Cinc
User Guide

Draft

I Stephenson

Adaptive Systems Engineering Group
Department of Electronics
University of York
York Y01 5DD

email	: ian@ohm.york.ac.uk
Tel	: (+44) 904 432381
Fax	: (+44) 904 432335

March 1992

This work is supported by the Royal Signals and Radar Establishment, Malvern, Worcs and the Science and Engineering Research Council.

Chapter 1

Introduction

This document provides an introduction to the Creatures model of processing, and to the use to the Cinc simulator. The theory and rational of the processing model are not discussed, as this may be found else where. It is the aim here to provided the practical details which are required to effectively use the Cinc toolset.

We will first discuss the operation of the simulator, before considering how position and rule files may be produced.

Chapter 2

Cinc

Creature programs may be run under NeXTStep using the Cinc simulator. The main window in Cinc is shown in figure 2.1. This contains most of the features of Cinc that are required to develop and examine simulations using the creatures model. The key elements are:

1. The number of creatures in the current generation.
2. The creature view which displays the active creatures.
3. The number of generations since the system was last reset.
4. The Scale slider, which allows the user to zoom in/out on the creature view.
5. The Step button, advances the simulation one time step.
6. The rule button reloads the current rule.
7. The Delay controls set the speed at which the simulation runs
8. The Position button resets the current position.
9. The Go/Stop button which sets the simulation running.

In addition an inspector panel allows the number of creatures to be examined by type, on a local and global basis. This is shown in figure 2.2.

1. Selects between local and Global selection.

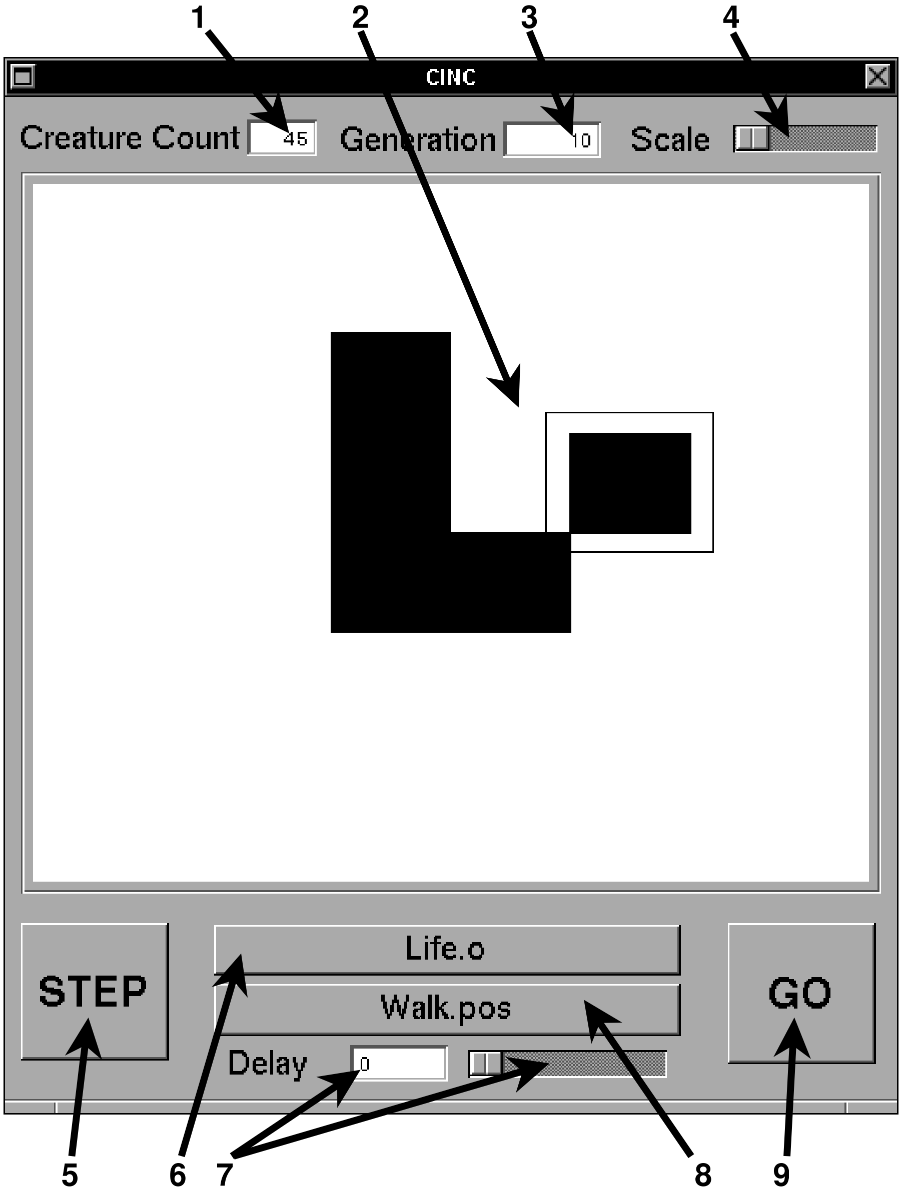


Figure 2.1: The Main Cinc display

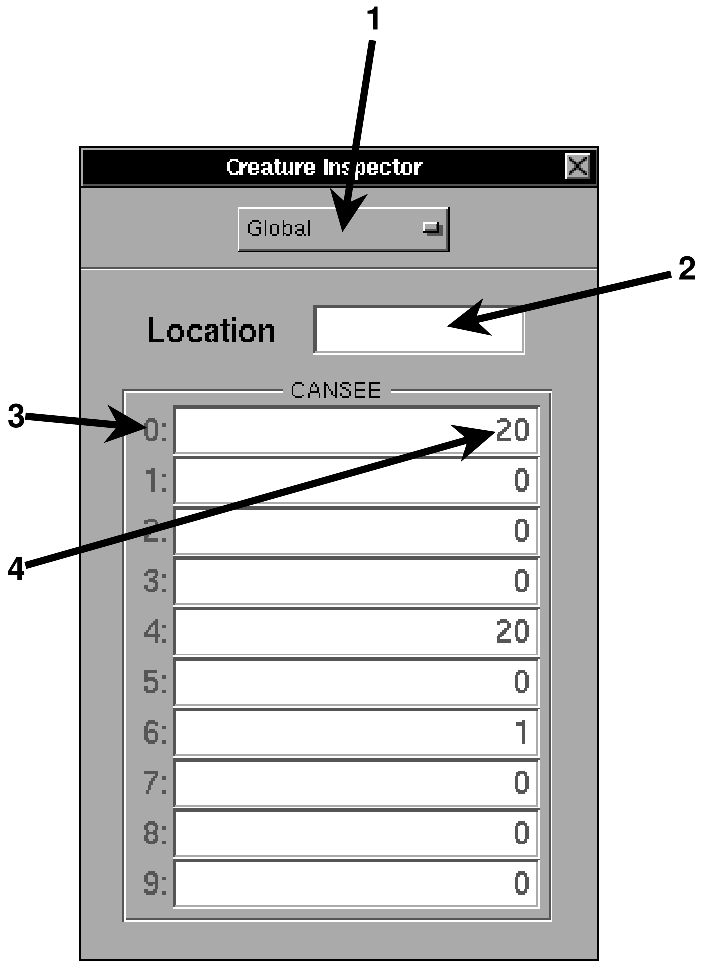


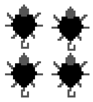
Figure 2.2: The Cinc Inspector Panel

2. Displays the selected location when in local mode.
3. Creature type number.
4. The number of creatures.

Finally there is the standard NeXT menu which allows files to be loaded and saved, the display to be printed, and other miscellaneous operations.

2.1 A typical Cinc session

A session may be started by double clicking the application icon , or opening a position file, which should appear in the workspace as:



Once Cinc is running the user must load a rule file (and a position file, if one has not been opened). This is done by selecting The appropriate option from the document sub-menu, and specifying the file name within the a standard open panel. When both a rule, and a position file have been loaded, then the simulation may started. This is done by pressing the "Go" button. The simulation may be slowed down, for more detailed inspection with the "delay" slider. Further control may be obtained by stopping the run, and single stepping with the "Step" button.

The inspector may be instigated by selecting it from the menu[1]. This allows the creatures to examined by type, either locally or globally. In global mode all creatures are displayed in the panel. In local mode, a box will appear on the creature view indicating which location is currently under inspection. This may be moved by clicking in the creature view. Only those creatures in the selected location will be counted.

The top line of the main window provides a count of the number of creatures, and the number of generations since the system was last reset. It also includes the scale slider which allows the creatures view to be zoomed in and out over the creature space.

[1] Use of the inspector will slow the simulator down, and hence it should only be brought up when required

Clicking on the Rule button will reload the rule from the current rule file. This is used during development — the user will typically be developing a rule, and constantly be reloading a modified version into the simulator for testing. Loading a rule will also reset the position.

The position may be explicitly reloaded by clicking on the rule button.

The user will typically start the system, load a position and rule set, run the simulation for a number of steps, examining the result using the inspector. The rule will then be modified, and recompiled (outside the simulator), and reloaded for further examination.

In addition the user may print the contents of the creature view, using the standard print panel. The current position may also be saved to a file, though it should be noted that this does not necessarily allow a simulation to be restarted, as the internal state of the creatures is not stored.

Chapter 3

Position Files

Files with a ".pos" extension are position files. These contain an initial set of creatures which may be loaded into the simulator. They specify each creature in the format

```
t:0 x:0 y:0
```

with one creature per line. The number following the t specifies the type of creature. The x and y values specify the location at which it should be placed. 0,0 is the center of the creature view.

Chapter 4

Rules Files

Rule files will generally be written in the custom creature definition language "Jam". This allows rules to be laid out in a straighforward fashion. These jam files may then be compiled into Objective C, and then into object files which may be loaded into Cinc.

Rules may also be written directly in objective C. This allows the programmer more control over how a creature interacts with the simulator. However this should be unnecessary for most rules. A mechanism is provided by which these modifications (in particular adjustments to the way creatures are displayed) may be included in Jam rules, abstracting the programmer from low level implementation details.

It is not the intention of this document to give full specifications of the programming language, rather an overview of the techniques is presented using a number of examples. We will first look at the Jam language, and how these may be compiled. We will then consider the use of raw Objective C.

4.1 Writing Rules in Jam

The Jam language may has been designed for writing creature rules. While lacking certain features that would be required in a conventional programming language (such as loops), it is particularly suited to the type of expressions required in the definition of rules. Two examples are presented : an ideal gas, and the game of life.

4.1.1 An ideal Gas

Figure 4.1 shows a simple model of an ideal gas, written in Jam. The definition begins with a specification of the neighborhood. This model uses the Von Neumann neighborhood — North, South, East and West (moore, and hex neighborhoods are also supported). These define the locations into which a creature may move, relative to its current position. We next define the creature types which exist in our model. This simple model does not hold any internal states for the creatures. These would be defined and initialized at this point (as we shall see in a later, more complex example).

The next stage of the definition is the specification of the rule itself. This is written as a number of tests and actions (which in turn may contain a list of tests). The Gas rule first tests if the creature is a box particle or an ice particle. If so, the creature will perform the CENTER action. This is a movement operation defined in the neighborhood declaration, and is therefore a terminal operation. The rest of the rule is ignored once a creature performs a terminal action. Only creatures which are not of the box or ice type will consider the next test. The rest of the rule consists of four similar sections, one for each of the remaining types. Upon the selection of one of these actions, a further set of tests is performed. Considering the creature type `np`, if it encounters a `sp` it will become an `ep` and start moving east. Similarly on encountering a `box` it will "bounce" off, becoming a `sp`, and start moving south. When an `ice` creature is encountered, the `np` is turned to ice, and moves back into its previous location, where it will be frozen, increasing the size of the ice crystal which grows across the screen.

4.1.2 The Game of Life

The code to produce the game of Life (figure 4.2) is somewhat more complex than the previous example, and introduces a number of new concepts. The Life rule uses the Moore neighborhood, as this allows access to the diagonal neighbors, which are not part of the Von Neummann neighborhood. To build the game of life within the creatures paradigm it is necessary to think of the system in terms of spores which spread from active cells into adjacent locations. To implement this we require each creature to be aware of its age. This is introduced by the VARS construct. The INIT statement is executed by every creature when it is created, and allows any variables to be initialized.

```
NEIGHBORHOOD:vn;

TYPES: np,sp,ep,wp,box,ice;

RULE:{
     iam(box) | iam(ice):  CENTER;
     iam(np):              {
                           cansee(sp)      : {become(ep);EAST;}
                           cansee(box)     : {become(sp);SOUTH;}
                           cansee(ice)     : {become(ice);SOUTH;}
                           true            : {NORTH;}
                           }
     iam(sp):              {
                           cansee(np)      : {become(wp);WEST;}
                           cansee(box)     : {become(np);NORTH;}
                           cansee(ice)     : {become(ice);NORTH;}
                           true            : {SOUTH;}
                           }
     iam(ep):              {
                           cansee(sp)      : {become(np);NORTH;}
                           cansee(box)     : {become(wp);WEST;}
                           cansee(ice)     : {become(ice);WEST;}
                           true            : {EAST;}
                           }
     iam(wp):              {
                           cansee(ep)      : {become(sp);SOUTH;}
                           cansee(box)     : {become(ep);EAST;}
                           cansee(ice)     : {become(ice);EAST;}
                           true            : {WEST;}
                           }
     }
```

Figure 4.1: An Ideal Gas

```
NEIGHBORHOOD:moore;

TYPES:cp,ep,nep,np,nwp,wp,swp,sp,sep;

VARS:int age;

INIT:{
     age=0;
     }

RULE: {
      age==0:   {
                true      : age=1;
                iam(cp)   : CENTER;
                iam(ep)   : EAST;
                iam(nep)  : NORTHEAST;
                iam(np)   : NORTH;
                iam(nwp)  : NORTHWEST;
                iam(wp)   : WEST;
                iam(swp)  : SOUTHWEST;
                iam(sp)   : SOUTH;
                iam(sep)  : SOUTHEAST;
                }

      !:        {
                iam(ep)  & cansee(cp)       : DIE;
                iam(nep) & cansee(cp...ep)  : DIE;
                iam(np)  & cansee(cp...nep) : DIE;
                iam(nwp) & cansee(cp...np)  : DIE;
                iam(wp)  & cansee(cp...nwp) : DIE;
                iam(swp) & cansee(cp...wp)  : DIE;
                iam(sp)  & cansee(cp...swp) : DIE;
                iam(sep) & cansee(cp...sp)  : DIE;

                (cansee(ep...sep)==2 & iam(cp))|cansee(ep...sep)==3 : birth(cp...sep)

                true : DIE;
                }
      }
```


Figure 4.2: The Game of Life

The first part of the RULE definition is activated if the creature has an age of zero. There are nine creature types which each move in a different direction, spreading the spores of a living cell. The second half of the rule will be effective on the second step the creature makes. A master for each potentially live location is elected. This decides whether the cell should be alive or not. It so, then a number of creatures are produced which will proceed to spread in the following phase.

This implements the game of live at half the clock speed of the creatures model.

This rule makes use of the range operator, which allows ranges of creatures to be easily examined, as one. It may be applied to the cansee, iam, and birth functions. Cansee will return the number of creatures within the range. Iam will return true for if the type of the current creature is within the range. Birth will produce one of each type of creature within the range.

There are two additional constructs which have not been used within either of these models. The first is a form of replicator which may only be used with the birth function. The operation `birth(type)(400)` would produce 400 creatures of type `type`. Both this and the range operator are statically bound at compile time. Variables may not be used to specifically ranges or replication numbers.

The final structure supported by the language is the `USE: file,file;` directive, which may be inserted immediately after the `VARS` declaration. This is used to include system specific modifications to creature behavior, such as drawing the creature in a different fashion. Typically, a selection of prewritten USE files will be supplied with a simulator. Cinc provides "color" which draws creatures in colour, and "hex" which approximates the drawing of creatures onto a hex lattice for use with the hex neighborhood). Creating new USE files requires a deeper understanding of how the Cinc simulator operates.

4.2 Using the pancake compiler

The pancake compiler will compile files in the jam language into objective C, for use with Cinc. Other compilers have been written to support other simulators. Pancake operates as a filter, reading from standard input, and writing to standard out. To create a rule file, the makefile provided should

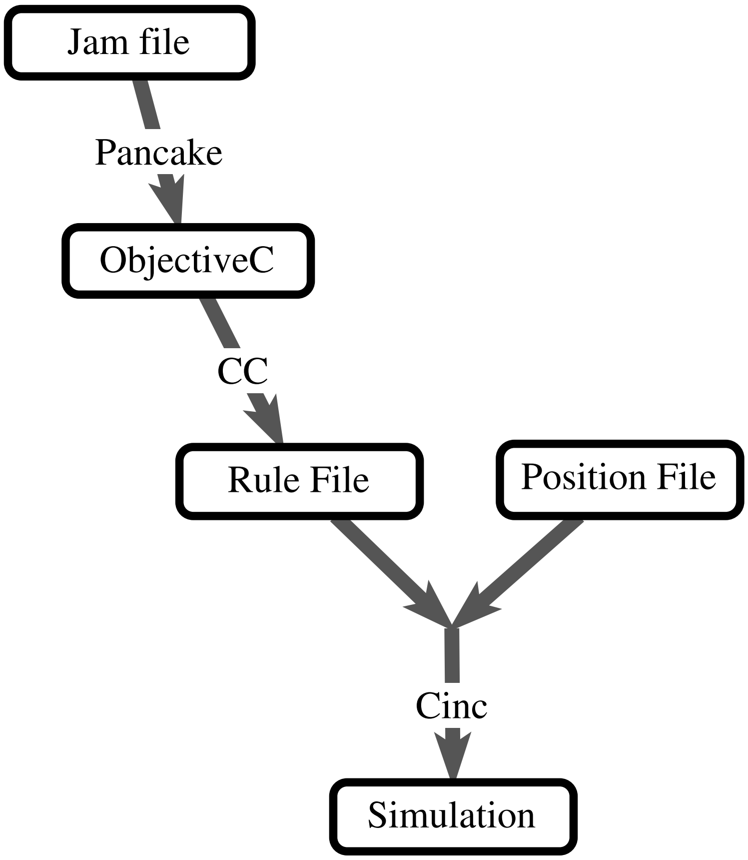


Figure 4.3: The Compile Cycle

be used, as a number of special include directories are required. Typing make creates a `.m` file from the `.jam` file, and compiles this into an object (`.o`) file which may be loaded into Cinc. This compile cycle is shown in figure 4.3.

4.3 Writing creatures in Objective C

Writing rules in Objective C is strongly discouraged, as it will prevent the code being run on simulators other than Cinc. However it is useful to examine how such a rule may be written, as it gives some insight into how Cinc operates. This low level understanding is essential for writing USE files.

Rules files are Objective C object modules which define a sub Class of the generic Creature class. However an understanding of Objective C is not required, though knowledge of standard C would be useful. It is quite possible to build complex rules from only a very limited subset of the C language. Most of the work is done transparently in the Creature class, and the supporting macro package.

The ideal gas rule previously examined, is presented again in figure 4.4. The model is now coded in Objective C. It should be noted that while this similar to the output of pancake, the code shown here is hand written.

The code overwrites the default `step: n: g:` method.The exact details of this operation are not essential to the writing of rules, as macros are defined to hide many of the details. These are included in `macro.h`. The neighborhood definitions are in `vn.h`. We assign an index to each of the creature type. What follows is almost identical to the Jam definition.

A further understanding may be obtained by examining the included header files, which define the macro set.

4.4 Creating USE files

Figure 4.5 shows the USE file which draw creatures in colour. The code overwrites the monochrome display method of the creature class, with a new colour drawing routine. The routine uses two instance variables (which we have otherwise chosen not to mention, as they are not part of the programming model). x and y which hold the position of the creature. It is convention that each creature be drawn in the square (x,y) (x+1,y) (x+1,y+1) (x,y+1) (x,y). This is not essential to the operation of the simulator, but any other mapping will produce undesirable behavior if the inspector is used.

It is also possible to overwrite other methods of the creature class, such as the `near` methods. This would (for example) allow higher dimension models to be build.

```
#import "Creature.h"
@implementation Gas:Creature
{
}

- step:nextgeneration n:(int *)neighbours g:graveyard
  {
  #include "macro.h"
  #include "vn.h"

  #define NP  0
  #define SP  1
  #define EP  2
  #define WP  3
  #define BOX 4
  #define ICE 5

  switch (type )
  {
    case BOX:
    case ICE:
      CENTER;

    case NP:{
      if(CANSEE(SP)!=0)   {BECOME(EP);EAST;}
      if(CANSEE(BOX)!=0)  {BECOME(SP);SOUTH;}
      if(CANSEE(ICE)!=0)  {BECOME(ICE);SOUTH;}
      NORTH;
      }
    case EP:{
      if(CANSEE(WP)!=0)   {BECOME(NP);NORTH;}
      if(CANSEE(BOX)!=0)  {BECOME(WP);WEST;}
      if(CANSEE(ICE)!=0)  {BECOME(ICE);WEST;}
      EAST;
      }
    case SP:{
      if(CANSEE(NP)!=0)   {BECOME(WP);WEST;}
      if(CANSEE(BOX)!=0)  {BECOME(NP);NORTH;}
      if(CANSEE(ICE)!=0)  {BECOME(ICE);NORTH; }
      SOUTH;
      }
    case WP:{
      if(CANSEE(EP)!=0)   {BECOME(SP);SOUTH;}
      if(CANSEE(BOX)!=0)  {BECOME(EP);EAST;}
      if(CANSEE(ICE)!=0)  {BECOME(ICE);EAST;}
      WEST;
      }
      default:
        {ALERT("Unknown Creature");DIE; }
    }
  }
@end
```


Figure 4.4: The Gas model in ObjC

```
#import <appkit/graphics.h>
#import <dpsclient/psops.h>
#import <appkit/color.h>

-display
{ /* over write display method to use colour */
    NXSetColor(NXConvertHSBToColor(((float)type)/10.0, 1, 1));

    PSrectfill(x,y,1,1);
return self;
}
```

Figure 4.5: The USE file “color”

Appendix A

Jam Grammer

⟨input ⟩ :
⟨neighbordeclaration ⟩ ⟨typedeclaration ⟩ ⟨vardeclaration ⟩ ⟨usedeclaration ⟩ ⟨initdeclaration ⟩ ⟨ruledeclaration ⟩ ■

neighbordeclaration :
NEIGHBORHOOD: ⟨variable ⟩ ; ■

typedeclaration :
TYPES: ⟨typelist ⟩ ; ■

⟨typelist ⟩ :
⟨typelist ⟩ , ⟨variable ⟩
| ⟨variable ⟩ ■

⟨vardeclaration ⟩ :
VARS: ⟨varlist ⟩ ;
| ■

⟨varlist ⟩ :
⟨varlist ⟩ , ⟨vartype ⟩ ⟨variable ⟩
| ⟨vartype ⟩ ⟨variable ⟩ ■

⟨usedeclaration ⟩ :
USE: ⟨uselist ⟩ ;
| ■

$\langle$uselist$\rangle$:
  $\langle$uselist$\rangle$, $\langle$variable$\rangle$
  | $\langle$variable$\rangle$ ■

$\langle$initdeclaration$\rangle$:
  INIT: $\langle$statement$\rangle$
  | ■

$\langle$ruledeclaration$\rangle$:
  RULE: action ■

$\langle$tplist$\rangle$:
  $\langle$expression$\rangle$ **:** $\langle$action$\rangle$ $\langle$tplist$\rangle$
  | $\langle$expression$\rangle$ **:** $\langle$action$\rangle$ **!:** $\langle$action$\rangle$ $\langle$tplist$\rangle$ ■

$\langle$action$\rangle$:
  { $\langle$tplist$\rangle$ }
  | $\langle$statement$\rangle$ ■

$\langle$statement$\rangle$:
  $\langle$expression$\rangle$;
  | { $\langle$statementlist$\rangle$ }
  | ; ■

$\langle$statementlist$\rangle$:
  $\langle$statement$\rangle$ $\langle$statementlist$\rangle$
  | $\langle$statement$\rangle$ ■

$\langle$expression$\rangle$:
  $\langle$variable$\rangle$
  | $\langle$constant$\rangle$
  | $\langle$function$\rangle$ **(** $\langle$expression$\rangle$ **)**
  | $\langle$function$\rangle$ **(** $\langle$variable$\rangle$ **...** $\langle$variable$\rangle$ **)**
  | $\langle$function$\rangle$ **(** $\langle$expression$\rangle$ **)** **(** $\langle$number$\rangle$ **)**
  | $\langle$movement$\rangle$
  | $\langle$fnumber$\rangle$
  | $\langle$number$\rangle$
  | **(** $\langle$expression$\rangle$ **)**
  | $\langle$expression$\rangle$ ***** $\langle$expression$\rangle$

| ⟨expression ⟩ / ⟨expression ⟩
| ⟨expression ⟩ % ⟨expression ⟩
| ⟨expression ⟩ + ⟨expression ⟩
| ⟨expression ⟩ - ⟨expression ⟩
| ⟨expression ⟩ & ⟨expression ⟩
| ⟨expression ⟩ — ⟨expression ⟩
| ⟨expression ⟩ == ⟨expression ⟩
| ⟨expression ⟩ > ⟨expression ⟩
| ⟨expression ⟩ < ⟨expression ⟩
| ⟨variable ⟩ = ⟨expression ⟩ ■

⟨fnumber ⟩ :
⟨number ⟩ . ⟨number ⟩ ■

⟨number ⟩ :
⟨digit ⟩
| ⟨number ⟩ ⟨digit ⟩ ■

⟨digit ⟩ :
0 . . . 9 ■

⟨constant ⟩ :
random
| **type**
| **true** ■

⟨function ⟩ :
cansee
| **alert**
| **birth**
| **become**
| **iam** ■

⟨movement] :
A . . . Z ⟨movement ⟩
| ⟨movement ⟩ ■

⟨variable ⟩ :
a . . . z ⟨variable ⟩
| ⟨variable ⟩ ■

$\langle$vartype $\rangle$:
int
| **float** ■

Bibliography

[1] M Gardner. (1971) *Wheels, Life, and other mathematical amusements.* Scientific American.

[2] B. Hasslacher, U. Frisch, Y. Pomeau. (1986) *Lattice gas automata for the Navier-Stokes equation.* Physics Review Letters, 56.

[3] P. Hogeweg. (1984) *Heterarchical Modeling* Encyclopedia of Systems and Control. Oxford: Pergamon Press

[4] I.Stephenson (1991) *Creature Processing: A simulation environment for Artificial Life* Technical Report ASEG92.02, Department of Electronics, University of York

[5] I.Stephenson (1992) *Creature Processing: An Alternative Cellular Architecture* Technical Report ASEG92.04, Department of Electronics, University of York