# ASCII Art with Quadtree Grammars

Leon Rische

*[2020-01-23 Thu 15:59]*

## Contents



## 1 Source Code

### 1.1 Text Canvas

All data is stored in a long string. This string contains one character for each "pixel" of the image plus a newline after each row.

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (defclass text-canvas () ((data :initarg :data) (width :initarg :width) (height :initarg :height)))

(defun make-text-canvas (width height optional background) "Create a text canvas of size (WIDTH, HEIGHT), optionally using BACKGROUND as background character. If BACKGROUND is nil, spaces are used." (let ((data (make-string (* height (1+ width)) (or background ?)))) ;; Add newlines (dotimes (y height) (setf (aref data (1- (* (1+ y) (1+ width)))) ?)) (make-instance 'text-canvas :width width :height height :data data)))

(defmethod set-pixel ((canvas text-canvas) x y v) "Set the pixel at position (x, y) to value V." (with-slots (width height data) canvas (setf (aref data (+ x (* (1+ width) y))) v)))

(defmethod set-rect ((canvas text-canvas) x y w h v) "Set the rectangle at position (x, y) with size (w, h) to value V." (loop for $x_from x below (+ x w) do (loop for $y_from y below (+ y h) d
$pixel canvas x_{yv}$))))

### 1.2 Quadtree Grammars

Now we can write a recursive function that expands a "quadtree grammar" to generate the contents of a text canvas.

I'll explain how this works in the next section.

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (defun cfg–expand (rules rule canvas x y size) (if (characterp rule) (set-rect canvas x y size size rule) (let ((half (floor size 2))) (destructuring-bind (rhs fallback) (alist-get rule rules) (if (= size 1) (set-rect canvas x y size size fallback) (destructuring-bind (a b c d) rhs (cfg–expand rules a canvas x y half) (cfg–expand rules b canvas (+ x half) y half) (cfg–expand rules c canvas x (+ y half) half) (cfg–expand rules d canvas (+ x half) (+ y half) half)))))))

(defun cfg-expand (size start grammar) (let* ((canvas (make-text-canvas size size))) (cfg–expand grammar start canvas 0 0 size) (slot-value canvas 'data)))

## 2 Introduction

A **quadtree** is a tree where each node has exactly four children. It can be interpreted / rendered e.g. as a (ASCII art) 2D image.

To do so, the image is split up into four **quadrants** of equal size (top left, top right, bottom left, bottom right) either containing another quadtree or a leave node.

Rules have the form `(name . (rhs fallback))`.

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (cfg-expand 16 'a '((a . ((?A ?B ?C ?D) ?A))))

```
AAAAAAAABBBBBBBB
AAAAAAAABBBBBBBB
AAAAAAAABBBBBBBB
AAAAAAAABBBBBBBB
AAAAAAAABBBBBBBB
AAAAAAAABBBBBBBB
AAAAAAAABBBBBBBB
AAAAAAAABBBBBBBB
CCCCCCCCDDDDDDDD
```

```
CCCCCCCCDDDDDDDD
CCCCCCCCDDDDDDDD
CCCCCCCCDDDDDDDD
CCCCCCCCDDDDDDDD
CCCCCCCCDDDDDDDD
CCCCCCCCDDDDDDDD
CCCCCCCCDDDDDDDD
```

`rhs` stands for **right hand side**. It is a list of four **terminals** or **non-terminals**, one for each quadrant of the quadtree.

Terminals are characters (`?A`, `?B`, ... in EmacsLisp). If a terminal is encountered during expansion, the whole region of the quadtree is filled with this character.

If a rule is expanded on a region of size 1, it is filled with the `fallback` character.

If the RHS of the rule being expanded contains a non-terminal, this is interpreted as a reference to another rule that is used to generate the contents of the quadrant.

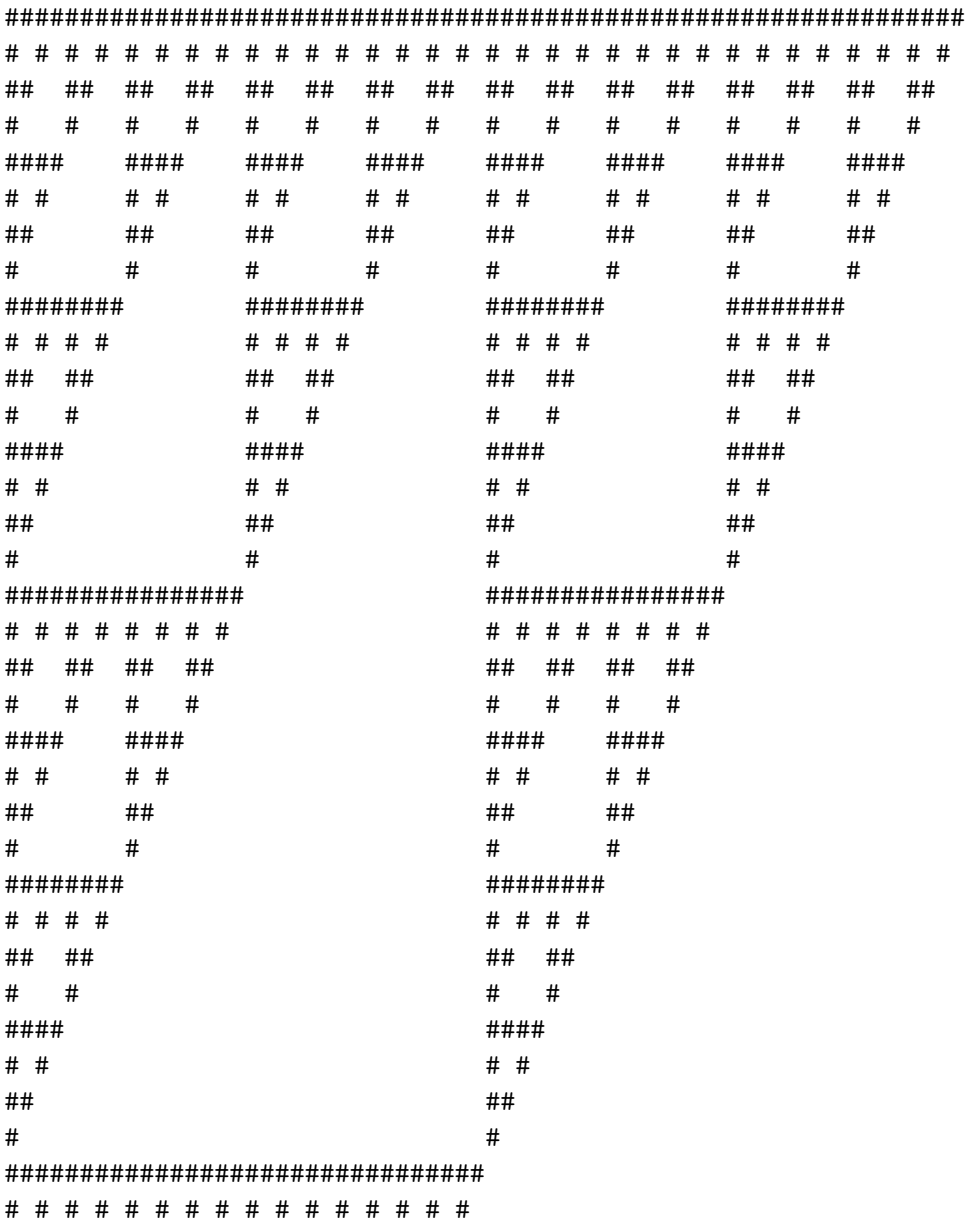
language=Lisp,label= ,caption= ,captionpos=b,numbers=none (cfg-expand 16 'a '((a . ((?A ?B ?C a) ?A))))

```
AAAAAAAABBBBBBBB
AAAAAAAABBBBBBBB
AAAAAAAABBBBBBBB
AAAAAAAABBBBBBBB
AAAAAAAABBBBBBBB
AAAAAAAABBBBBBBB
AAAAAAAABBBBBBBB
AAAAAAAABBBBBBBB
CCCCCCCCAAAABBBB
CCCCCCCCAAAABBBB
CCCCCCCCAAAABBBB
CCCCCCCCAAAABBBB
CCCCCCCCCCCCAABB
CCCCCCCCCCCCAABB
CCCCCCCCCCCCCCAB
CCCCCCCCCCCCCCCA
```

# 3 Examples

## 3.1 Sierpinski Triangle

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (cfg-expand 64 'a '((a . ((a a a ?) ?))))

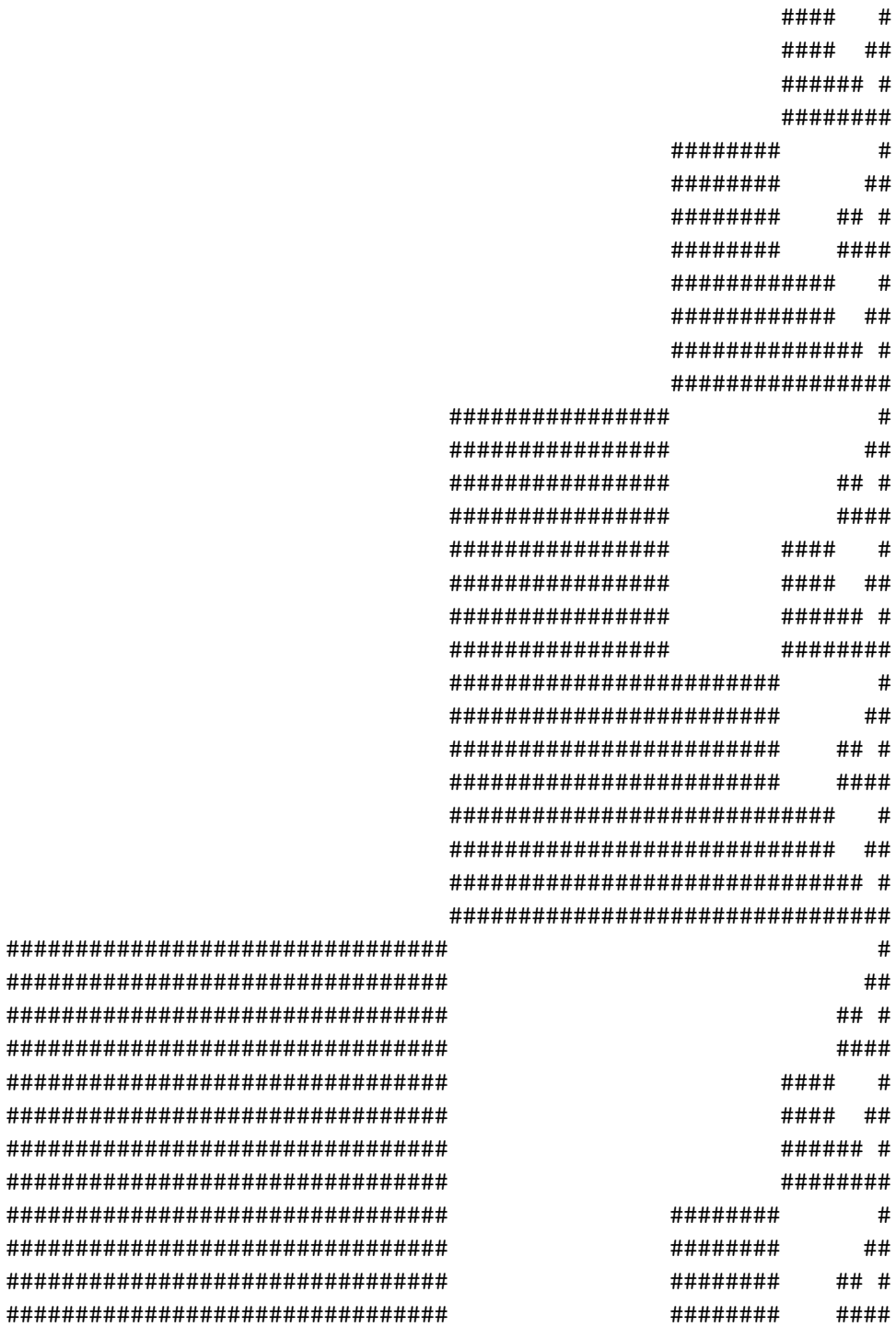


```
##  ##  ##  ##  ##  ##  ##  ##
#   #   #   #   #   #   #   #
####    ####    ####    ####
# #     # #     # #     # #
##      ##      ##      ##
#       #       #       #
########        ########
# # # #         # # # #
##  ##          ##  ##
#   #           #   #
####            ####
# #             # #
##              ##
#               #
################
# # # # # # # #
##  ##  ##  ##
#   #   #   #
####    ####
# #     # #
##      ##
#       #
########
# # # #
##  ##
#   #
####
# #
##
#
```

### 3.2 Example 2

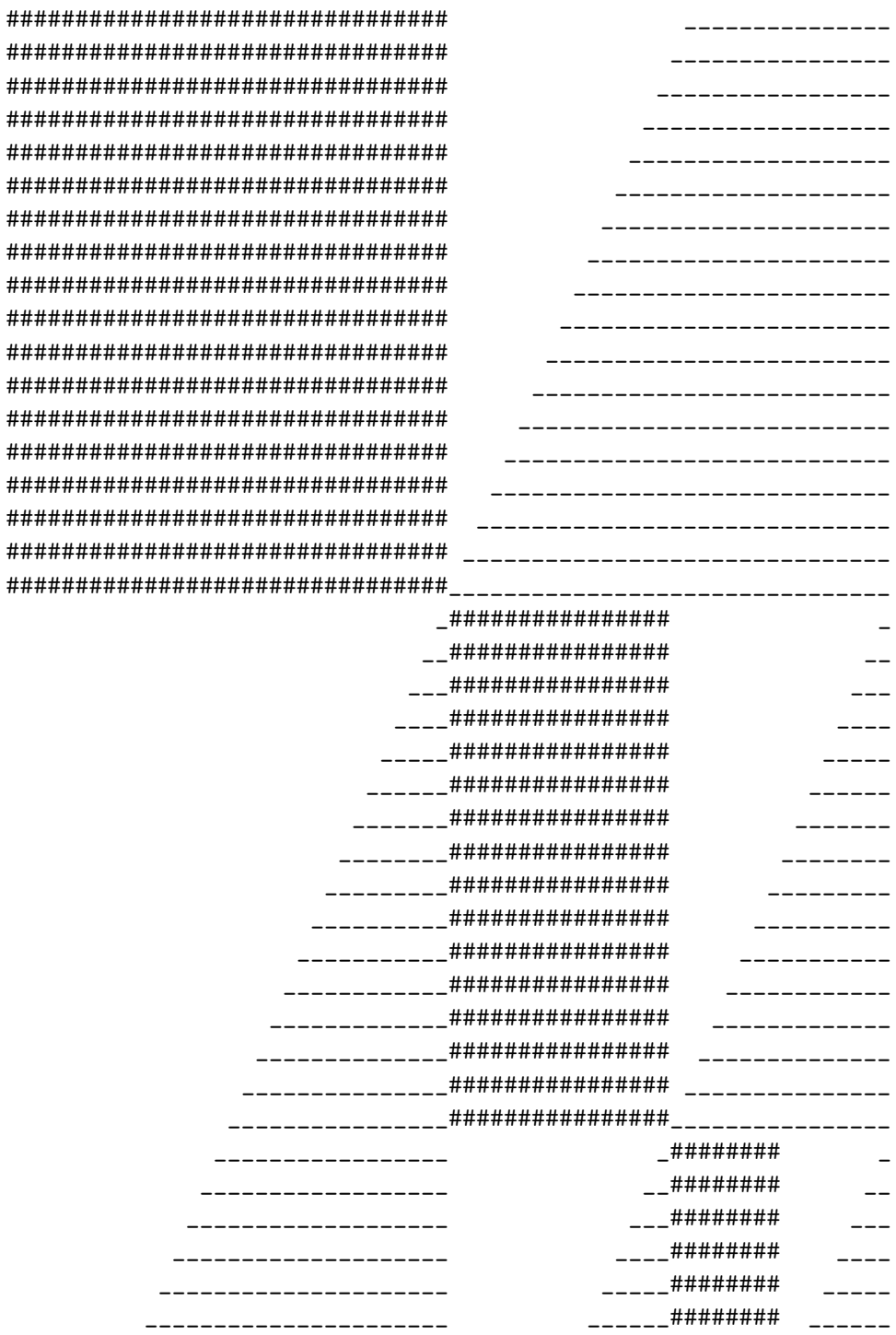
language=Lisp,label= ,caption= ,captionpos=b,numbers=none (cfg-expand 64 'a '((a . ((?a ? a) ?))))

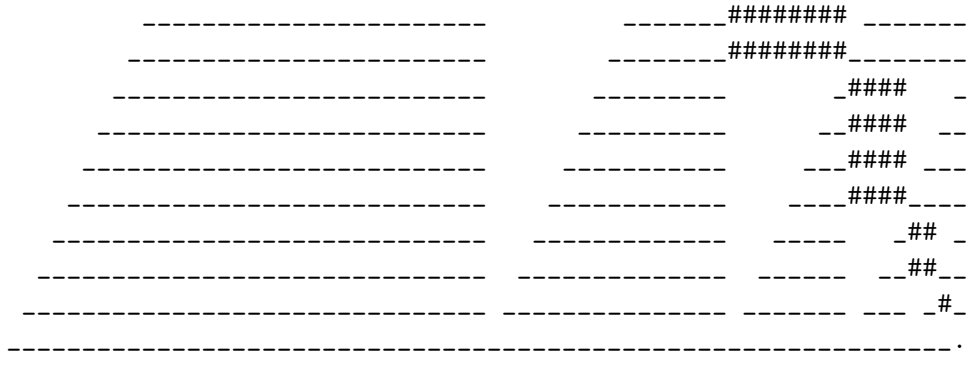
```
                                                                #
                                                               ##
                                                              ## #
                                                              ####
```



### 3.3 Example 3

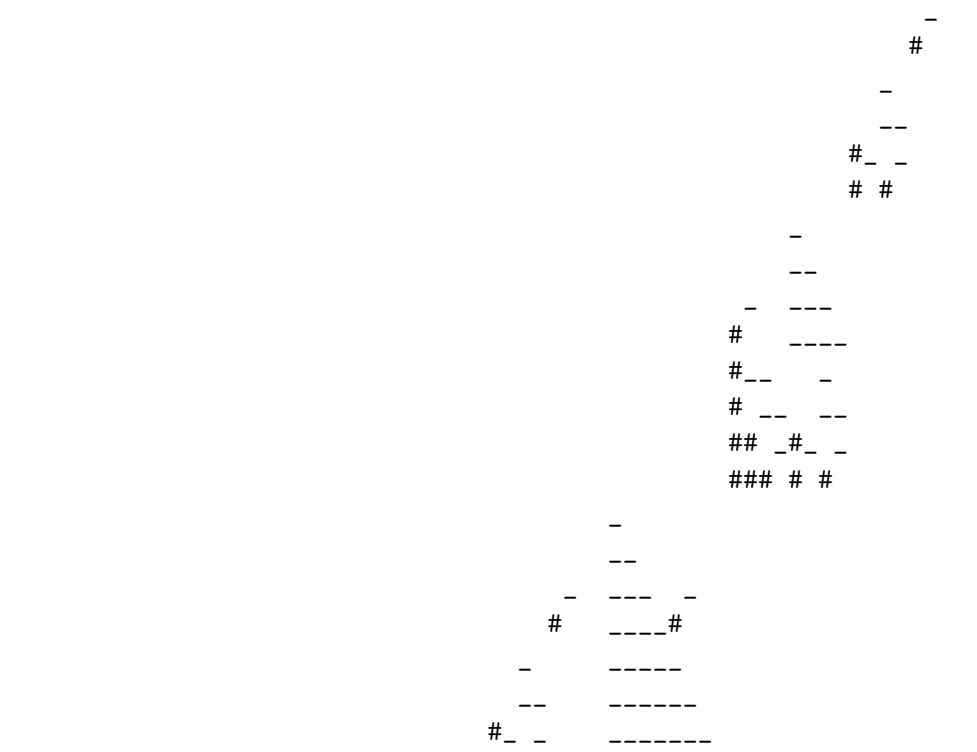
language=Lisp,label= ,caption= ,captionpos=b,numbers=none (cfg-expand 64 'a '((a . ((b c c a) ?.)) (b . ((? b b ?) ?)) (c . ((?c c ?)?)))))

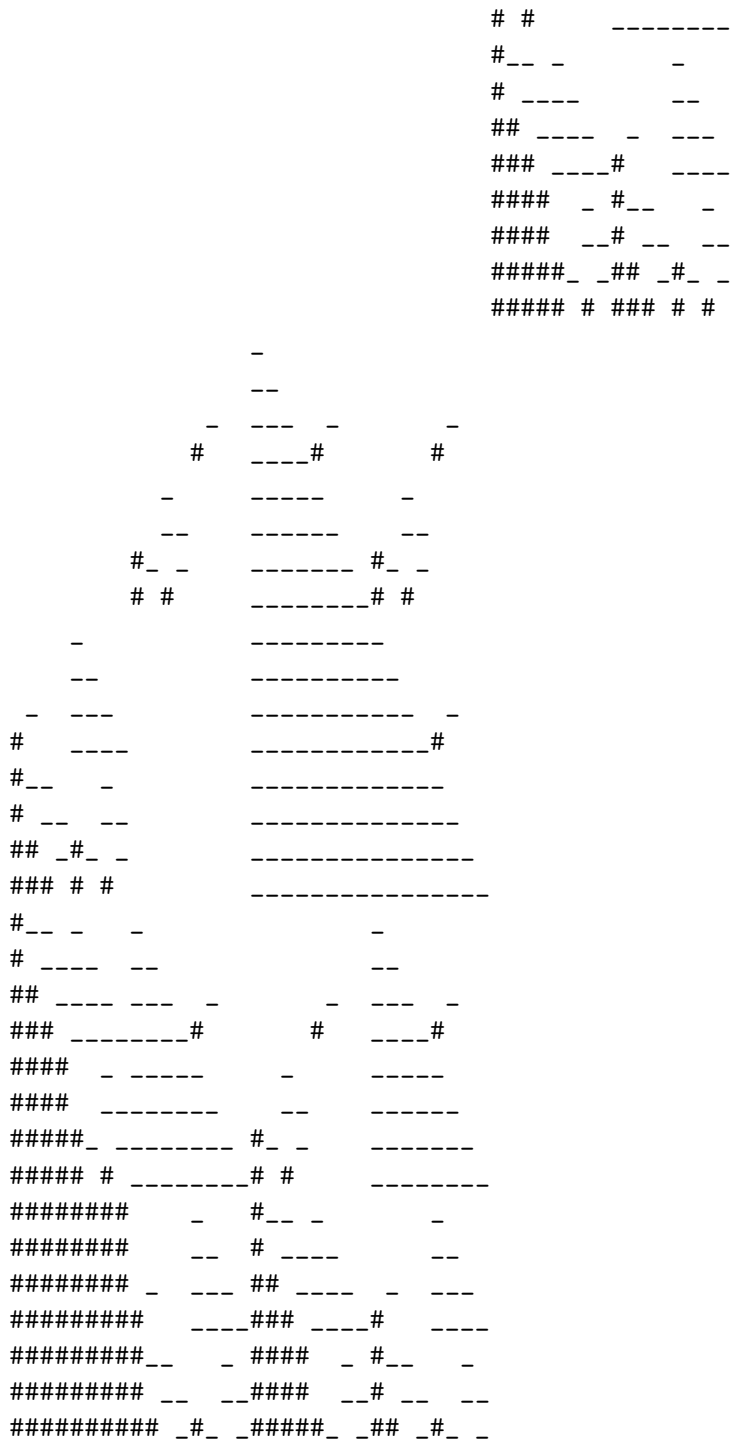




### 3.4 Example 4

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (cfg-expand 64 'a '((a . ((?a c ?) ?)) (b . ((b a ?_ b) ?_)) (c . ((a b d c) ?)) (d . ((d b ? c) ?))))

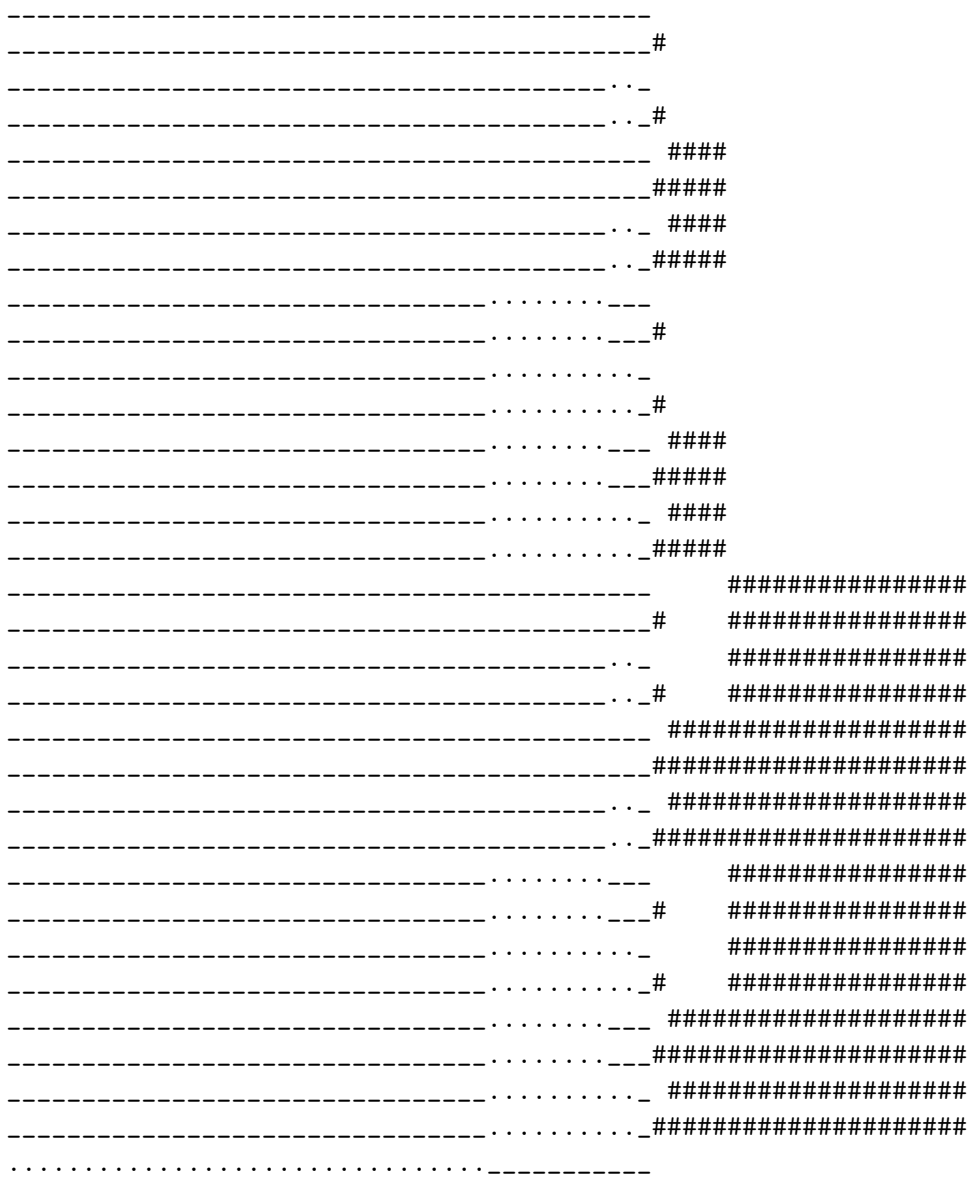


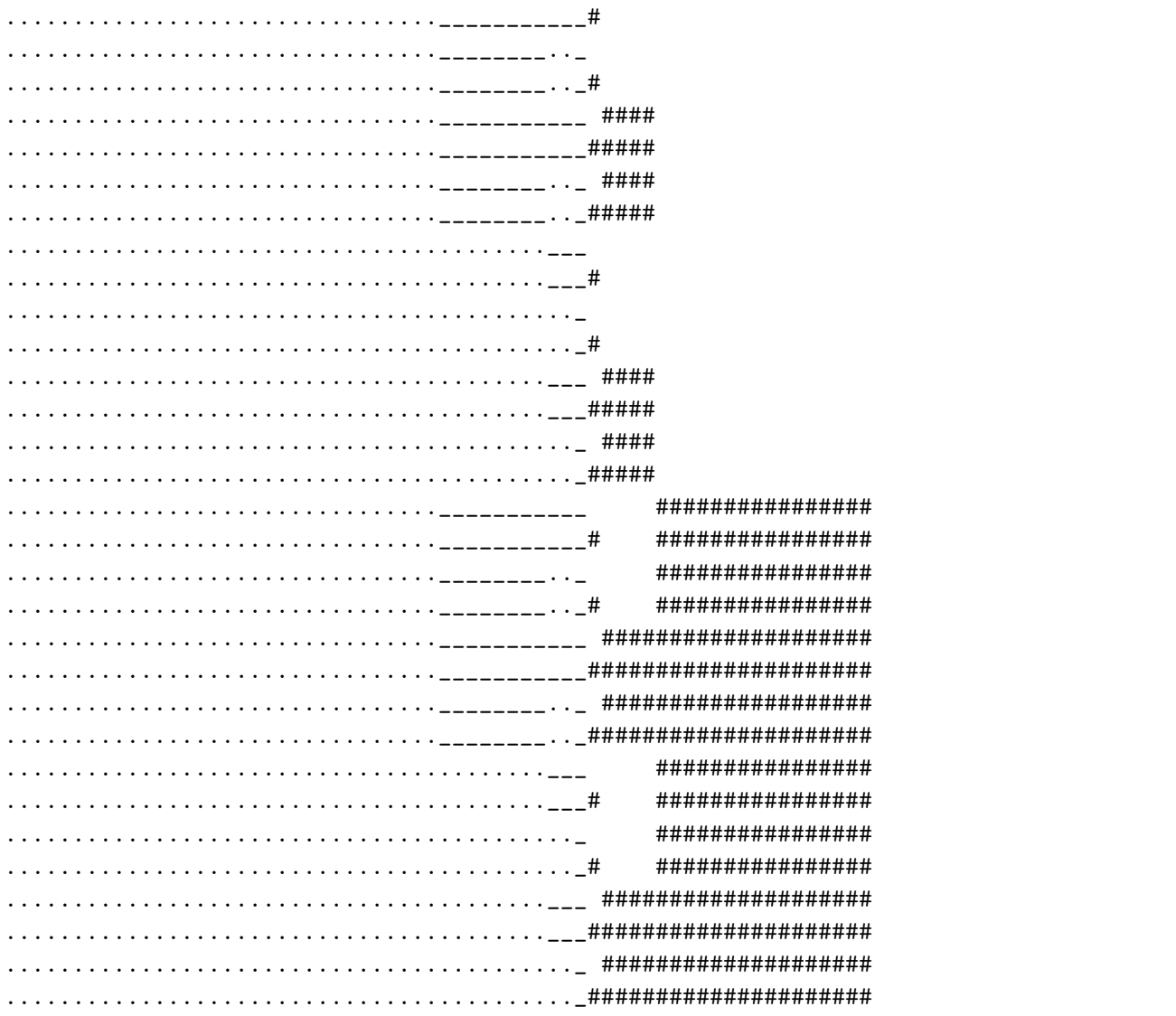


########### # # ##### # ### # #

### 3.5 Example 5

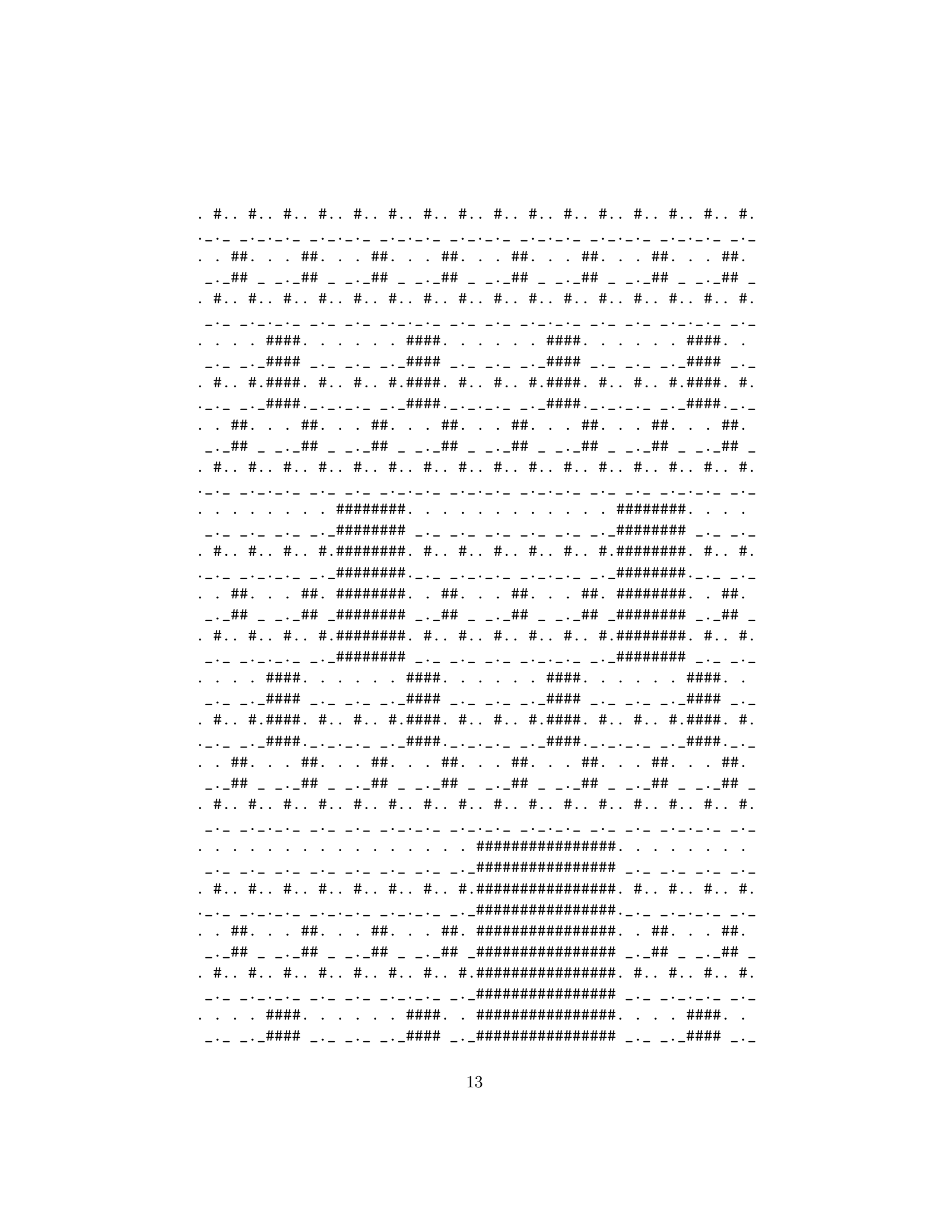
language=Lisp,label= ,caption= ,captionpos=b,numbers=none (cfg-expand 64 'a '((a . ((?$_b$?.$b$)?$_)$)($b$.(($a$? $a$?)? ))))

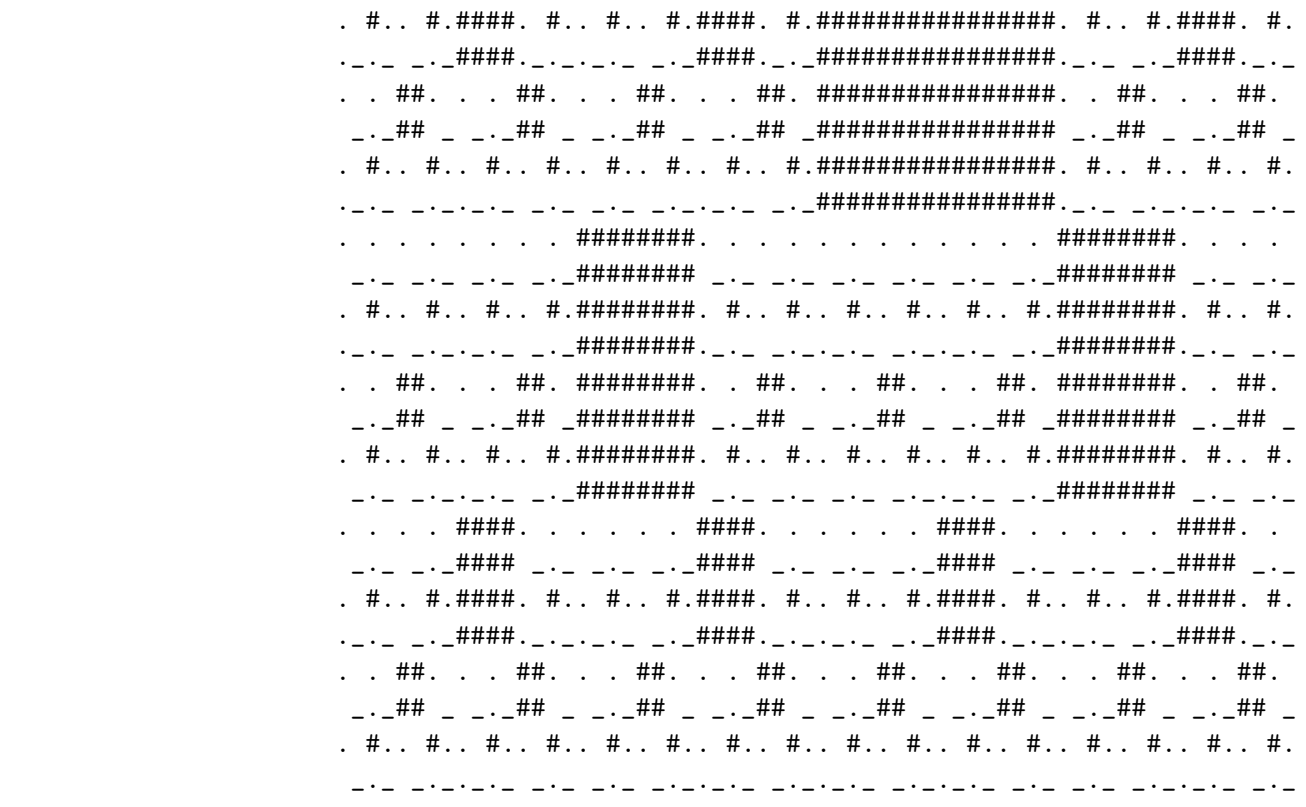




### 3.6 Example 6

0 -> 2 0 2 1 | white 1 -> black 2 2 1 | black 2 -> 2 0 0 1 | black
language=Lisp,label= ,caption= ,captionpos=b,numbers=none (cfg-expand 64 'a '((a . ((c a c b) ?)) (b . ((? c c b) ?))(*c*.((*caab*)?.))))





### 3.7 Example 7

language=Lisp,label= ,caption= ,captionpos=b,numbers=none (cfg-expand 64 'a '((a . ((b a b ?) ?|)) (b . ((b c ?a) ?=)) (c . ((b ?$_c$a)?−))))