CAL:
The Cellular Automata Language

Language Reference Manual

Ramses Driskell (rd2491) - Systems Architect
Percee Goings (pbg2111) - Group Manager
Fei-Tzin Lee (fl2301) - Systems Integrator
Geoffrey Loss (gml2151) - Tester and Validator
Andrei Tapai (amt2197) - Language Guru

Programming Language & Translators (COMS W4115)
Professor: Alfred Aho
Mentor: Kevin Walters
Spring 2015
Wednesday, March 25, 2015

Table of Contents

1. Lexical Elements

1.1 Identifiers

1.2 Keywords

1.3 Comments

1.4 Separators

2. Data Types

2.1 Primitive Types

2.2 Other Types

2.3 Grid and Cell

2.4 Typing

3. Expressions and Operators

3.1 Expressions

3.2 Assignment Operators

3.3 Incrementing and Decrementing

3.4 Arithmetic Operators

3.5 Comparison Operators

3.6 Logical Operators

3.7 Array Subscripts

3.8 Dot Operator

3.9 Functions as Expressions

3.10 Operator Precedence

4. Statements, Blocks, and Control Flow

4.1 Expression Statement

4.2 If, Else, Else-if Statements

4.3 While Statement

4.4 For and For-each Statements

4.5 Blocks

4.6 Null Statement

4.7 Break, Continue, and Return

5. Functions

5.1 Function Definitions

5.2 Function Calls

5.3 cal_it()

5.4 random()

6. Program Structure and Scope

6.1 Overall Structure

6.2 The Program Header

6.3 The Program Body and Control Flow

6.4 Scope

7. Full Grammar for the CAL Programming Language (Work in Progress)

1. Lexical Elements

1.1 Identifiers

Identifiers are sequences of alphanumeric characters plus underscores to define new functions, variables, and data types. They cannot start with numbers. Identifiers are case-sensitive.

1.2 Keywords

Keywords are reserved identifiers for use by the language which cannot be used for any other purpose. CAL contains the following keywords:

> grid is square triangular hexagonal cell cells neighbor neighbors that-is have if for else while else-if for-each color return continue break and or equals greater-than less-than xor greater-equals less-equals not nand nor in void int float double boolean true false char string

The words *square, triangular,* and *hexagonal* are reserved to describe shapes of cells. We are only implementing square cells for now but we plan to implement other shapes later so we have reserved *triangular* and *hexagonal* preemptively.

color we also are not yet implementing but are reserving preemptively.

have is used to describe properties of cells. *is* is used to describe properties of grids.

1.3 Comments

Comments are marked by > and <, as in:

```
>This is a single-line comment.<
>This is
a
multi-line
comment.<
```

1.4 Separators

CAL contains the following separators to separate tokens:

```
| ( ) [ ] , x
```

| is used for block statements.
(and) are used for parameters.
[and] are used for arrays.
, is used to separate parameters and between statements in a for loop.
x is used to separate the height and width of the grid in the “grid is” assignment.

White space is also a separator but is not a token of its own. Newline is used to separate statements.

The following is an example of these separators:

```
grid g is 30x30

int[8] x
cells have x

cell.x[7] = 0
for (i = 0, i less-than 8, i ++)
|
        if (i not-equals 7)
        |
                x[i] = i
        |
        for-each (cell in cells)
        |
                cell.x[7] += cell.x[i]    >I snuck a comment in this example too!<
        |
|
```

2. Data Types

2.1 Primitive Types

CAL contains the following primitive data types, all of which are keywords:

int   float   double   boolean   char

These all correspond to the same data types in Java. We did not include Java's other primitives for simplicity. These can all be returned by functions.

2.2 Other Types

CAL also contains non-primitive types *array* and *string*. These likewise correspond to their equivalents in Java. These can also be returned.

2.3 Grid and Cell

CAL also contains the types *grid* and *cell*. A grid is made of cells and is of a certain dimension (e.g. 50x50). For now, cells must be square, but later CAL may be expanded to include triangular and hexagonal cells. Cells have certain properties assigned by the user; this is discussed more in **Program Structure and Scope**. *cell* and *grid* cannot be returned.

2.4 Typing

CAL is statically typed.

3. Expressions and Operators

3.1 Expressions

An *expression* is n-ary where n is the number of operands, and is at least one. There are only two types of unary expressions: expressions with a unary operator and/or function calls. Beyond unary expressions, arity is not restricted. For example,

```
foo( )
++cal
```

An *operand* may either be a constant, variable or function call. For example,

```
int density = random( )
float PI = 3.14159265
grid g is 100x100
```

An *operator* performs either a one or two-step operation on its operand(s). Operators may have at most two operands. Grid, neighbor, and cell types are alphabetically exclusive meaning their operators can only be of the alphabetical type. For example,

```
boolean b = false
string reproduction = random("sexual", "asexual")
grid g is 100x100
g += 50x50 invalid
```

Parentheses can be used to increase subexpression precedence. For example,

```
(1 -  (3 / 4)) * 6
```

results in 1.5.

3.2 Assignment Operators

The assignment operator stores the right operand into the variable represented as the left operand. There are both symbolic and alphabetical assignment operators that can be used on primitive and extended types respectively. In addition, there are two-step assignment operators that do one step operations and assign the result into the left operand. For example,

```
char a  = 'a'
```

```
string b = “b”
grid g is 1337x1337
```

Here is a table of all the assignment operators:

Operator	***Description***	***Example***
=	Assigns right operand to left operand (primitive only)	white = “leukocyte”
is	Assigns right operand to left operand (grid only)	g is square
have	Assigns right operand to left operand (cells only)	cells have neighbors
+=	Adds right operand to left operand, then assigns result to left operand	even += 2
-=	Subtracts right operand from left operand, then assigns result to left operand	negative -= 1
*=	Multiplies right operand by left operand, then assigns result to left operand	square *= 2
/=	Divides right operand by left operand, then assigns result to left operand	self /= 1
//=	Floor divides right operand by left operand, then assigns result to left operand	century //= 100
%=	The remainder of the division of the right operand and left operand is assigned to the left operand	binary %= 2

3.3 Incrementing and Decrementing

Incrementing and decrementing are unary operations and can be either prefix or postfix. A valid increment and decrement operand must be a variable *ONLY*. The increment operator ++ adds one to the operand and the decrement operator -- subtracts one from the operand. For example,

```
> good code segment<
int i = 0
float f = 0.0
```

```
f++
i++

>bad code segment<
int i = 0
float f = 0.0
boolean b = false
f = 0++                              >variables only<
b--                                  >boolean is false or true<
i = --i                             >non standard; ambiguous :-( <
```

3.4 Arithmetic Operators

CAL provides six symbolic arithmetic operators: addition, subtraction, multiplication, division, floor division, and modular division. In addition the negation operation is possible through the use of the *not* keyword for booleans and the '-' symbol for numbers.

The addition operator is left associative and take two operands. For example,

```
mass = protons + neutrons
```

The subtraction operator is left associative and take two operands as well. For example,

```
neutrons = mass - protons
```

The multiplication operator is left associative and take two operands. For example,

```
acceleration = mass * velocity
```

Basic division, floor division, and modular division are all left associative and each take two operands. For example,

```
slice = pizza / 8
remain = pizza // slice
crust  = pizza % slice
```

Negation is a unary operation and is right associative. For example,

```
int i = -j
boolean here = not there
```

3.5 Comparison Operators

Comparison operations are alphabetically exclusive. Each operation uses boolean logic to determine if the left operand is equivalent to the right operand. Comparison operands may be either variables, literals, or functions excluding the aforementioned of the extended type.

The *equals* keyword denotes the operation that compares the equivalence of two operands and returns true iff both operands are of equal type and value. The *greater-than* keyword applies a similar operation but instead returns true iff the left operand is greater than the right operand. Similarly, *less-than* operation returns true iff the left operand is less than the right operand. The *greater-equals* operation returns true if either the right operand equals the left operand or it is less than the left operand. Lastly, the *less-equals* operation returns true if either the left operand equals the right operand or it is less than the right operand.

The not operator mentioned earlier can also be applied to each of the comparison operators to return the negation of the aforementioned operations.

```
> sample code snippet <
int water = 9
int fire = 10
int smoke = 10
while(water not equals fire)
|
        if(fire equals smoke)
        |
                fire /= smoke
        |
        smoke--
        if(smoke less-equals fire)
        |
                break
        |
|
call (water, fire, smoke)
```

3.6 Logical Operators

All logical operations are also alphabetically exclusive. Logical operators apply boolean rulesets to left and right operands. Operands to logical operators include variables, literals and functions again excluding the extended types. There are three basic logical operations that may be applied to operands: *and*, *or*, and *xor*.

For convenience, CAL also provides the *nand* and *nor* operations which are the complement to the *and* and *or* operations. ***Note: this is only for convenience, there is no semantic difference between the sets {nand, nor} and {not and, not or}.***

For example,

a	b	expression
0	0	a and b = a or b = false a xor b = false a nand b = a nor b = false
0	1	a and b = false a or b = true a xor b = true a nand b = true a nor b = false
1	0	a and b = false a or b = true a xor b = true a nand b = true a nor b = false
1	1	a and b = a or b = true a xor b = false a nand b = a nor b = false

3.7 Array Subscripts

Array members can be accessed by using the array name, subscript index, and square brackets. By placing the subscript index in square brackets, the member pointed to by that index in the array will be returned as a value.

3.8 Dot Operator

The dot operator is used to access built in methods for the extended types.

3.9 Functions as Expressions

Functions can be used in and as expressions. For example, statements

```
int number = random(1~9, 11, 14)
cal_it( )
```

are both valid uses of functions in CAL.

3.10 Operator Precedence

An expression with multiple operators adheres to the following precedence rules:

Operator Precedence (descending)
function calls, array subscripting
unary operators
multiplication, division, floor, division, modular division,
addition, subtraction
greater-than, less-than, greater-equals, less-equals
equals, not equals, is, have
and
xor
or
assignment
comma

4. Statements, Blocks, and Control Flow

4.1 Expression Statement

CAL treats any newline terminated expression as a statement. A statement in CAL allows some operation to take effect either as input, output, or memory manipulation.

There have been many example statements so far. Here is another.

```
boolean life
cells have life
```

4.2 If, Else, Else-if Statements

The *if* statement allows CAL users to conditionally run particular blocks of the program. Syntactically, the *if* statement takes a condition inside of parenthesis and a block of code to execute which is denoted by vertical bars. For example,

```
cells have life
if (cells not have live-neighbors)
|
        life  = 0
|
```

if statements can be executed in a logical manner. To specify a block of code to execute in the event the condition fails, *else-if* and *else* statements may be used. The difference between *else-if* and *else* flow control statements is that *else-if* allows another condition to be evaluated whereas the *else* is always executed in the event that the *if* and *else-if* conditions preceding it fails. *else-if* is only evaluated if all preceding *if* and *else-if* conditions have been false.

The example below will return 0 even though the *else-if* condition is true, since the *if* condition preceding it was also true.

```
int i = 1
if (i equals 1)
|
        i--
|
else-if (i equals 0)
|
        i = 4
|
else
|
        i = 5
|
return i
```

4.3 While Statement

The *while* statement allows a block of statements to be executed N times where N >= 0. The syntax for a *while* statement is similar to *if* and *else-if* statements. A condition must be specified within parentheses and a block of code to execute must be enclosed in vertical bars. Conditions can only use variables that have been declared prior to the *while* statement. The boolean values *true* and *false* may also be used in conditions. For example,

```
frame = 0
while(true)
|
        frame++
|
```

4.4 For and For-each Statements

CAL also has made the use of *for* loops valid. Syntax for the *for* statement is as such

```
for (expression, condition, expression)
|
        > code goes here <
|
```

Unlike the while statement, variables may be declared within the parenthesis of the *for* statement. As the code inside the block is optional, here is a valid for loop statement.

```
for (float b = 0.0, b less-than 10.0,  b++)
|     > nothing done here <          |
```

The *for-each* statement takes a condition within parenthesis. The condition must be group of cells, neighbors, or other iterable structure. For each member in the iterable structure, a block of statements are executed. The syntax is as follows,

```
for-each (s in set)
```

Let's put it all together.

```
boolean life = false
cells have life
cells have neighbors

for-each(cell in cells)
|
        if(cell.is_alive( ))
        |
                > do something with live cell <
        |
|
```

4.5 Blocks

A block in CAL is a series of one or more statements enclosed in vertical bars. CAL does not support anonymous nested blocks. Compare function1 with function2:

```
function1( )
|
        > code a <
        if(true)
        |
                > code b <
        |
|

function2( )
|
        > code c <
        |
                > code d <
        |
|
```

As seen above there is not a unique symbol for beginning and ending a block (such as { vs }). Therefore, there is no way to determine whether function 2 ends after code c, or after code d. However, it is clear that both code a and code b belong to function1 and will execute correctly.

4.6 Null Statement

CAL statements do not require a semicolon as some other programming languages. Instead, each line is treated as a statement. The null statement is a blank line that is newline terminated. This statement has no semantic effect on the program flow execution and is ignored.

4.7 Break, Continue, and Return

The *break* statement will stop the execution of one and only one containing control statement. Valid control statements include the *for*, *while* and *for-each* statements. For example,

```
for (int x = 1, x less-than MAX, x++)
|
        if (x % 2 equals 0)
        |
                break
        |
|
```

The above *for* statement will execute twice and terminate.

The *continue* statement stops execution flow and moves immediately to the next condition of the containing control statement.

The *return* statement stops execution flow of a function and immediately returns a value to the expression calling the function. Functions may return any primitive data type, arrays, and strings. Void functions may NOT use the return statement which will lead to compile errors.

For reference here is a snippet using *void* and *int* functions, as well as the *continue* statement.

```
int function1(int a )
|
        int b = 1
        for(a; a > 0; a--)
        |
                if(a % 2 != 0)
                |
                        continue
                |
                b *= a
        |
        return b
```

```
|

void function2( )
|
        int c = function( 20 )
        > do something with c
        and do not return <

|
```

5. Functions

5.1 Function Definitions

Functions provide a convenient way to group together instructions that might modify or return information associated with a CAL object. A typical CAL function will update the value of a particular cell or grid attribute based on the values of other object attributes. Every CAL program must have a cal_it() function which provides instructions for generating a subsequent grid configuration based on current cell and grid attributes.

CAL functions are defined in C-like manner; a function definition must first give the return type, then the function name and function parameters respectively and finally the function body.

```
Return Type    Function Name    (Param 1, Param 2 ...)
|
        Function Body
|
```

CAL, however, does not require function declaration. Functions that do not return values must have a return type of *void* and cannot contain *return* statements. Function names have the same restrictions on them as variable names do, they must begin with a letter and can be composed only of letters, numbers and underscores.

5.2 Function Calls

Functions are called by indicating the function name and providing a parenthesized, comma separated list of parameters.

```
        updateColor(cellA, “blue”)
```

Parameters can be CAL primitives, arrays, or strings.

```
        isEqual(“blue”, CellB.color)
```

Function calls can be nested within expressions and statements, and thus can be used for assignment and for evaluating more complex expressions.

```
        if (isEqual(“blue”, CellB.color))
        |
                CellB.color = randomColor()
```

```
    |
```

5.3 cal_it()

Every CAL program must have exactly one cal_it() function which functions as the main function and drives simulation of the cellular automaton. The cal_it() function's return type is void and it takes no parameters. Once the objects associated with a CAL program have been initialized, control is passed to the cal_it() function which is responsible for describing all of the logic required to generate subsequent states of the automaton. This function is repeatedly executed until program termination.

```
    void cal_it()
    |
        cal_it() function body
    |
```

There are no restrictions on the function body in regards to which cell or grid attributes get updated, any existing subset of cells can be modified on each iteration of the cal_it() function. Below, the same cell in the grid is updated upon each cal_it() function execution, and any remaining grid cells have their state left unchanged for the duration of the automaton simulation.

```
    void cal_it()
    |
        if(isAlive(CellA)
        |
            CellA.alive = 0;
        |
        else
        |
            CellA.alive 1;
        |
    |
```

A more useful cal_it() function would utilize the *for-each* statement to evaluate and update a group of cells with a common set of instructions.

```
    void cal_it()
    |
        for-each (cell in aliveCells)
        |
            updateA(cell)
        |
        for-each (cell in deadCells)
```

```
          |
                    updateB(cell)
          |
|
```

5.4 random()

The random() function is used for ease of initializing cell states. If the user wants a particular cell to start with a particular value for one of its properties, they may initialize it that way; if it is not important they may call the random() function to initialize it instead.

random() may be called with arguments or without arguments. If it is called without arguments, any value in the domain of that property. Strings may be any set of characters up to length 255, including the empty string. For instance:

```
int a
boolean maybe
boolean word
cells have a
cells have maybe
cells have word

for-each (cell in cells)
|
          cell.a = random()           >cell.a can be any integer value<
          cell.maybe = random()      >cell.maybe can be true or false<
          cell.word = random()       >cell.word can be any set of chars of any length 0-255<
|
```

Arguments passed to random specify the domain of random values which the property can take. '~' indicates a range and ',' separates different values. Strings can be randomized in three different ways: they can be chosen from a set of given strings, be random strings of a given length if one int is passed, or be random strings of a given range of lengths if a range of ints or multiple ints is passed. For instance:

```
int a
float b
string fourletters
string mood
string someletters
cells have a
cells have b
cells have fourletters
cells have mood
cells have someletters

for-each (cell in cells)
|
        cell.a = random(0~4, 7, 9)
        >cell.a can be 0, 1, 2, 3, 4, 7, or 9<

        cell.b = random(-3.5~-2.6, -1~4)
        >cell.b can be any float between -3.5 & -2.6 or -1 & 4<

        cell.fourletters = random(4)
        >cell.fourletters can be any string of length 4<

        cell.someletters = random(4~6, 8)
        >cell.someletters can be any string of length 4, 5, 6, or 8<

        cell.mood = random("happy", "sad", "apathetic", "crrrazy")
        >cell.mood can be any of the four strings given<

|
```

6. Program Structure and Scope

6.1 Overall Structure

A CAL program has two main sections: a header and a body. The header section includes initializations and global instantiations; the body includes function definitions and the cal_it() function, which behaves like the main() method in other languages.

6.2 The Program Header

The header of a CAL program can be broken down into four subsections:

1. Grid initialization
2. Global variable declarations
3. Cells-have statements
4. Cell initialization

Grid initialization includes the specification of grid size and shape, although for now grids may only be square and contain square cells. Global variable declarations may be placed anywhere within the program, as long as they are not within a code block, but by convention they should be placed immediately after the grid initialization. “Cells have” statements specify which global variables are attributes of all cells. Cell initialization statements initialize any attributes or values associated with each cell.

6.3 The Program Body and Control Flow

The only required component of the program body is the declaration of the cal_it() function. Other function declarations are optional. The GUI opens when the cal_it() function executes. CAL includes the basic control flow statements standard to most programming languages: if/else/else-if statements, for and while loops, and break and continue statements. These are equivalent to the Java versions of the same statements (the else-if keyword works the same way as an “else if” in Java). CAL also includes a special for-each cell statement, which iterates over every cell in the grid.

6.4 Scope

Variables defined within a block are called *local variables*; their *scope* is said to be limited to that block (that is, they exist only within the block they are defined in). Variables defined outside *any* block are called global variables; they are valid or visible within any block in the file.

Variables defined outside a file are not visible within the file; there is one CAL program per file, and no interactions between programs/files.

7. Full Grammar for the CAL Programming Language (Work in Progress)

Legend:

bold items are terminals
non-bold items are non-terminals

Grammar:

program → declaration-list
declaration-list → declaration-list declaration | declaration
declaration → grid-declaration | cells-declaration | variable-declaration | function-declaration

grid-declaration → **grid grid-id =** size-by-size | **grid is square**
size-by-size → number **x** number

cells-declaration → **cells have variable-id** | **cells.variable-id** = expression

variable-declaration → type-specifier variable-init
variable-init → **variable-id** | **variable-id =** expression
type-specifier → **int** | **float** | **double** | **boolean** | **char** | **string**
type-specifier → **int[**number**]** | **float[**number**]** | ...

function-declaration → type-specifier **function-id (** param-list **)** multi-statement
param-list → type-specifier **param-id** | type-specifier **param-id** param-list | ε

function-statement → **|** variable-declaration-list statement-list **|**
variable-declaration-list → variable-declaration variable-declaration-list | ε
statement-list → statement statement-list | ε
statement → expression | condi-statement | iterative-statement | return-statement | cont-break-statement

condi-statement → **if** (expression) statement-list | **if** (expression) statement-list **else-if (**expression) statement-list else-if-statement **else** statement-list
else-if-statement → **else-if (**expression) statement-list | ε

iterative-statement → **while** (expression) statement-list | **for(**variable-declaration, conditional, expression) statement-list | **for-each**(iterable **in** iterables) statement-list

iterable → **cell | neighbor**
iterables → **cells | neighbors**

return-statement → **return variable-id**
cont-break-statement → **break | continue**

expression → T | not T | V | X | Y
T → U | W
U → U + U | U - U | U * U | U / U | id
V → A = U | A += U | A -= U | A *= U | A /= U | A //= U | A %= U | V,V
W → W and W | W or W | W nand W | W nor W | W xor W
W → B equals B | B greater-than B | B less-than B | B greater-equals B | B less-equals B
X → C is D | C.D
Y → E have F | E.F
A → var
B → A | id
C → grid
D → grid-key
F → cell-key