Cellang: Language Reference Manual

J Dana Eckart
Radford University

15 July 1997

1 Introduction

The Cellang programming language is a special purpose programming language for the programming of a wide variety of cellular automata. The language is quite small and simple. During its design, care was taken to create a language that would be expressive enough to allow a wide variety of cellular automata to be written while restricting the possible programs to those considered truly cellular in nature. Cellang is intended both as an aid in introducing university students to the wonderful world of cellular automata as well as a common language in which to describe such systems for both teaching and research purposes.

This reference manual specifies the form and meaning of Cellang programs. Its purpose is to ensure the portability of Cellang programs. Thus the set of accepted programs and the set of rejected programs should be identical for all Cellang compilers. Furthermore, the proper execution of an accepted program should produce the same result regardless of the compiler used.

1.1 Language Summary

Cellang programs are composed of comments, statements and a cell declaration. The cell declaration indicates the number of dimensions of the automata, the number of field values contained within each cell of the automata, and the range of values associated with each field. The cell declaration must appear before any statements are given. Statements are either loops, selections or assignments. Assignments provide the only mechanism whereby cells can change state. Variables are implicitly declared at the point of first assignment and correspond to either a complete set of field values associated with a cell or to the builtin data type, integer.

1.2 Syntax Notation

A variant of Backus-Naur-Form is used to describe the syntax of Cellang programs. Narrative rules describe both the effects of each construct and the composition rules for constructs. The particular variations of Backus-Naur-Form used are:

1. Lower case words, possibly containing underscores, are used to denote syntactic categories. Whenever the name of a syntactic category is used apart from the syntax rules themselves, spaces take the place of the underscores.
2. Boldface is used to denote reserved keywords.
3. Alternatives are separated by right arrows ($\rightarrow$).
4. Curly braces enclose items that may appear zero (0) or more times.

2 Lexical Elements

Programs are represented by collections of lexical elements. Lexical elements are one of: comments, reserved keywords, identifiers, operators, separators or numeric literals. Lexical elements, in turn, are collections of ASCII (American Standard Code for Information Interchange) characters.

2.1 Comments

A comment starts with a sharp (#) and extends to the end of the line. A comment can appear anywhere within a program. The presence or absence of comments has no influence on whether a program is legal or illegal. Furthermore, comments do not influence the effect of a program. Any character may appear within a comment, except of course any character denoting the end of a line.

2.2 Reserved Keywords

For readability of this manual, the reserved keywords appear in lower case boldface. The reserved keywords are:

agent	**const**	**dimensions**	**else**
elsif	**end**	**exit**	**for**
forall	**if**	**of**	**otherwise**
then	**when**		

The case of the alphabetic characters that form a reserved keyword is ignored, thus `ELsE` and `eLSe` both represent the keyword **else**.

2.3 Identifiers

Identifiers are used as names. Identifiers must consist of at least one character, but are otherwise unlimited in length.

identifier $\rightarrow$ letter { underscore_option letter_or_digit }

underscore_option $\rightarrow$ underscore
$\rightarrow \lambda$

Any identifier constructed from the above rules are legal except when it also names a reserved keyword. In this case the lexical element is a reserved keyword. All characters of an identifier are significant, including any underscore characters; however, as with reserved keywords, the case of the alphabetic characters that form an identifier are ignored. All non-reserved identifiers represent variables. There are three predefined identifiers: `cell`, `time`, and `random`. While `time` and `random` can only be used within expressions (i.e. not assigned), `cell` can be used within expressions as well as assigned new values and refers to the cell with relative index `[0, ..., 0]`. The first lexical appearance of an identifier, on the left hand side of an assignment statement, implicitly declares it as a variable.

2.4 Operators and Separators

The simple (one character) operators are:

* / + – % = < > & | !

and the compound (two character) operators are:

<= >= != +% –% –>

In some cases an explicit separator is required to separate adjacent lexical elements (namely, when without separation, interpretation as a single lexical element is possible). A separator is any of:

[ ] , : . := .. ( )

Space characters, format effectors, and the end of a line are also separators. A space character is a separator except within a comment. Format effectors other than horizontal tabulation are always separators. Horizontal tabulation is a separator except within a comment. One or more separators are allowed before, after or between lexical elements.

The end of a line is always a separator. The language does not define what causes the end of a line. However if, for a given implementation, the end of a line is signified by one or more characters, then these characters must be format effectors.

2.5 Numeric Literals

A numeric literal is expressed in the conventional integer decimal notation. Underlines are ignored in determining the value of the numeric literal, and should be used to increase readability.

numeric_literal $\rightarrow$ digit { underscore_option digit }

2.6 String Literals

A string literal is a non-empty sequence of characters contained within double quotation marks. String literals are used for indicating the name of a file from which the values contained within a constant array may be read. String literals may not contain double quotation marks nor may they extend past the end of a line.

```
string_literal → double_quote character
                    { character } double_quote
```

3 Syntax and Semantics

The lexical elements of the previous section are combined in accordance with the rules given in this section to form correct programs. The semantics of these programs are also described. Examples are used to clarify rules that might otherwise be confusing.

3.1 Constant Declarations

Constant declarations represent numeric values that do not change while the program runs. Their use supports improved readability and greater flexibility in program design.

```
declare_constant → simple_constant
                 → array_constant

simple_constant  → const identifier
                      := signed_constant

array_constant   → const identifier [ ]
                      for constant
                      := signed_constant
                         { , signed_constant }

constant         → numeric_literal
                 → identifier

signed_constant  → sign_option constant

sign_option      → +
                 → -
                 → λ
```

A constant is either a numeric literal or an identifier which is the name of a previously declared simple constant. Simple constants associate a signed constant with an identifier. The size of an array constant is given by a constant, the value of which must be positive. The values of an array constant may either be given directly as a list of signed constants, or read from a file. If the array values are given in the code, the number of values given must match the size of the array. If the array values are read from a file, the file must contain at least as many values as the size of the array. The indexing of array elements begins with zero (0).

Examples:

```
const min_value := -15
const max_value := -min_value
const size := 5
const rates[ ] for 3 := 1, 2, 3
const times[ ] for size := 1, 2, 3, 4, 5
const groups[ ] for 256 := "groups_table"
```

3.2 Cell Declaration

The cell declaration describes the set of cells that make up the cellular automata. The declaration must appear only once in the program, and must precede any statements, with the exception of constant declarations which may appear before the cell declaration.

```
cell_declaration → constant dimensions
                      of field_list

field_list       → range
                 → { const_option field }
                      agent_fields end

const_option     → const
                 → λ

agent_fields     → agent of field { field }
                 → λ

field            → const_option field_type of range

field_type       → array_field
                 → ident_list

array_field      → array_name [ ] for constant
                 → λ

array_name       → identifier

range            → signed_constant .. signed_constant

ident_list       → identifier { , identifier }
```

The constant appearing before **dimensions** specifies the number of dimensions, in a discrete Cartesian space, to be used in computations. The maximum number of dimensions allowed, and the size of each, is determined by the implementation. The field list gives a description of the information associated with each cell of the automata. If only a single field is given, then no field name (identifier) is needed. When more than one field is used, then each field is represented by an identifier and associated range. An ident list allows a number of identifiers to be associated with a single range.

The fields given before the reserved keyword **agent** are known as cell fields, while those appearing after are called agent fields. If agent fields are given, then zero (0) or more agents may exist at a cell in the cell universe at any time step during the computation. The number of agents associated with a cell is potentially unbounded with each agent's state being specified by the contents of its fields. Each identifier used for a field name or array field, whether cell or agent, must be unique. If no cell fields are given, then agent fields must be declared.

When an array field is given, the array name is used to collectively refer to a number of elements. The constant given indicates the number of elements that the array field will contain. The indexing of array elements begins with zero (0).

The **const** keyword may only appear at the beginning of a cell field (i.e., before the **agent** keyword). If the **const** keyword is given, then the ident list or array name declared by the field is constant. Constant cell fields cannot have their value changed by any statement of the cellular automata program. The value of a constant field may only be altered by reading new values from input. Implicitly declared variables with the same structure as a cell may, however, have their constant fields assigned.

A range defines an inclusive set of integer values. The first is the lower bound and must be less than or equal to the second which is the upper bound of the range. A range indicates the permissible values that a field can be assigned. It is an error to assign a field a value outside of its associated range. The maximal size and allowed bounds of a range are determined by the implementation.

Examples:

```
2 dimensions of 0..255

const dims := 3
dims dimensions of
    const x, y of –9..9
    distance of 0..999
end

const size := 4
2 dimensions of
    count of 0..3
    const value[] for size of –9..9
agent of
    type of 0..5
    size of 0..25
end

const max := 5
2 dimensions of
agent of
    type of –max..max
end
```

3.3 Expressions

An expression is a formula that defines a computation.

```
expression      → expression
                      { binary_op expression }
                → unary_op primary
                → primary

unary_op        → +
                → –
                → !

binary_op       → +
                → –
                → *
                → /
                → %
                → &
                → |
                → =
                → <
                → >
                → <=
```

	→ >=
	→ !=
primary	→ index
	→ identifier field_reference
	→ relative_index field_reference
	→ array_reference field_reference
	→ array_constant
	→ (expression)
relative_index	→ [index { , index }]
array_reference	→ identifier [expression]
field_reference	→ . identifier
	→ . array_reference
	→ λ
array_constant	→ identifier [expression]
index	→ signed_constant shift_ops
	→ index_variable shift_ops
shift_ops	→ +% shift_amount shift_ops
	→ −% shift_amount shift_ops
	→ λ
shift_amount	→ constant
	→ index_variable

The predefined identifier `time` evaluates to the current step of the execution. The initial value of `time` is zero (0), and is increased by one (1) each time all of the cells in the automata have been updated. The predefined identifier `random` evaluates to a new uniformly distributed random value each time it is used. The random value will be zero (0) or greater with implementations required to have a minimum upper bound of 32767 on the range of random values. The predefined identifier `cell` yields the value of the current cell, which is also the value of `[0, ..., 0]`

There are 7 levels of precedence for the unary, binary and shift operators. Operators belonging to the same level are evaluated from left to right. The precedence levels, from highest to lowest are:

1. +% −%
2. ! (logical negation)
3. (unary) + −
4. * / (quotient) % (remainder)
5. (binary) + −
6. = < > != <= >=
7. & (logical and) | (logical or)

The relational and logical operations return integer values, with a zero (0) being returned when the condition is false and a one (1) when the condition is true. Unlike all of the other binary operations, the results of binary relational operations must not be immediately used by another binary relational operation. If no fields, of a multi–field value, are specified then only the equal (=) and not equal (!=) operations are applicable. In this case, both sides of the operand must be values of the same kind (cell or agent containing only fields of that type), with all of and only the fields of that kind being used in the comparison.

The quotient (/) and remainder (%) operators when applied to `a` and `b` (e.g. `a % b`) yield the following results.

- The quotient has the same sign as `a * b` and of the two integer values on either side of the real valued result, its magnitude is the one whose value is closest to 0.
- The remainder is given by `a - b * (a / b)` when `b` is non-negative and by `a + b * (a / b)` otherwise.

If an array reference or an array constant is used, then the identifier naming the array must be a previously declared, either as a field, a variable, or a constant array name. The range of the array is assumed to be zero (0) based and the value of the expression must be a non-negative integer. If the expression used to reference an element of the array contains an index variable, then if the range of the index variable was not given by the forall loop the range of the index variable will be inferred to be the size of the array it is being used to reference. Inferred ranges are always zero (0) based. If the range of an index variable was not given in the forall loop, all of the inferred ranges must be the same.

The shift operators can only be applied to index variables of forall loops. The associated simple constant or index variable indicates the amount to be shifted. The shift operation performs a circular shift taking the index range associated with the index variable as a cyclic group. Thus the +% operator performs an addition modulo the size of the group and the −% operator performs a subtraction modulo the size of the group. The magnitude of the combined amounts of a sequence of circular shifts must be less than or equal to the range of the index variable being shifted.

When referencing a neighboring cell using a relative index, the number of indices given must match exactly the number of dimensions indicated by the cell declaration. Such a reference is taken relative to the current cell of computation. Since it is possible that adding the index to the current cell location might give an index value outside the range of that dimension, "wrap around" is used to insure that the reference is a legal one. Thus neighboring cell references can "wrap around" the edge of the cells of the automata. Furthermore, each relative index must have an absolute value no larger than the size of the corresponding dimension (as determined by the implementation).

The optional field reference indicates which field value to use. Field references which are agent fields must only appear as a field reference for identifiers representing (index) variables describing an agent value, whereas cell fields may only be specified for identifiers representing (index) variables describing a cell value.

3.4 Statements

A statement defines an action to be performed; the process by which a statement achieves its action is called execution of the statement. Cellang programs may contain any number of statements.

```
statement → declare_constant
          → assignment_statement
          → if_statement
          → forall_statement
          → exit_statement
          → agent_statement
```

All of the statements in a Cellang program are performed for each cell, once during each `time` value. Programs whose behavior depends on the relative order of statement execution between cells of the automata are erroneous.

3.4.1 Assignment Statement

An assignment statement permits the value of the variable on the left hand side of the first := to be set or changed.

```
assignment_statement → variable := expression_list
                                  rest_expression

rest_expression      → rest_when
                     → λ

rest_when            → when expression
                            := expression_list
                               rest_when_other
                     → when expression

rest_when_other      → rest_when
                     → otherwise

expression_list      → expression
                            { , expression }

variable             → identifier field_reference
                     → array_name [ ]
                            field_reference
                            array_for_option
                     → array_reference
                            field_reference

array_for_option     → for numeric_literal
                     → λ
```

If the identifier has not previously appeared on the left hand side of an assignment statement, then the identifier is implicitly declared as a variable. The variable is (statically) declared as either a single or multiple cell (or agent) field variable depending upon the value(s) of the expression list and whether or not a field reference was given. This implies that the expression list value(s) of the assignment statement must be either all single field or all multi-field expressions. An expression list should contain multiple expressions only when the the left hand side of the assignment statement is an unindexed array. Furthermore, the variable may contain at most one (1) unindexed array.

If the left hand side of the assignment constitutes the declaration of an array (by being the first lexical occurrence) then a non-empty array for option must be given. If the left hand side does not constitute a declaration then the array for option should be empty. When the array for option specifies a numeric literal, the numeric literal given indicates the number of elements that the array will contain.

The range of values that can be assigned to the field(s) of a variable is determined by the implementation, but must be at least as large as the union of all field ranges given in the cell declaration. Programs which assign the field(s) of variable values outside of its associated range are erroneous, though compilers are not required to generate run-time checks to detect such occurrences.

If a single expression (or a single expression followed by **otherwise**) appears to the right of the := then that expression is assigned to the entity on the left hand side of the :=. If one or more **when** clauses appear then the expression to the right of **when** is evaluated first, if it is non-zero the value of the expression to the left of **when** is assigned and execution of the assignment statement is complete. In the case that the expression to the right of **when** evaluates to zero (0), then successive **when** clauses are tried in order. If all expressions to the right of **when** in all such clauses yield a zero (0) value and if an expression to the left of **otherwise** appears, its value is assigned. If none of the **when** conditions yield a non-zero value and if no **otherwise** appears, then the value is unchanged.

Assigning a value to `cell` causes the value of the current cell to become that value. This is the only way that the value of cells of the automata can be altered. Note, however, that that assignment(s) to `cell` determine the value that the current cell will have beginning with the next `time` value. Thus an assignment to `cell` will NOT alter the value that `cell` yields in an expression, for computations performed during the same `time` value. The variable `cell` is the only variable that expresses this dual nature of present versus future value.

If the value(s) on the right hand side of the assignment are cell values, and if cell fields were declared for the cells of the automata, and if no field reference was specified for a multi-field value, then all of and only the cell fields are assigned. Similarly, if the value(s) on the right hand side of the assignment are agent values, and if agent fields were declared for the cells of the automata, and if no field reference was specified for a multi-field value, then all of and only the agent fields are assigned.

If an array is given on the left hand side of the assignment, then an expression list containing one (1) or more expressions may be given on any of the right hand sides. The right hand sides need not agree in the number of expressions given, though if more than one (1) the number of expressions given in the expression list must be the same as the size of the array. If multiple expressions are given, then each is assigned to the corresponding array entry. If only one (1) expression is given, then *all* elements of the array are assigned that value.

Examples:

```
value   := max_value – 1

cell    := blue when time = 0
        := orange otherwise

maybe := 1 when random = x
        := 2 when random = y

max     := north
max     := south when south > max
max     := east when east > max

neighbor[] for 4
        := [0, 1], [0, –1], [1, 0], [–1, 0]
```

3.4.2 If Statements

An if statement selects for execution one or none of the enclosed statement lists, depending on the value of one or more conditions.

```
if_statement → if expression then
                   statement_list
               { elsif expression then
                   statement_list }
               else_option
               end

else_option  → else statement_list
             → λ
```

For the execution of an if statement, the condition specified after **if**, and any conditions specified after **elsif**, are evaluated in succession (treating a final "**else**" as "**elsif** non-zero **then**"), until one evaluates to a non-zero value or all conditions are evaluated and yield zero (0) values. If a condition evaluates to non-zero, then the corresponding statement list is executed; otherwise none of the statement lists are executed.

Example:

```
if neigbor_count = 4 then
    cell := 0
elsif neigbor_count = 3 then
    cell := 1
    count := count + 1
else
    cell := 0
end
```

3.4.3 Forall Statement

A forall statement is used to iterate through a range of values, usually to examine/change the identifiers associated with an array.

```
forall_statement → forall index_variable
                          statement_list

index_variable   → identifier index_range

index_range      → : range
                 → : agent
                 → λ
```

The index variable is implicitly declared, but is only visible and usable within the statement list of the loop body. If the index range is given, it is either a range or the reserved keyword **agent**. Index variables cannot be assigned.

If the index range is **agent**, then for each iteration through the loop the index variable will hold the value of the next agent associated with this cell. Programs which assume a particular ordering of agent iteration are erroneous. Furthermore, the ordering can be different for each forall statement, even when executed during the same time step. Only programs that declare one or more agent fields in the cell declaration may use this option.

If the index range is a range the index variable will take on the successive values of the range (i.e. sequential execution). If the index range is not given, then the index variable must be used as the index for an array and the size of the index range is inferred from that usage. The index variable can only be used apart from an index for an array when the index range has been given or can be inferred from such a usage. If inferred, all of the inferred sizes of the index range must be equal in size. If the index range was given, then the specified index range must be the same size as all inferred sizes of the index range. If the index range was not given, then its lower bound is taken as 0 and it must be possible to infer the size of the range from an array usage in the body of the loop.

If the index variable is to be used to iterate through the agents associated with the cell, then the index range **agent** must be given as it is not possible to infer this type of index range.

Examples:

```
forall a:agent
    count := count + 1 when a.type = 2
end

forall i:0..3
    forall j
        sum[i] := (neighbor[j] = i +% 1) + sum[i]
    end
end

forall i
    in := in + 1 when neighbor[i].speed[i +% 3]
end
```

3.4.4 Exit Statement

An exit statement may only be used within a forall statement.

```
exit_statement → exit
```

The effect of performing an exit statement is to cause the immediately enclosing loop to terminate, transferring control to the point just after the end of the enclosing forall loop.

Example:

```
forall a:agent
    if a.type = 2 then
        count := count + 1
    else
        error := 1
        exit
    end
end
```

3.4.5 Agent Statement

An agent statement is used to associate an agent with with a cell. Only programs that declare one or more agent fields in the cell declaration may use this statement.

```
agent_statement → agent_value –> associated_cell
                          rest_cell

rest_cell       → rest_cell_when
                → λ

rest_cell_when  → when expression
                      –> associated_cell
                          rest_cell_when
                → when expression
                → otherwise

agent_value     → agent ( expression_list )
                → identifier
                → array_reference

associated_cell → identifier
                → relative_index
```

All of the agents associated with a cell at a particular time step are automatically discarded before the next time step. Thus, if the information contained within an agent is to persist, a new agent containing this information must be associated with the cell. An agent statement associates the agent value with the associated cell during the next time step. If an identifier or array reference is given, then it must specify a single agent value, otherwise a new agent value is created. If a new agent is being created, the number of expressions in the expression list must match exactly the number of agent fields specified in the cell declaration. If an identifier is used for the associated cell, it must be the predefined identifier `cell`.

The associated cell to the right of the –> gets the agent value that was given on the left as one of its associated agents during the next time step. When one or more **when** clauses appear then the expression to the right of **when** is evaluated first, if it is non-zero the agent value is associated with the associated cell to the left of **when**. In the case that the expression to the right of **when** evaluates to zero (0), then successive **when** clauses are tried in order. If all expressions to the right of **when** in all such clauses yield a zero (0) value and if an associated cell to the left of **otherwise** appears, that is the cell with which the agent value is associated. If none of the **when** conditions yield a non-zero value and if no **otherwise** appears, then no the agent value is not associated with any cell.

Examples:

```
agent(3, –2) –> [1, 1]

agent(9, 2)  –> [–1, –1] when found
             –> cell otherwise

a[3] –> [–1, 3]

c –> [0, 1] when c.direction = 1
  –> [1, 0] when c.direction = 2
  –> [0, –1] when c.direction = 3
  –> [–1, 0] otherwise
```

4 Input and Output

The form of the input and output for Cellang programs is identical.

```
i/o_form    → { time_block }

time_block  → time { cell_value }

cell_value  → [ integer { , integer } ] = value_list

value_list  → field_value { , field_value }

field_value → integer
            → λ
```

time	$\rightarrow$ numeric_literal
integer	$\rightarrow$ sign_option numeric_literal

The time must be a non-negative number, which should be strictly larger than any previous time that has appeared in the input (for input) or output (for output). The cell value is an absolute reference to a cell within the cells of the automata[1] with the field values of the indicated cell taken as the respective numeric literal values given by the value list. These are associated with the fields in the same order as the field identifiers were given in the cell declaration. The number of numeric literals in the value list must be no greater than the number of fields given in the cell declaration.

As input, the cell values appearing after a time (and before the next time) give the new field values for all of the indicated cells; beginning at the specified time. The field values of all cells at time zero (0) is zero (0) unless otherwise specified. No cells other than those specified by a cell value have their values altered and only those fields specified (indicated by supplying an integer value) are assigned new values. If agent fields were specified as part of the cell declaration, then any field values given beyond the number of cell fields will be taken as agent field values for agents to be created and associated with that cell. As many agents as necessary will be created and associated as are necessary to utilize all of the field values given.

In the output, cell values appearing after a specific time indicate the field value of cells at the beginning of that time. The field value(s) given are the field value(s) of the cell at that time. As with the input, the initial value of all cells of the automata zero (0) unless otherwise indicated. Only those value(s) which have been altered since the last output are required to be output. Thus if no fields have been altered, no field values are required to be output (although doing so is permitted). If agents are associated with the cell, the set of field values for each agent will be output following the output of the cell fields. As with the cell fields, the agent fields will be output in the order declared.

It is important to note that the input of agent values is treated differently from output. Specifically, any agent field values given on input, including zero (0), will cause the creation of a new agent for the associated cell. On output, however, only the agent field values of existing agents will be output. Thus, when an agent is no longer associated with a cell *only* its absence from the output is an indicator of this change. In addition, programs which assume a particular ordering of agents for either input or output are erroneous. The relative ordering of agent fields is determined by the cell declaration, but actual agents may be given in any order.

Unlike the syntax for programs, the syntax of the input (output) must also be given (appear) in a line oriented fashion. In particular the time and cell value information must each reside on a single line of input (output) with no intervening blank lines.[2]

[1]The absolute index values for a dimension always begin at zero (0) and extend in the positive direction.

[2]This syntactic restriction makes the construction of filter programs easier.