Rule 110 in SOS

Christopher Eltschka

April 12, 2022

1 Program structure

First read the initial state from stdin, then in an infinite loop first output the current state, then calculate the next state.

1a ⟨*rule110.sos* 1a⟩≡

```
  ⟨read state 4c⟩
  (             main loop
    ⟨write state 6c⟩
    ⟨update state 8c⟩
  )             end main loop
```

2 General snippets

These are some general snippets that are used below.

2.1 Adding bits to the stack

Appending a 0 just means pushing an empty stack.

1b ⟨*0* 1b⟩≡ (3c 7–9 11 13)

```
  +             Push 0
```

Appending a 1 means adding a stack containing an empty stack.

1c ⟨*1* 1c⟩≡ (3c 7a 8a 11 13)

```
  +>+<          Push 1
```

2.2 Condition blocks

For an iftrue, the top of the stack represents the condition. If the stack is non-empty, it is considered true, otherwise false. Note that this top bit is consumed by the end of the construct.

Note also that the top bit is modified by the if and else, and must still be on the top when reaching else or endif.

2a ⟨*iftrue* 2a⟩≡ (3c 11) 2b▷

```
 (             Start if block
   >             Enter the condition.
```

At this point, we test for non-emptyness, and at the same time toggle the bit, so that the else part sees the complement of thje condition.

2b ⟨*iftrue* 2a⟩+≡ (3c 11) ◁2a

```
   +             Add another stack.
   %             Exchange top elements. Fails if stack was empty.
   (-)           Empty the stack.
   <             Go back up to the parent stack.
```

The else part closes the if block, and starts an else block. Note that this assumes that at the end of the if block, the emptied condition (or another empty stack replacing it) is still (or again) on the top of the current stack.

2c ⟨*else* 2c⟩≡ (3c 11) 2d▷

```
   >             Enter the condition again.
   -)            Terminate the if block by removing from empty stack
```

At this point, we are inside the toggled condition bit. Therefore we can directly start the else block by checking that the condition is not empty. If that check fails, the rest of the block is skipped.

2d ⟨*else* 2c⟩+≡ (3c 11) ◁2c 2e▷

```
   (-            Start else block.
```

We finally leave the condition again, so that the else block can again operate on the parent stack.

2e ⟨*else* 2c⟩+≡ (3c 11) ◁2d

```
   <             Leave the condition.
```

The endif will always be reached with an empty condition on the top. Therefore we can simply terminate the current block by entering it and tryping to delete from it (just like at the beginning of the else block).

3a ⟨*endif* 3a⟩≡ (3c 11) 3b▷
```
    >              Enter the condition again.
    -)             Terminate the block.
```

Finally, we leave and delete the condition stack (which now no longer contains useful information anyway)

3b ⟨*endif* 3a⟩+≡ (3c 11) ◁3a
```
    <-             Leave and delete the condition.
```

2.2.1 Test condition block

We test the condition block by first doing an if/else with a condition 0, then another with a condition 1. In both cases, the if block outputs an asterisk, the else block outputs a dot.

Afterwards we test that the remaining stack is indeed empty by trying to remove the top element, and then outputting another asterisk.

If everything works as it should, the output of this code should be a single dot followed by a single asterisk.

3c ⟨*testif.sos* 3c⟩≡
```
  ⟨0 1b⟩
  ⟨iftrue 2a⟩
    ⟨write asterisk 4b⟩
  ⟨else 2c⟩
    ⟨write dot 4a⟩
  ⟨endif 3a⟩

  ⟨1 1c⟩
  ⟨iftrue 2a⟩
    ⟨write asterisk 4b⟩
  ⟨else 2c⟩
    ⟨write dot 4a⟩
  ⟨endif 3a⟩

  -              Try to delete fr empty stack, terminates program

  ⟨write asterisk 4b⟩ Should never be executed.
```

2.3 Constant character outputs

These to characters are used in the test programs.

We use the following auxiliary routine to output a single dot.

4a ⟨*write dot* 4a⟩≡ (3c 5a)

```
+>!!+!-!+!!!-!<-
```

We also use the following routine to output an asterisk

4b ⟨*write asterisk* 4b⟩≡ (3c 5a 7a)

```
+>!!+!-!+!-!+!-!<-
```

3 Read state

The state is given in stdin as sequence of 0 and 1. Actually only the last bit of the byte is read, so anything with an even ASCII code can be used for 0, and anything with an odd ASCII code can be used as 1.

The loop is terminated on EOF. Since that happens in the middle of the loop, we need to clean up afterwards.

4c ⟨*read state* 4c⟩≡ (1a 5a)

```
+>             Create a new stack to hold the input and enter it
(              read loop
  ⟨read bit 5b⟩
)              end read loop
⟨cleanup read 6b⟩
<              Leave the input stack and return to root.
```

3.0.1 Testing

We can do several tests of this code. If all goes well, this should print four dots, then the bit string that was input (using the output routine below)

5a ⟨*testreadstate.sos* 5a⟩≡

```
⟨write dot 4a⟩
⟨read state 4c⟩
⟨write dot 4a⟩
(
  %             Exchange. If there's only one element on the root
                stack, this terminates the loop.
  ⟨write asterisk 4b⟩
  (-)           Delete everything. This ensures the next test will
                terminate the program.
-)
⟨write dot 4a⟩
=             Duplicate. If the root stack is empty, this terminates
              the program.
⟨write dot 4a⟩
⟨write state 6c⟩
```

3.1 Read a bit

To read a bit, we read a byte and discard everything but the last bit.

If the read bit was 0, an empty stack is added to the current stack. Otherwise, a stack containing an empty stack is added to the current stack.

5b ⟨*read bit* 5b⟩≡ (4c 6a)

```
+>            Create an emtpy stack and enter it.
???????       Read the first 7 bits of the byte.
(-)           Discard everything read so far.
?             Read the lowest bit.
<             Leave the stack of the new bit, returning to the parent stack.
```

3.1.1 Testing bit reading

This code can be tested by a program that simply reads bits and outputs them again (using the output routine defined further below).

6a ⟨*testreadbit.sos* 6a⟩≡

```
(              loop until EOF
  ⟨read bit 5b⟩
  ⟨write and delete bit 7b⟩
)
```

3.2 Cleaning up after read

On encountering EOF, we are inside a newly created empty stack. We need to leave and deletethat empty stack.

6b ⟨*cleanup read* 6b⟩≡ (4c)

```
<-             Delete extra bit.
```

4 Write state

To write the current state, we first duplicate the stack containing the state, and then output the copy bit by bit, consuming the state as we go, so we can and the loop on empty stack.

After we've written all bits, we write a newline and leave/remove the now empty auxiliary stack.

Note that we print the stack bottom to top (which is the same order we've read it). Therefore in each iteration the bottom-most bit has to be brought to the top in order to write it.

6c ⟨*write state* 6c⟩≡ (1a 5a 7a)

```
=>             Duplicate state stack and enter duplicate
(              Write loop
  {              Bring up the next bit to write.
  ⟨write and delete bit 7b⟩
)
⟨write newline 8b⟩
<-             Leave and remove the duplicate stack.
```

4.1 Test writing the state

This code creates the bit sequence 01101110, and then outputs it. It then verifies that the original sequence is still on the top of the stack by outputting it again. Finally, it tests that there's nothing else left on the stack by trying to exchange (which should end the program) and then outputting an asterisk (which should therefore never appear).

In other words, if all works well, the output of this code is `0110111001001110`.

7a ⟨*testwritestate.sos* 7a⟩≡

```
+>
⟨0 1b⟩⟨1 1c⟩⟨1 1c⟩⟨0 1b⟩⟨1 1c⟩⟨1 1c⟩⟨1 1c⟩⟨0 1b⟩
<
⟨write state 6c⟩
⟨write state 6c⟩
%
⟨write asterisk 4b⟩
```

4.2 Write a single bit

The bit is output as either the digit 0 or the digit 1. The ASCII code for bit value `x` is `0011000x`. Thus we can just output those 7 bits, followed by the actual bit.

7b ⟨*write and delete bit* 7b⟩≡ (6 8a 13)

```
>             Enter bit to output. Terminates loop if none left.
+>            Create and enter auxiliary empty stack
!!+!!-!!!     Output 0011000.
<-            Leave and destroy auxiliary stack.
!             Output the current bit.
<-            Leave and delete the bit.
```

4.2.1 Testing write bit

We can test this routine with a simple routine that outputs a 0 and an 1 digit, then tries to write a nonexistent digit, which should terminate the program.

To check that this really works, another zero bit output follows, which should not be executed

8a ⟨*testwritebit.sos* 8a⟩≡
  ⟨*0* 1b⟩
  ⟨*write and delete bit* 7b⟩
  ⟨*1* 1c⟩
  ⟨*write and delete bit* 7b⟩
  ⟨*write and delete bit* 7b⟩
  ⟨*0* 1b⟩
  ⟨*write and delete bit* 7b⟩

4.3 Write newline

When we reach this code, we are in the emptied duplicate stack. Therefore we don't need another auxiliary stack.

The line feed character has bit pattern 00001010.

8b ⟨*write newline* 8b⟩≡ (6c 13)
```
  !!!!+!-!+!-!  Line feed
```

5 Update state

To update the state, we first add an extra zero to both ends of the current state. Then we create the new state, consuming the old one in the process, and finally we destroy the old stack.

8c ⟨*update state* 8c⟩≡ (1a)
  ⟨*pad current state* 9a⟩
  ⟨*calculate next state* 9b⟩
  ⟨*discard old state* 14⟩

5.1 Pad current state with zeros

We push two zeroes to the top (that is the end), and then rotate the stack so that one of the zeros goes

9a ⟨*pad current state* 9a⟩≡ (8c)

```
  >             Enter the current state
  ⟨0 1b⟩
  ⟨0 1b⟩
  }             Put the second zero at the beginning
  <             Leave the state again.
```

5.2 Calculating the next state

To calculate the next state, we first push an empty stack to get the new state in, which we bury at the bottom of the current stack.

9b ⟨*calculate next state* 9b⟩≡ (8c) 9c▷

```
  +}            Create stack for new state at bottom of current stack.
```

Next we enter a loop, where we first put the top three bits into a substack (in reverse order). This will fail if the stack contains less than three elements.

9c ⟨*calculate next state* 9b⟩+≡ (8c) ◁9b 9d▷

```
  (
    >             Enter current state
    +%^%^%^       Move top three bits into substack, reversing order
```

We copy two of them back onto the state stack, as we will need them for the next iteration.

9d ⟨*calculate next state* 9b⟩+≡ (8c) ◁9c 9e▷

```
    >=<_%         Copy topmost bit back
    >}=<_%>{<     Copy second to top bit back
```

Next, we pop the new three-bit stack from the current state, so we can work on it from the root stack.

9e ⟨*calculate next state* 9b⟩+≡ (8c) ◁9d 10a▷

```
    <_            Go back to root and pop the three bit stack.
```

Now we apply rule 110 to those three bits. This replaces that stack with the single bit output.

10a ⟨*calculate next state* 9b⟩+≡ (8c) ◁9e 10b▷
```
    ⟨apply rule 110 10e⟩
```

Finally, we add that state on the new stack. Since we are working backwards, we have to move it to the beginning there.

10b ⟨*calculate next state* 9b⟩+≡ (8c) ◁10a 10c▷
```
    {             Get new state back from the bottom
    %             Get new bit up from below it
    ^             Push the new bit
    >}<           Move the new bit to the bottom of the new state
    }             Move the new state back to the bottom.
```

With this, the loop is complete.

10c ⟨*calculate next state* 9b⟩+≡ (8c) ◁10b 10d▷
```
  )
```

Since we are inside the old state when leaving the loop, we have to go back up afterwards.

10d ⟨*calculate next state* 9b⟩+≡ (8c) ◁10c
```
  <             Move back up to root.
```

5.2.1 Applying rule 110 to three bits

This is the actual implementation of rule 110. It takes a stack of three bits, with the first on the top, and replaces it

10e ⟨*apply rule 110* 10e⟩≡ (10a 13) 11▷
```
  >             Enter the three bit stack
```

At this point, the current stack contains the three bita needed to calculate the new state's bit. Note that each bit is consumed by the time we reach endif.

Rule 110 is given by the following table:

111	110	101	100	011	010	001	000
0	1	1	0	1	1	1	0

This translates into a list of nested ifs. Note that each if tests (and consumes) the top bit.

```
11   ⟨apply rule 110 10e⟩+≡                                          (10a 13) ◁10e
       ⟨iftrue 2a⟩
         }              Get to the next bit
         ⟨iftrue 2a⟩
           }              Get to the next bit
           ⟨iftrue 2a⟩
             }              Bury that bit
             ⟨0 1b⟩
             {              Bring it back up for the else
           ⟨else 2c⟩
             }
             ⟨1 1c⟩
             {
           ⟨endif 3a⟩
           {
         ⟨else 2c⟩
           }
           ⟨iftrue 2a⟩
             }
             ⟨1 1c⟩
             {
           ⟨else 2c⟩
             }
             ⟨0 1b⟩
             {
           ⟨endif 3a⟩
           {
         ⟨endif 3a⟩
         {
       ⟨else 2c⟩
         }
         ⟨iftrue 2a⟩
           }
           -              Here we need no nested if since both cases are 1.
           ⟨1 1c⟩
           {
         ⟨else 2c⟩
```

```
      }
      ⟨iftrue 2a⟩
         }
         ⟨1 1c⟩
         {
      ⟨else 2c⟩
         }
         ⟨0 1b⟩
         {
      ⟨endif 3a⟩
      {
   ⟨endif 3a⟩
{
⟨endif 3a⟩
<_%-          Move the resulting bit one level up.
```

5.2.2 Testing application of rule 110

This program teststhe above code, by generating all eight three bit combinations in the order of the table above (remembering that the leftmost bit must be on the top, thus the order is reversed in the code), then applying that code, and writing the resulting bit.

If working correctly, the test code should print 01101110, or the binary representation of the number 110.

13 ⟨*test110.sos* 13⟩≡

```
+>
⟨1 1c⟩⟨1 1c⟩⟨1 1c⟩
<
⟨apply rule 110 10e⟩
⟨write and delete bit 7b⟩

+>
⟨0 1b⟩⟨1 1c⟩⟨1 1c⟩
<
⟨apply rule 110 10e⟩
⟨write and delete bit 7b⟩

+>
⟨1 1c⟩⟨0 1b⟩⟨1 1c⟩
<
⟨apply rule 110 10e⟩
⟨write and delete bit 7b⟩

+>
⟨0 1b⟩⟨0 1b⟩⟨1 1c⟩
<
⟨apply rule 110 10e⟩
⟨write and delete bit 7b⟩

+>
⟨1 1c⟩⟨1 1c⟩⟨0 1b⟩
<
⟨apply rule 110 10e⟩
⟨write and delete bit 7b⟩

+>
⟨0 1b⟩⟨1 1c⟩⟨0 1b⟩
<
⟨apply rule 110 10e⟩
⟨write and delete bit 7b⟩
```

```
+>
⟨1 1c⟩⟨0 1b⟩⟨0 1b⟩
<
⟨apply rule 110 10e⟩
⟨write and delete bit 7b⟩

+>
⟨0 1b⟩⟨0 1b⟩⟨0 1b⟩
<
⟨apply rule 110 10e⟩
⟨write and delete bit 7b⟩

⟨write newline 8b⟩
```

5.3 Discarding the remains of the old state

Since when we get here, the old state is at the top of the stack, we can just delete it.

14 ⟨*discard old state* 14⟩≡ (8c)

```
-              Discard remains of old state
```