);
				grid2[i][j] = grid1[i][j];
			}
		}
		break;
	//FILL
	case 3:
		for(i = startx; i < width; i++){
			for(j = starty; j < height; j++){
				init(atype_1, &grid1[i][j]);
```

```
                    grid2[i][j] = grid1[i][j];
                }
            }
            break;
    }
}

void read_grid(char* file_path, ACTOR** grid1, ACTOR** grid2, int width, int height){

}

char cellat(int direction, ACTOR** grid, int xpos, int ypos, int width, int height){
    int xstart, ystart;
    if(xpos <= 0){
        xstart = width - 1;
    }
    else{
        xstart = xpos - 1;
    }
    if(ypos <= 0){
        ystart = height - 1;
    }
    else{
        ystart = ypos - 1;
    }
    switch(direction){
        case SOUTHWEST:
            return grid[xstart%width][(ystart%height)].type;
        case WEST:
            return grid[xstart%width][((ystart+1)%height)].type;
        case NORTHWEST:
            return grid[xstart%width][((ystart+2)%height)].type;
        case SOUTH:
            return grid[(xstart+1)%width][(ystart%height)].type;
        case CENTER:
            return grid[(xstart+1)%width][((ystart+1)%height)].type;
        case NORTH:
            return grid[(xstart+1)%width][((ystart+2)%height)].type;
        case SOUTHEAST:
            return grid[(xstart+2)%width][(ystart%height)].type;
        case EAST:
            return grid[(xstart+2)%width][((ystart+1)%height)].type;
        case NORTHEAST:
```

```
				return grid[(xstart+2)%width][((ystart+2)%height)].type;
			default:
				return grid[(xstart+1)%width][((ystart+1)%height)].type;
		}
}

void assign_type(int direction, char type, ACTOR** grid, int xpos, int ypos, int width, int height){
		int xstart, ystart;
		if(xpos <= 0){
			xstart = width - 1;
		}
		else{
			xstart = xpos - 1;
		}
		if(ypos <= 0){
			ystart = height - 1;
		}
		else{
			ystart = ypos - 1;
		}
		switch(direction){
			case SOUTHWEST:
				init(type, &(grid[xstart%width][(ystart%height)]));
				break;
			case WEST:
				init(type, &(grid[xstart%width][((ystart+1)%height)]));
				break;
			case NORTHWEST:
				init(type, &(grid[xstart%width][((ystart+2)%height)]));
				break;
			case SOUTH:
				init(type, &(grid[(xstart+1)%width][(ystart%height)]));
				break;
			case CENTER:
				init(type, &(grid[(xstart+1)%width][((ystart+1)%height)]));
				break;
			case NORTH:
				init(type, &(grid[(xstart+1)%width][((ystart+2)%height)]));
				break;
			case SOUTHEAST:
				init(type, &(grid[(xstart+2)%width][(ystart%height)]));
				break;
			case EAST:
```

```
			init(type, &(grid[(xstart+2)%width][((ystart+1)%height)]));
			break;
		case NORTHEAST:
			init(type, &(grid[(xstart+2)%width][((ystart+2)%height)]));
			break;
		default:
			init(type, &(grid[(xstart+1)%width][((ystart+1)%height)]));
	}
}

void move(int direction, char replace_type, ACTOR** read_grid, ACTOR** write_grid, int xpos,
int ypos, int width, int height){

	char og_type = read_grid[xpos][ypos].type;


	//assign_type(direction, og_type, grid, xpos, ypos);

	int xstart, ystart;

	if(xpos <= 0){
		xstart = width - 1;
	}
	else{
		xstart = xpos - 1;
	}
	if(ypos <= 0){
		ystart = height - 1;
	}
	else{
		ystart = ypos - 1;
	}

	switch(direction){
		case SOUTHWEST:
			write_grid[xstart%width][(ystart%height)] = write_grid[xpos][ypos];
			break;
		case WEST:
			write_grid[xstart%width][((ystart+1)%height)] = write_grid[xpos][ypos];
			break;
		case NORTHWEST:
			write_grid[xstart%width][((ystart+2)%height)] = write_grid[xpos][ypos];
			break;
```

```
			case SOUTH:
				write_grid[(xstart+1)%width][(ystart%height)] = write_grid[xpos][ypos];
				break;
			case CENTER:
				write_grid[(xstart+1)%width][((ystart+1)%height)] = write_grid[xpos][ypos];
				break;
			case NORTH:
				write_grid[(xstart+1)%width][((ystart+2)%height)] = write_grid[xpos][ypos];
				break;
			case SOUTHEAST:
				write_grid[(xstart+2)%width][(ystart%height)] = write_grid[xpos][ypos];
				break;
			case EAST:
				write_grid[(xstart+2)%width][((ystart+1)%height)] = write_grid[xpos][ypos];
				break;
			case NORTHEAST:
				write_grid[(xstart+2)%width][((ystart+2)%height)] = write_grid[xpos][ypos];
				break;
			default:
				write_grid[(xstart+1)%width][((ystart+1)%height)] = write_grid[xpos][ypos];
		}
		init(replace_type, &(write_grid[xpos][ypos]));

}

int neighborhood(char type, ACTOR** grid, int xpos, int ypos, int width, int height){
		int neighbor_count = 0;

		int i,j;
		int xstart, ystart;
		int curr_x, curr_y;
		int current_direction = 0;

		if(xpos <= 0){
			xstart = width - 1;
		}
		else{
			xstart = xpos - 1;
		}
		if(ypos <= 0){
			ystart = height - 1;
		}
		else{
```

```
		ystart = ypos - 1;
	}

	for(i = 0; i < 3; i++){
		for(j = 0; j < 3; j++){
			curr_x = (xstart+i)%width;
			curr_y = (ystart+j)%height;
			if(current_direction == 4){
				continue;
			}
			if(grid[curr_x][curr_y].type == type){
				neighbor_count++;
			}
		}
	}

	return neighbor_count;
}

int randomof(char type, ACTOR** grid, int xpos, int ypos, int width, int height){
	int neighborhood[8];
	int neighbor_count = 0;
	int current_direction = 0;

	int i,j;
	int xstart, ystart;
	int curr_x, curr_y;
	if(xpos <= 0){
		xstart = width - 1;
	}
	else{
		xstart = xpos - 1;
	}
	if(ypos <= 0){
		ystart = height - 1;
	}
	else{
		ystart = ypos - 1;
	}

	for(i = 0; i < 3; i++){
		for(j = 0; j < 3; j++){
			curr_x = (xstart+i)%width;
```

```
                        curr_y = (ystart+j)%height;
                        if(current_direction == 4){
                                current_direction++;
                                continue;
                        }
                        if(grid[curr_x][curr_y].type == type){
                                neighborhood[neighbor_count] = current_direction;
                                neighbor_count++;
                        }
                        current_direction++;
                }
        }

        if(neighbor_count > 0){
                return neighborhood[rand() % neighbor_count];
        }
        else{
                return -1;
        }

}


void update(ACTOR** read_grid, ACTOR** write_grid, int width, int height){
        int i,j;
        char new_type;
        for(i = 0; i < width; i++){
                for(j = 0; j < height; j++){
                        write_grid[i][j] = read_grid[i][j];
                }
        }
        for(i = 0; i < width; i++){
                for(j = 0; j < height; j++){
                        switch(read_grid[i][j].type){"

let std_template5 = "
                        }
                }
        }
}

void colorblock(SDL_Rect rect, SDL_Rect offset, SDL_Surface* screen, SDL_Surface* surface, int
xpos, int ypos, int width, int height, Uint32 color){
```

```
    rect.x = 0;
    rect.y = 0;
    rect.w = width;
    rect.h = height;

    offset.x = xpos;
    offset.y = ypos;
    offset.w = width;
    offset.h = height;

    SDL_FillRect(surface, &rect, color);
    SDL_BlitSurface(surface, NULL, screen, &offset);
}"

let std_template6 = "

int main(int argc, char* args[]){
        srand(time(NULL));

        int MAX_X = 100;
        int MAX_Y = 100;
        int CELL_SIZE = 4;
        int STEP_TIME = 50;
        int i,j,k;

        ACTOR** grid1 = malloc(sizeof(ACTOR*) * MAX_X);
        ACTOR** grid2 = malloc(sizeof(ACTOR*) * MAX_X);


        for(i = 0; i < MAX_X; i++){
                grid1[i] = malloc(sizeof(ACTOR) * MAX_Y);
                grid2[i] = malloc(sizeof(ACTOR) * MAX_Y);
        }
        set_grid_random(grid1,grid2,MAX_X,MAX_Y);
"
let std_template7 = "
        printf(\"%d,%d\\n\", MAX_X, MAX_Y);
        int grid_switch = 1;

         //Setup sdl screen stuff
    SDL_Surface* screen = NULL;
    SDL_Surface* surface = NULL;
```

```
    SDL_Rect rect;
    SDL_Rect offset;
    //Start SDL
    SDL_Init( SDL_INIT_EVERYTHING );
    surface = SDL_CreateRGBSurface(0, CELL_SIZE, CELL_SIZE, 32, 0, 0, 0, 0);

    screen = SDL_SetVideoMode( CELL_SIZE * MAX_X, CELL_SIZE * MAX_Y, 32,
SDL_SWSURFACE );
    SDL_WM_SetCaption(PROGRAM_NAME, NULL);



    ACTOR** current_grid;
    Uint32 color;
    SDL_Event event;
    int quit = 0;
    int pause = 0;
    while(!quit){
        if(!pause){
            if(grid_switch){
                current_grid = grid1;
                }
                else{
                    current_grid = grid2;
                }
                for(i = 0; i < MAX_X; i++){
                    for(j = 0; j < MAX_Y; j++){
                        switch(current_grid[i][j].type){
"

let std_template8 =                        "}
                        colorblock(rect, offset, screen, surface, CELL_SIZE * i,
CELL_SIZE * j, CELL_SIZE, CELL_SIZE, color);
                    }
                }
                if(grid_switch){
                    update(grid1, grid2, MAX_X, MAX_Y);
                    grid_switch = 0;
                }
                else{
                    update(grid2, grid1, MAX_X, MAX_Y);
                    grid_switch = 1;
                }
```

```
                              if( SDL_Flip( screen ) == -1 ){
                        return 1;
                    }
                }
        while(SDL_PollEvent(&event)){
           switch(event.type){
                        case SDL_KEYDOWN:
                              pause = !pause;
                              break;
                        case SDL_QUIT:
                              quit = 1;
                              break;
           }
        }
           SDL_Delay( STEP_TIME );
           }


    for(i = 0; i < MAX_X; i++){
           free(grid1[i]);
           free(grid2[i]);
    }
    free(grid1);
    free(grid2);
    //Free the loaded image
    SDL_FreeSurface(surface);
    SDL_FreeSurface(screen);
    //Quit SDL
    SDL_Quit();

           return 0;
}";


(* cal.ml *)
exception NoInputFile

let usage = Printf.sprintf "Usage: cal <filepath>"


let get_prog_name source_file_path =
      let split_path = (Str.split (Str.regexp_string "/") source_file_path) in
      let file_name = List.nth split_path ((List.length split_path) - 1) in
```

```
	let split_name = (Str.split (Str.regexp_string ".") file_name) in
		List.nth split_name ((List.length split_name) - 2)

let _ =
try
		let file_name =
			if Array.length Sys.argv > 1 then
				get_prog_name Sys.argv.(1)
			else raise NoInputFile in
		let input_chan = open_in Sys.argv.(1) in
		let lexbuf = Lexing.from_channel input_chan in
		let rev_prog = Parser.program Scanner.token lexbuf in
		let program = List.rev rev_prog in
		let semantic_check = Semantic.check_program program in
		let comp_result = if semantic_check = true then
			Compile.translate program file_name
			else raise(Failure("\nInvalid program.\n")) in
		print_string comp_result
with
	|	NoInputFile -> ignore (Printf.printf "Invalid filepath\n%s" usage
```