Cellular Automata Language (CAL)
Language Reference Manual

Calvin Hu, Nathan Keane, Eugene Kim
{ch2880, nak2126, esk2152}@columbia.edu

Columbia University
COMS 4115: Programming Languages and Translators

July 24, 2013

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.

2 Lexical Conventions

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.
```

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.

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
this
rules
return
def
init
this
actor_type

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.

2.4.1 Boolean constants

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

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

}

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;

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.

2.4.4 String constants

String constants are exactly the same as C string constants.
They are surrounded by double quotes and support escape characters such as
backslash: '\\'
newline: '\n'
tab: '\t'
double quotation marks: '\"'

Examples;
"hello";
string ss = "hello"; //declaration

2.4.5 Grid constants

Grid constants literals are created using ‘[‘ and]’. The ‘;’ means to start a new line on the grid. The ‘,’ means to start a new cell on the same line.

grid g = [A, B, 2,B; B, A, 1, C; A, A, A ,A];

This creates the following grid, g:

A	B	2	B
B	A	1	C
A	A	A	A

2.4.6 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>
                }

|
```

2.4.7 Direction constants

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

example:
direction d = north;

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.

4 Expressions and Operators

4.1 Unary Operators - group right-to-left

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

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.

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.

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.

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.

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.

5 Statements

5.1 Expression Statement

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

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}
```

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}
```

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;
```

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.

7 Declarations

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

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, …)

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) - Assigns the actor at the cell in direction d to actor_type a.

def void assign_type(grid g, int x, int y, actor_type a) - Assigns the actor at the cell at location (i,j) in grid g.

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(grid g, int x, int y) - Given a grid, x position, and y position, returns the actor in the cell location at the position.

def actor create_actor(actor_type a) - Returns a new actor of actor_type a.

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 void display(grid g) - Displays the grid.

def void run(grid g, int chronons) - Runs the cellular automata in the grid for chronons steps.

def void step(grid g) - Increments the state of the grid by 1 step.

def void print(string s) - Outputs the string to console.

def grid read_gridfile(string filepath) - Returns a grid based on a configuration in a file located at the filepath.

def grid create_grid(int width, int height) - Returns an empty grid with the given height and width. The grid is filled with Empty actors.

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);
```