UNIVERSITÉ TOULOUSE III PAUL SABATIER

Labwork 1 – AutoCell
a language for cellular automata

...

These labworks are automatically assessed based on an archive you have to deliver on time on the Moodle webpage. To make the archive, you have to type the command:

```
> make archive
```

This produces a file named `archive.tgz` that you have to deposit. As the compilation labworks are held each week, deposit deadline is set one day before your next session (to perform the assessment). This deadline is not strict but delay will have a negative impact on your mark and you will not benefit from the comments of your teacher.

This labwork is mainly an introduction to the compiling and execution environment you will have to use. To help you to organize your work, a time indication is given on each section of this protocol.

1 Cellular automata (10 mn)

A cellular automaton [1] is a discrete model of computation based on a grid possibly containing a set of values. The automaton evolves at each step by computing a new value for each cell from the cells surrounding it. Each new grid obtained after the complete computation of a cell is called a generation. The computation can then be iterated as many generations as we like. To get an evolved example of this, you can play the Griffeath automaton movie on Moodle.

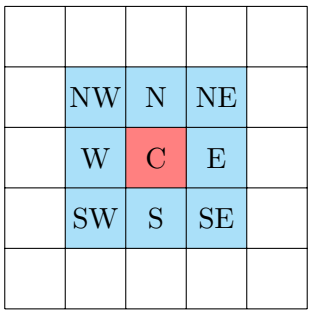
Maybe the most known sample of cellular automaton is Conway's game of life[2]. In this automaton, each cell may only have two states: white (dead) or black (alive) and the following rules apply at each generation:

- any live cell, with fewer than 2 live neighbours, dies (isolation);
- any live cell, with 2 or 3 live neighbours, stay alive;

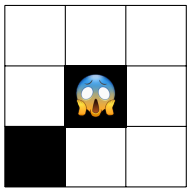
[1] `https://en.wikipedia.org/wiki/Cellular_automaton`
[2] `https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`

- any live cell, with more than 3 neighbours, dies (surpopulation);
- any dead cell, with exactly 3 alive neighbours, gets alive (spreading).

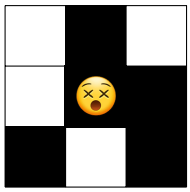
The neighbours are the 8 cells that are around the current cell:



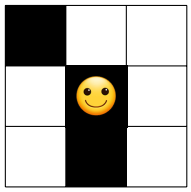
Examples of behaviours:



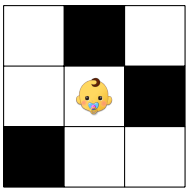
Isolation.



Surpopulation.



Persistance.



Spreading.

Surprisingly, these simple rules give birth to a large range of behaviours that can be composed to make even richer applications. It has been shown that Conway's game of life (a) can implement itself and (b) is Turing-complete: it is possible to implement the same behaviour as *NAND gates*, the microprocessors are made of, and in turn, microprocessors are able to execute Turing-complete programming languages.

There exists a lot of other cellular automata often named after their inventor, often with more complex cell states: Ulam's lattice network, Von Neuman's self-replicating systems, Langton's ant, etc.

2 Autocell (10 mn)

Cellular automata can be implemented with usual programming languages but dedicated languages (or *Domain Specific Languages*) allow to get rid of the burden of managing the grid and the generation switches and storage. With such programming languages, the developer only has to focus on the current cell and its neighbours and let the machinery, the *run-time* added at compilation time, manage the rest of the calculations.

This labwork aims to perform the compilation of such a language, named *Cellang*. The program below shows how Conways's game of life is written in *Cellang*:

```
1 2 dimensions of 0..1 end
2
3 sum := [-1, -1] + [0, -1] + [1, -1] + [1, 0]
```

```
4       + [1, 1] + [0, 1] + [-1, 1] + [-1, 0]
5
6  if [0, 0] = 1 then
7      if sum < 2 then
8          [0, 0] := 0
9      elsif sum > 3 then
10         [0, 0] := 0
11         end
12 else
13     if sum = 3 then
14         [0, 0] := 1
15     end
16 end
```

In *Cellang*, there are two types of memory locations. The named memory locations like `sum` represent variables. Memories of the form `[i, j]` represent values of cells in the grid with $i, j \in \{-1, 0, +1\}$ according to the scheme below:

[-1, -1]	[0, -1]	[+1, -1]
[-1, 0]	[0, 0]	[+1, 0]
[-1, +1]	[0, +1]	[+1, +1]

The memory locations support only one type: integer. They can be assigned with the symbol `:=` and an expression using usual structure made of constants, memory locations and operators. Through the cell locations, only the cell at `[0, 0]` can be assigned.

The program is executed in order, one instruction after the other one. The language only contains one structured instruction: the `if` ... `then` ... `else` ... `end`.

This program above works this way:

line 1 The cellular automaton works only with 2 dimension grids (and our compilation is limited to this configuration) and the values of the cells are between 0 and 1.

line 3 The variable `sum` is the sum of all neighbours of the current cell. Notice that the variables have not to be declared.

lines 6, 7, 8 If the current cell is alive (1) and the number of neighbours is less than 2, then it dies.

lines 6, 9, 10 If the current cell is alive and there are too many neighbours, then it dies.

lines 12, 13, 14 If the current cell is dead (not 1) and if it has exactly 3 neighbours, then it gets born.

In all other cases, the state of the cell is unchanged.

The following sections present the environment where the *Cellang* language will be translated and executed.

3 AutoCell Environment (15 mn)

Autocell is a software suite dedicated to the execution and observation of cellular automata implemented in *Cellang*. It contains:

- a *virtual machine* (VM) capable of executing programs and of generating a cellular automaton grid,
- `autocc` – a compiler from *Cellang* to VM assembly,
- `autoas` – an assembler from VM assembly to VM executable,
- `autocell` – a graphics environment running the VM, displaying the grid and providing interaction with the VM (for display or debug purposes).

To Do

1. Download and unpack the archive `autocell.tgz in your preferred directory.`

   ```
   > tar xvf autocell.tgz
   ```

2. Compile the code of the *Autocell* environment.

   ```
   > make
   ```

3. Look at your first program in *Cellang*.

   ```
   > cat autos/shift.auto
   ```

4. Compile it.

   ```
   > ./autocc autos/shift.auto
   ```

5. Look at the translation in VM assembly of this program.

   ```
   > cat autos/shift.s
   ```

6. Assemble it.

   ```
   > ./autoas autos/shift.s
   ```

7. Run it in `autocell` with an initial grid (also called a map).

```
> ./autocell autos/shift.exe maps/circ.map
```

The last command opens your browser with the display of *Autocell* user interface. It displays on the left the grid of the automaton and on the right the assembly code of the program. Below, the register values can be expanded and a console is provided.

Breakpoints, as found in any debugger, can be set on the assembly instructions (by clicking on its left) and on particular cells (by clicking on it). The following menu commands are available:

Reset Reset the state of the grid to initial values.

Next Move to the next generation.

Step Execute one assembly instruction.

Continue Execute the program until a breakpoint is found.

Run Execute the program (ignoring breakpoints).

About Display information about *Autocell*.

Quit Leave the user interface.

To Do Play a bit with the user interface to understand how the different commands work.

4 The virtual machine (30 mn)

This labwork uses a *Virtual Machine* for 2 reasons:

1. The compiler is portable and independent of any specific hardware.
2. This will make the compiler life easier by getting rid of real hardware constraints, such as a bounded number of registers.

The program in the VM is structured as a sequence of instructions, located at consecutive addresses in memory. Each instruction may be made of four components (and are therefore also called *quadruplets*):

- the command to perform,
- 0 to 3 operands.

The operands may be register numbers or integer constants. For example, the instruction $ADD\ R_0, R_1,\ R_2$ performs the addition of $R_2 + R_3$ and stores the result into R_0.

The following calculation instructions are available:

$ADD\ R_i, R_j, R_k$ – addition

$SUB\ R_i, R_j, R_k$ – subtraction

$MUL\ R_i, R_j, R_k$ – multiplication

$DIV\ R_i, R_j, R_k$ – division

$MOD\ R_i, R_j, R_k$ – modulo

In addition, the instruction $SET\ R_i, R_j$ copies the content of R_j into R_i and $SETI\ R_i, k$ assigns to R_i the constant value k.

To manage the execution flow, as classic assembly languages, branch instructions are used. Addresses of instructions are obtained by prefixing them with a label, L :. Then, to branch to this instruction, the instruction $GOTO\ L$ can be used. Conditional branches also exist like $GOTO\_cond\ L, R_i, R_j$ with *cond* comparing R_i with R_j and branching if the condition is true. The following cond are available:

EQ – $R_i = R_j$

NE – $R_i \neq R_j$

LT – $R_i < R_j$

LE – $R_i \leq R_j$

GT – $R_i > R_j$

GE – $R_i \geq R_j$

Finally, the program must end with an instruction $STOP$.

For instance, the program below performs 2 iterations:

```
        SETI R0, #0
        SETI R1, #1
        SETI R2, #2
L1:
        GOTO_GE L2, R0, R2
        ADD R0, R0, R1
        GOTO L1
L2:
        STOP
```

To Do Type the program in the file `code/ex1.s`, assembly it and execute it step by step in *Autocell*.

To interact with the manager of the cell automaton grid, the program has to perform run-time calls using instruction $INVOKE\ c, a, b$. These calls allow to get and set values in the grid. c is always a command code while the meaning of a and b depends on the

command itself. a and b may be integer constants, register numbers to get values from or register numbers to store something inside.

The following c commands are available (code in parenthesis):

cSIZE (1) – to get the size of the grid (register of number a get the width and register of number b get the height). (example: INVOKE 1, 0, 5)

cMOVE (3) – to move the current cell (register of number a provides the x and register of number b provides the y). (example: INVOKE 3, 1, 6)

cSET (4) – set the value of the current cell (register of number a provides the value to set, b must be set to 0).

cGET (5) – get the value of an involved cell, current or neightbours (a is the register number to store the cell value, b is a direction constant as defined below).

Notice The only way to assign a cell (**cSET**) is to make it the current cell (**cMOVE**).

The position in b can be one of:

pCENTER (0) – current cell

pNORTH (1) – cell at [0, -1]

pNORTHWEST (2) – cell at [-1, -1]

pWEST (3) – cell at [-1, 0]

pSOUTHWEST (4) – cell at [-1, +1]

pSOUTH (5) – cell at [0, +1]

pSOUTHEAST (6) – cell at [+1, +1]

pEAST (7) – cell at [+1, 0]

pNORTHEAST (8) – cell at [+1, -1]

The invocation below get the value of the cell at south west in the register R_0:

```
INVOKE 5, 0, 4
```

To Do In the file `code/ex2.s`, write the program that sets 1 in the four corners of the current map. Assemble it and test it in the *Autocell* environment (with the map `maps/circ.map`). *Notice that the new generation is only displayed when the program completes after the STOP instruction.*

Alternatively, it is often faster to use the `autoexec` command that displays the execution in textual format:

```
> ./autoexec -v -s 1 code/ex2.exe maps/circ.map
```

5 Exercises

1. In VM assembly, write a program (in file `code/ex3.s`) that sets all cells to 1 and test it with `maps/circ.map`.

2. In VM assembly, write a program (in file `code/ex4.s`) that update the whole grid by setting each cell to 1 if the cell at the left (west direction) is equal to 1.

3. In VM assembly, write a program (in file `code/ex5.s`) that, for the whole grid:

   - If the cell is 1, set it to 0.
   - If the cell is 0, set it to 1 if one of the neighbour cells is 1.

   Test it with `maps/circ.map`.