CAL:
The Cellular Automata Language

Language Tutorial

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

1.0 – Tutorial Introduction to CAL	3
1.1 – Standards and Compiling	3
1.1.1 – File Standards	3
1.1.2 – Compiling and Running a CAL Program	3
1.2 – Your First Program in CAL	4-5
1.3 – A More Complicated Program in CAL	6-8
1.4 - The For Statement	8-10
1.5 - Using Arrays	10-12
1.6 - Conclusion	12

1.0 – Tutorial Introduction to CAL

The purpose of this introductory chapter is to immediately immerse new-comers into the world of CAL through exemplary programs that are both complete and correct. We will be going through multiple example programs from the very to more complex. Each example is intended to illustrate one or more concepts and features of the language. For experienced programmers, the following will be readily recognizable and intuitive (we hope!). For all those new to programming in any form, consulting the later chapters is strongly advised in order to understand in greater depth the components that make up a source program in CAL.

1.1 – Standards and Compiling

1.1.1 – File Standards

Source program files in CAL should contain ".cal" as a postfix or file type. For example:

my_program.cal
test.cal

are valid CAL source files, while *myprogram.c* is valid C source program but not a valid CAL file.

Each ".cal" file is a self-contained program meaning that it does not and cannot depend on any other file (".cal" or otherwise) in order to compile and run. Among other things, this implies that CAL does not support importing or building external libraries of functions.

Upon compilation, ".cal" files will become Java ".class" files which can then be run with the help of a program built specifically to display a graphical representation of your program.

1.1.2 – Compiling and Running a CAL Program

The specific details and command line input needed in order to take a CAL source program and turn it into an executable piece of code are currently to be decided upon by the development team. There is however no doubt that the output language of the compiler will be Java and thus users should have a Java environment installed on their machine. Additionally, it is more than likely that command line inputs will be a part of this process, so users should be familiar with UNIX command prompts.

1.2 – Your First Program in CAL

There is a tendency among introductory program tutorials to feature a so-called "Hello, World" program as the gateway program for beginners. This usually involves I/O operations such as a print statement. CAL does not naturally lend itself to user input and output. In fact, there is no mechanism in place for such operations outside of the initialization done inside the source program. Instead of a "Hello, World" program we'll take a look at another very simple program dubbed "Eternal Life" in CAL:

```
grid g is 3x3
g is square

boolean life

cells have life

cells.life = true

void cal_it()
|

|
```

Running the above piece of code will bring up a GUI representing a three-by-three grid of squares which will all be "alive" or, in graphical terms, colored in. The program itself does nothing beyond this simple initialization. However, in order to understand the structure and parts that go into writing a CAL program, it is more than sufficient.

Going through this program line by line, we will first tackle the first two lines. They make a reference to a "grid". The first line names the grid "g" and also defines its size "3x3". Thus, each CAL program starts with a definition of a grid by giving it an alphanumeric name and a size, specified by *length* x *width*. Next, we say "g is square". This further defines the grid g by specifying the shape of its constituent cells. Notice that this is a definition of only the shape of the constituent cells, not of the grid itself. Declaring "grid is hexagonal" (feature not implemented) does not make the grid itself hexagonal in shape. Rather, the cells themselves become hexagonal.

At this point, we have a full definition of the space on which our program will run. Next we have the global variable declaration "boolean life". This statement declares a boolean

called “life” to exist. It is as of yet uninitialized. Global variables are within the scope of any function. CAL provides several primitive types (see table below):

int	an integer
float	single-precision floating point
double	double-precision floating point
boolean	*true* or *false*
char	a single byte character

In addition to these primitive types, CAL also provides the types *array* and *string*. See the language reference manual for more details on these.

However, our next statement, “cells have life” takes the global variable life and makes it a property of cells. Now each cell has within its structure an accessible boolean called life. This value may be different for two different cells. Further, this value can be accessed with the dot operator on “cells”.

This is demonstrated in our next statement “cells.life = true”. This statement accesses the life boolean in all cells and initializes it to true.

Next, we come to the completing step of our program, the cal_it() function. This function serves as a “main” function for CAL. Similar to other languages that feature a mandatory function that controls the overall flow of the program, CAL uses and requires the “cal_it()” function in the same way. Notice three points here. The return type of “cal_it()” is always void since the function will never return a value. Second, its parameters block is always empty since the user will never supply it with input. Third, the function itself can be and is empty, indicating that it does nothing.

1.3 – A More Complicated Program in CAL

Next we will attempt to add many more features of CAL into a sample program in order to demonstrate the whole range of options at your disposal when programming in CAL.

```
>This program is an implementation of Conway's Game of Life. What you are
currently reading is a properly formatted multi-line comment in CAL.<

grid g is 100x100
g is square

boolean life

cells have life

cells.life = random()

int num_live_neighbors(cell)
|
    int live = 0

    for-each cell.neighbors
    |
            if (neighbor.life equals true)
            |
                    live++
            |
    |

    return live
|

void cal_it()
|
    int liveneighbors = 0

    for-each (cell in cells)
    |
            liveneighbors = num_live_neighbors(cell)
```

```
            if (cell.life equals true and liveneighbors less-than 2)
            |
                    cell.life = false
            |
            else-if (cell.life equals true and liveneighbors greater-than 3)
            |
                    cell.life = false
            |
            else-if (cell.life equals false and liveneighbors equals 3)
            |
                    cell.life = true
            |
            else
            |
            |
        |
    |
```

The beginning of this program is what we'd expect from a CAL program, given our current knowledge. The first three lines initialize the grid and declare a global. The fourth line features something new, however. It initializes life with the built-in function *random()*. It is sufficient to state that this function assign a random value to each instance of life in each cell. random() can be passed parameters such as single values or ranges of values (eg. 4-9, 6-15, etc.) However, when it is left blank, random() chooses a value from the entire possible set of values for the respective variable. In our case, that set comprises only *true* and *false*.

Next we have our first function! And it has a non-void return type! It returns an int. Notice that the naming scheme for functions is alpha-numeric with underscores. In the parameters block, we have our first passed parameter. It is designated simply as cell. This is meant to indicate that it operates on the set of cells. Normally if you run a function on one cell, you run it on all cells (all cells in a system are subject to the same rules). Moving on to the body of the function, we have our first local variable in the form of *int live*. This variable can only be accessed and modified by *num_live_neighbors*.

Next we have our first loop in the form of the for-each statement. This statement provides a way of iterating through a group of data. In this case, cell.neighbors. Each cell has, by definition, x neighbors (x can be 8, 5, or 3 neighbors depending on the cell's position). This for-each statement then goes through the passed cell's neighbors and operates on them. In this case, we make use of an *if* statement and pass it the condition *neighbor.life equals true*. So if the neighboring cell is alive, we increment the local variable by one (live++). After we have checked and incremented appropriately, we return the local variable live's value to the calling function.

Next we have our mandatory cal_it() function. In here we have statements and declarations like we have in our previous function. The only difference being that we expand on our use of *if* statements. We add *else-if*, and *else* as control blocks to expand the our control over conditions that we want to check for and act on. In this case we check for the number of neighbors and for the current cell's status. Based on this information, we change the state of

the current cell from alive to dead and vice versa. Cells can also remain alive if alive and dead if dead if not conditions are met.

1.4 - The For Statement

In addition to the for-each loop there are other constructs in CAL that allow one to iterate through a set of values. The simple *for* loop is one such tool. It allows the user to not only declare and initiate a variable but to use that variable as part of a condition for executing a block of code. Below is the format for using a for loop:

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

Now lets see it in action!

```
grid g is 100x100
g is square

boolean life
int number

cells have life
cells have number

cells.life = random()
cells.number = random(0~9999)

void cal_it()
|
        for(int i = 0, i less-than 10000, i++)
        |
                for-each (cell in cells)
                |
                        if (cell.number equals i)
                        |
                                cell.life = true
                        |
                        else-if (cell.number greater-than i)
                        |
                                cell.life = false
                        |
                |
        |
|
```

First thing to notice is the random function. We've passed it a parameter! Before, it contained no parameters. The tilde (~) indicates a range of values to choose from. In this instance we specified the range as being anywhere from 0 to 9,999 (inclusive). See below and in the reference manual for more on random() parameters

```
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<
```

If you look at a *for* statement, it should seem like compressed *while* statement. They both are generally used in the same way except in a *for* statement the initialization and step phases are passed as expressions at the beginning of the block or loop.

Another concept to understand is that of nested blocks. In the above example program we have *for-each* loop *nested* inside a *for* loop. What effect does this have? In order to understand it fully, first realize that in each run of a *for* or *for-each* block, all the code inside it is run once. Therefore, if we are to look at the outer *for* loop in our example, we realize that:

1. The *for-each* loop resides within the *for* loop and is therefore run on every iteration of the *for* loop.
2. The *for-each loop* itself runs until it has iterated over each cell in the set of cells.
3. Therefore, the amount of times the *for-each* loop runs is 10,000 (the number of iterations it takes for i to invalidate "i less-than 10000") *times* 10,000 (the number of cells on the grid to iterate over). This amounts to *one hundred million* runs of the code inside the *for-each* loop.

The above should serve as a cautionary segment on nesting *for* loops and what effect they may have on the run time of your program.

We have spoken at length about the *for* loop but have not, as of yet, discussed what our example program actually does. Firstly, notice that our cells now have two properties instead

of just one: *life* and *number*. Number can range from 0 to 9,999 and is random for each cell. Cells are also randomly alive or dead.

Next, we can consider our cal_it() function and its contents. In our *for* loop we specify the condition that it should run until the value of *i* reaches 10,000. Then *for* every value of *i*, our *for-each* loop iterates through all the cells and checks their stored *number* value. If it is equal to the current value of i, the cells comes to life or stays alive; if the value of the cell is greater than i, it becomes dead.

In this way, after the first pass of the *for-each* loop, all cells with *number* equal to 0 will be alive and all other cells will be dead. In subsequent passes, the algorithm will light up cells in ascending order until the entire grid is filled in.

1.5 - Using Arrays

In addition to the primitive data types featured in CAL, there is also a readily available data structure in the form of a one-dimensional array. A user can form an array of any of the primitive types. For example:

int[10] intarray
boolean[15] boolarray

are valid examples. However,

cell[200] cellarray
grid[15] gridarray

are not.

Arrays always begin at index 0 in CAL as in other languages. This means that an array initialized as int[5] will have five slots to store ints indexed {0, 1, 2, 3, 4}. This convention is often reflected in the formatting of *for* statement expressions and conditions.

Lets take a look at a sample program that uses arrays:

```
grid g is 100x100
g is square

boolean life
boolean[8] neighbor_life

cells have life

cells.life = random()

boolean next_generation(cell)
```

```
|
        int i = 0
        boolean temp

        for-each(neighbor in cell.neighbors)
        |
                neighbor_life[i] = neighbor.life
                i++
        |

        for(int j = 0, j less-than i, j++)
        |
                if (j equals 0)
                |
                        temp = neighbor_life[j]
                |
                else
                |
                        temp = temp xor neighbor_life[j]
                |
        |

        return temp
|




void cal_it()
|
        for-each (cell in cells)
        |
                cell.life = next_generation(cell)
        |
|
```

In this example, our array is “boolean[8] neighbor_life”. You can see from its construction that it specifies an array with eight slots that hold boolean primitive values.

The program itself does a simple task: it determines the next state of each cell by xor-ing the states of all its current neighbors. The bulk of the work is handled in the next_generation function. This function is passed a cell from cal_it() and iterates through that cell’s neighbors with a *for-each* loop, populating the boolean array with the neighbor’s life state. Notice that we index into the array by specifying a location within the brackets.

Next we take this filled array and iterate through its contents, xor-ing the current value with the next index as we go along. Xor is a special operator featured in CAL. For a full description of operators check the language reference manual.

We then return the final boolean value to cal_it() where it becomes the next state of the cell we are operating on.

1.6 - Conclusion

This has been a tutorial for the CAL programming language. Hopefully, you found it both easy to read and informative. If you feel as if you are ready to begin programming in CAL as you are reading this, then we have done our job. We would advise you to take some time and look over the language reference manual for a more in depth look at certain aspects of our language.

Best,

Team CAL

END