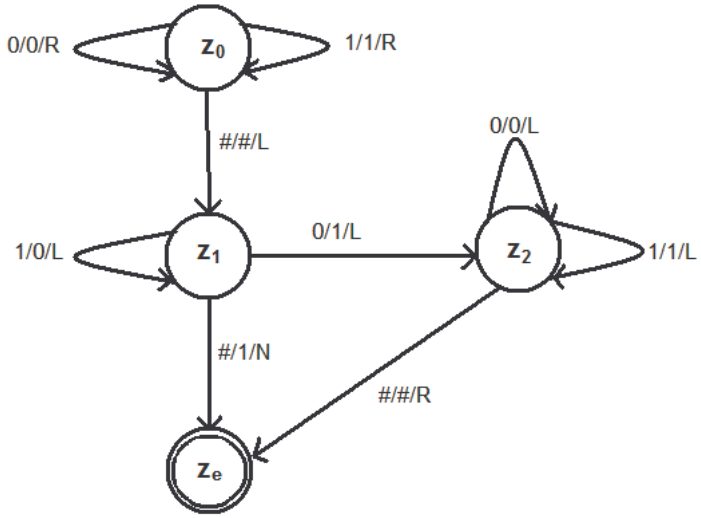
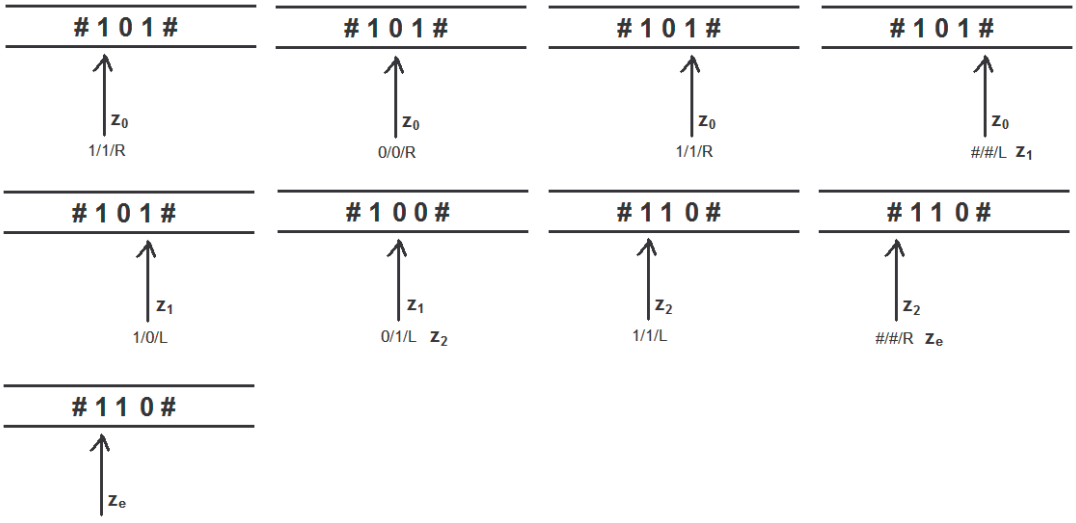
Solutions
Exercises 2: Software language theory

1) **Use the turing machine from the lecture for adding 1 to a given bit representation and write the complete path when #101# (# = blank symbol) is the starting input on the tape.**
E.g. the transition $\delta(z_0,0) = (z_0,0,R)$ says when machine is in state z_0 and reads 0 on tape the new state is z_0 again and the machine writes 0 and the read/write-head moves one position to right (R)
When you watch the complete description it is possible to draw a state chart (e.g. 0/0/R means read 0, write 0 and move Right):



When you follow the path through the state graph it is possible to write the automation steps based on a given input on the tape



A short form of that is:
#S101# -> #1S01# -> #10S1# -> #101S# -> #10S1# -> #1S00# -> #S110# -> S#110# -> #S110#

2) **Write a simple grammar with production rules to check the correctness of a given German postal address like:**
<first name> <family name>
<street name> <street number>
<country code>-<postal code> <town name>
(<country>)
e.g.
Andrew Exampleman
Examplestreet 5
D-80585 Example Town
(Germany)

(\n stands for new line, \s stands for white space, red colour is used for non-terminals and green is used for terminals)
Address -> Name '\n' Street '\n' Town '\n'
Name -> Firstname '\s' Familyname
Street-> Streetname '\s' Streetnumber
Town -> 'D–' Postalcode '\s' Townname '\n' '(Germany)' |
'A-' Postalcode '\s' Townname '\n' '(Austria)' |
...
Firstname -> String
Familyname -> String | String '-' String
Streetname -> String | Streetname '\s' String
Streetnumber -> Digit | Digit Character
Postalcode -> Digit Digit Digit Digit Digit
Townname-> String | Townname '\s' String
String -> Character | String
Character -> 'A' | 'B' | 'C' | ... | 'Y' | 'Z' | 'a' | 'b' | 'c' | ... | 'y' | 'z'
Digit -> '0' | '1' | '2' | ... | '9'

3) **Which type of Chomsky language is needed for those checks?**
A first view shows a Chomsky language type 2. But it is possible to reduce the production rules to types A -> aB or A -> a where "A", "B" are non-terminal symbols and "a" is a terminal symbol. Each following state bases just on the previous one. The check can be done with regular expressions. So it is a type 3 grammar (regular language)
E.g.: You can define it by (terminal symbols a marked with ''):
First name
A0 -> 'A' | 'B' | ... | 'Z'| 'a' | ... | 'z' | 'A' A0 | 'B' A0 | ... | 'Z' A0 | 'a' A0 | ... | 'z' A0 |
'\s' A1
Family name
A1 -> 'A' | 'B' | ... | 'Z'| 'a' | ... | 'z' | 'A' A1 | 'B' A1 | ... | 'Z' A1 | 'a' A1 | ... | 'z' A1 |
'-' A2
A2 -> 'A' | 'B' | ... | 'Z'| 'a' | ... | 'z' | 'A' A2 | 'B' A2 | ... | 'Z' A2 | 'a' A2 | ... | 'z' A2|
'\n' A3
Street name
A3 -> 'A' | 'B' | ... | 'Z'| 'a' | ... | 'z' | 'A' A3 | 'B' A3 | ... | 'Z' A3 | 'a' A3 | ... | 'z' A3 |
'\s' A3 | '\s0' A4 | '\s1' A4 | ... | '\s9' A4

Street number
A4 -> ‘0’ | ... | ‘9’ | ‘0’ A4 | ... | ‘9’ A4 |
‘\n’ A5
Country code
A5 -> ‘D-‘ A6 | ‘A-‘ A20 | ...
Postal code Germany
A6 -> ‘0’ A7 | ... | ‘9’ A7
A7 -> ‘0’ A8 | ... | ‘9’ A8
A8 -> ‘0’ A9 | ... | ‘9’ A9
A9 -> ‘0’ A10 | ... | ‘9’ A10
A10 -> ‘0’ A11 | ... | ‘9’ A11
A11 -> ‘\s’ A12
German town
A12 -> ‘A’ | ‘B’ | ... | ‘Z’| ‘a’ | ... | ‘z’ | ‘A’ A12 | ‘B’ A12 | ... | ‘Z’ A12 | ‘a’ A12 | ... |
‘z’ A12 | ‘\s‘ A12 | ‘\n(Germany)’
Postal code Austria
A20 ->...
...

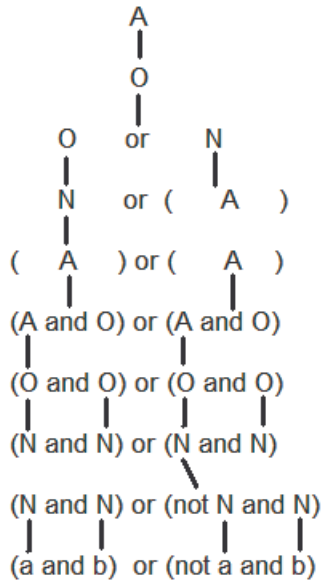
4) **Adapt the given ETF-grammar (presentation slide no. 21) for logical expressions (and,or,not).**

ETF-rammar:
Non-terminal symbols: E, T, F
Terminal symbols: +, *, (,), a, b, c, ..., z
Startsymbol: E
E → T | E+T
T → F | T*F
F → (E) | a | b | c | d | … | z

New grammar for logical expressions:
Non-terminal symbols: A, O, N
Terminal symbols: and, or, not, (,), a, b, c, ..., z
Startsymbol: A
Production rules:
(-> Replace left side of arrow with right side of arrow;
| can be optionally replaced by one or the other, e.g. A -> O | A and O means A can be replaced by O or by A and O)
A -> O | A and O
O -> N | O or N
N -> (A) | not N | a | b | c | ... | z

For example to produce “(a and b) or (not a and b)” we start with non-terminal “A” and replace it by non-terminal “O”. After we replace “O” by “O or N”. Then replace “O” by “N” and “N” by “(A)” and so on until the logical expression is produced. During each step terminal symbols are replaced according to production rules and non-terminal symbols are kept. Step by step we can follow the production rules until only terminal symbols remains.

(a and b) or (not a and b)



5) **Write a simple grammar which produces**
if (condition) then
{
block
}
else
{
block
}
Where block can be an encapsulated if-then-else and the else-part is not necessarily needed.

Non-terminal symbols: Start, A, B
Terminal symbols: 'if (condition) then', 'if (condition) then', 'else', 'block', {, }
Startsymbol: Start
Production rules:
Start -> 'if (condition) then' B | 'if (condition) then' B 'else' B
A -> Start | 'block'
B -> {A}