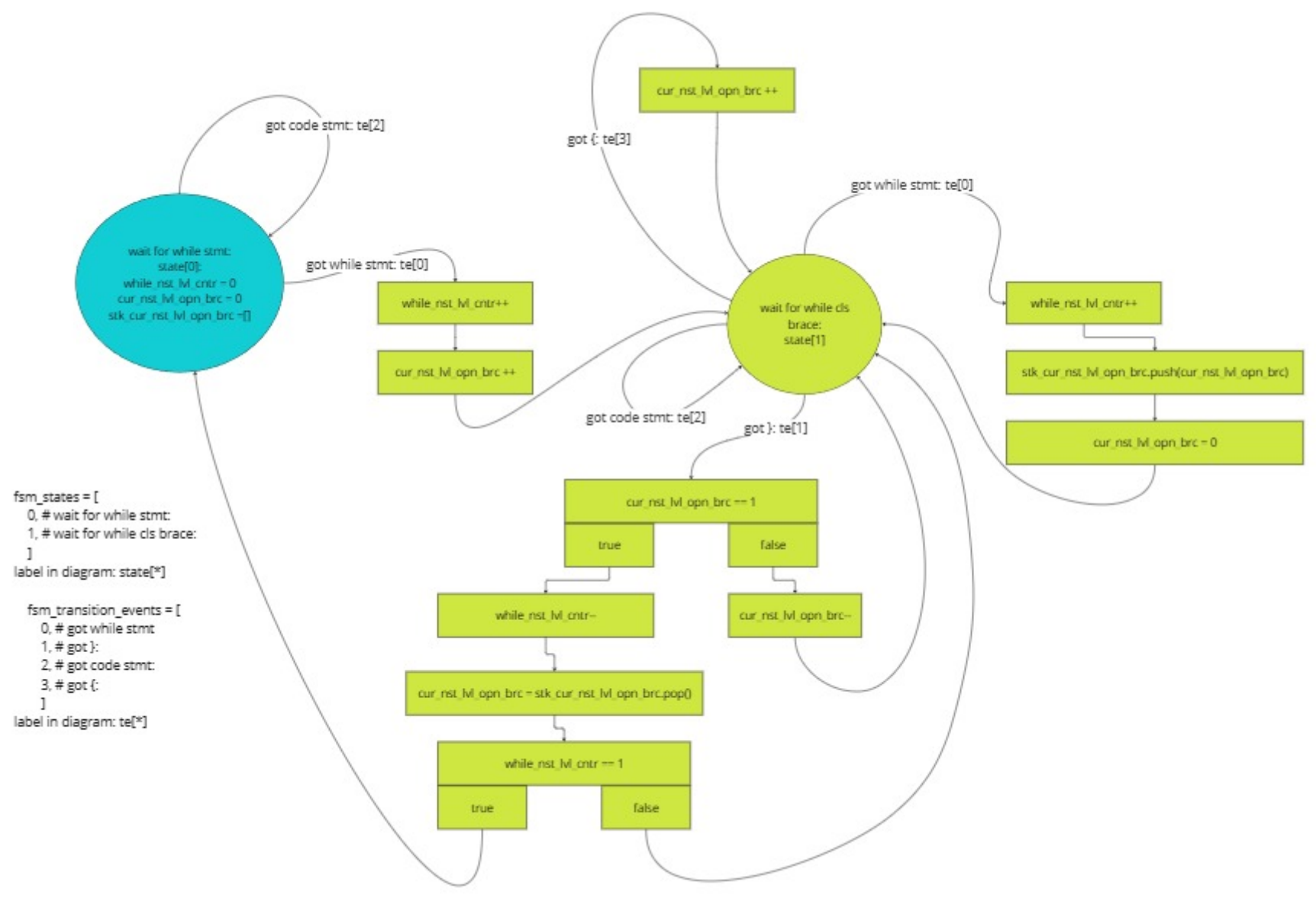
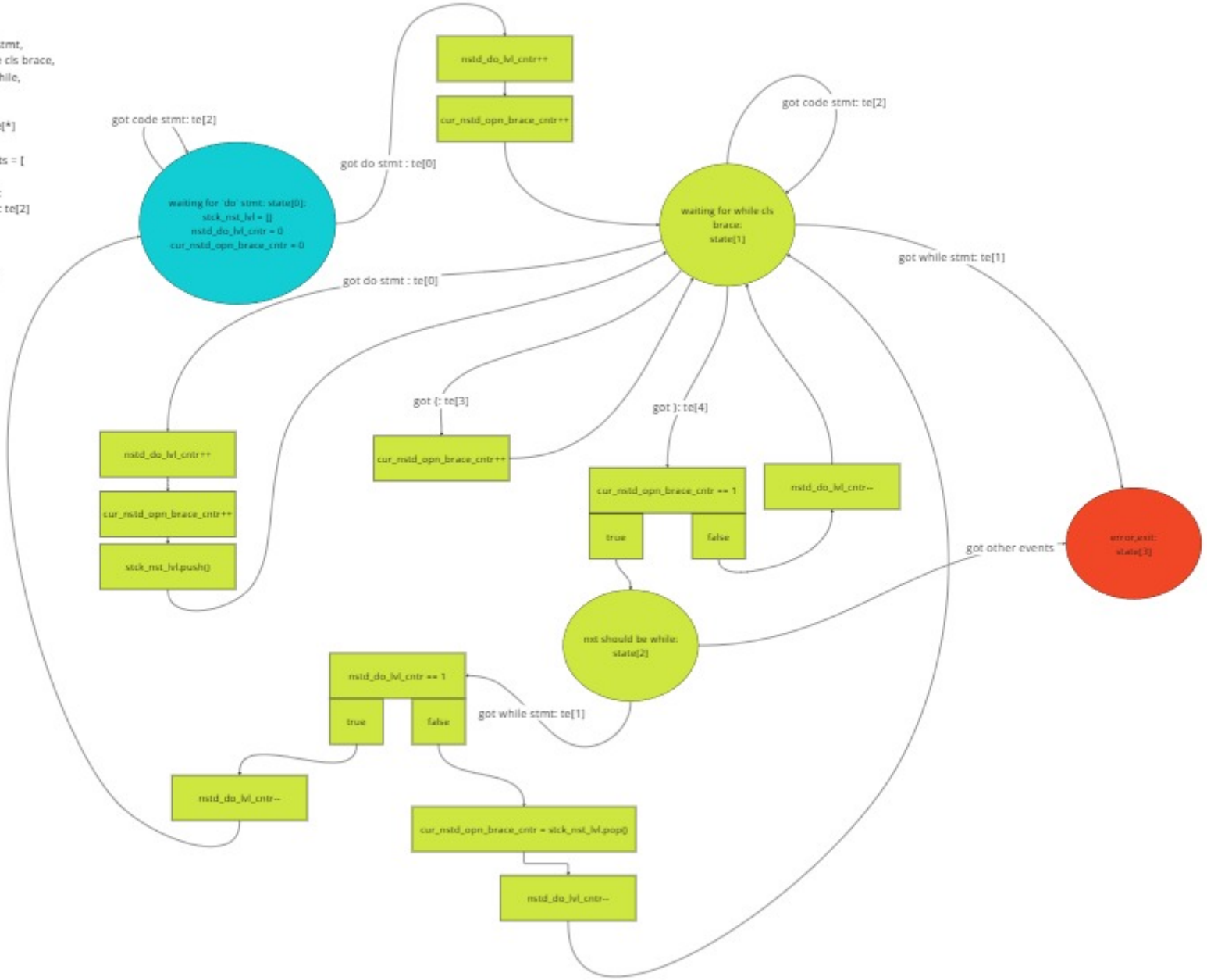

got code stmt: te[2]
wait for while stmt:
state[0]:
while_nst_lvl_cntr = 0
cur_nst_lvl_opn_brc = 0
stk_cur_nst_lvl_opn_brc =[]
got while stmt: te[0]
while_nst_lvl_cntr++
cur_nst_lvl_opn_brc ++
cur_nst_lvl_opn_brc ++
got {: te[3]
got while stmt: te[0]
wait for while cls
brace:
state[1]
while_nst_lvl_cntr++
stk_cur_nst_lvl_opn_brc.push(cur_nst_lvl_opn_brc)
cur_nst_lvl_opn_brc = 0
got code stmt: te[2]
got }: te[1]
cur_nst_lvl_opn_brc == 1
true
false
while_nst_lvl_cntr--
cur_nst_lvl_opn_brc--
cur_nst_lvl_opn_brc = stk_cur_nst_lvl_opn_brc.pop()
while_nst_lvl_cntr == 1
true
false
fsm_states = [
0, # wait for while stmt:
1, # wait for while cls brace:
]
label in diagram: state[*]
fsm_transition_events = [
0, # got while stmt
1, # got }:
2, # got code stmt:
3, # got {:
]
label in diagram: te[*]

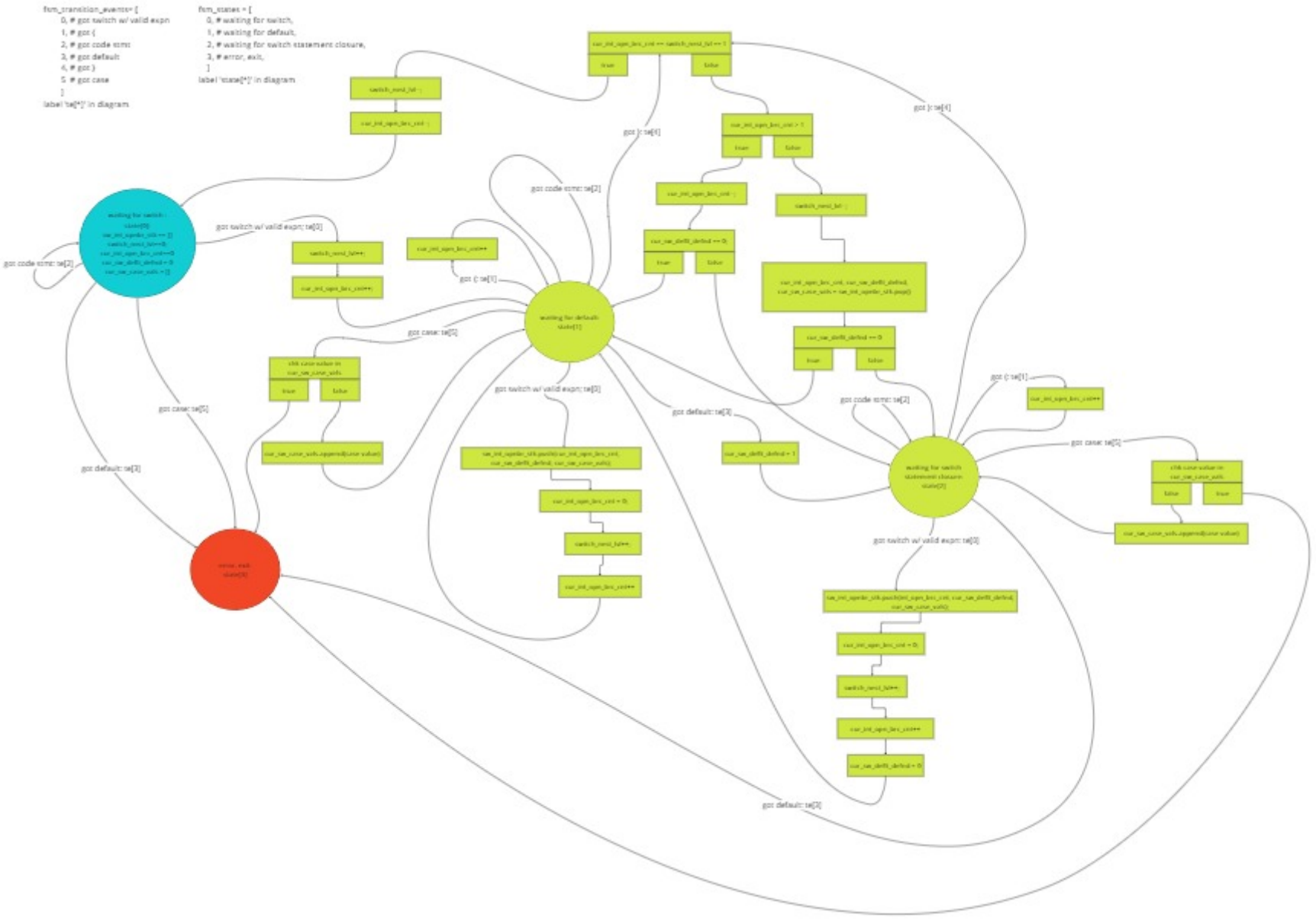
```
fsm_states = [
    0, # waiting for 'do' stmt,
    1, # waiting for while cls brace,
    2, # nxt should be while,
    3, # error, exit,
    ]
label in duagram: state[*]

    fsm_transition_events = [
        0, # got do stmt
        1, # got while stmt
        2, # got code stmt: te[2]
        3, # got (
        4  # got )
        ]
label in diagram: te[*]
```



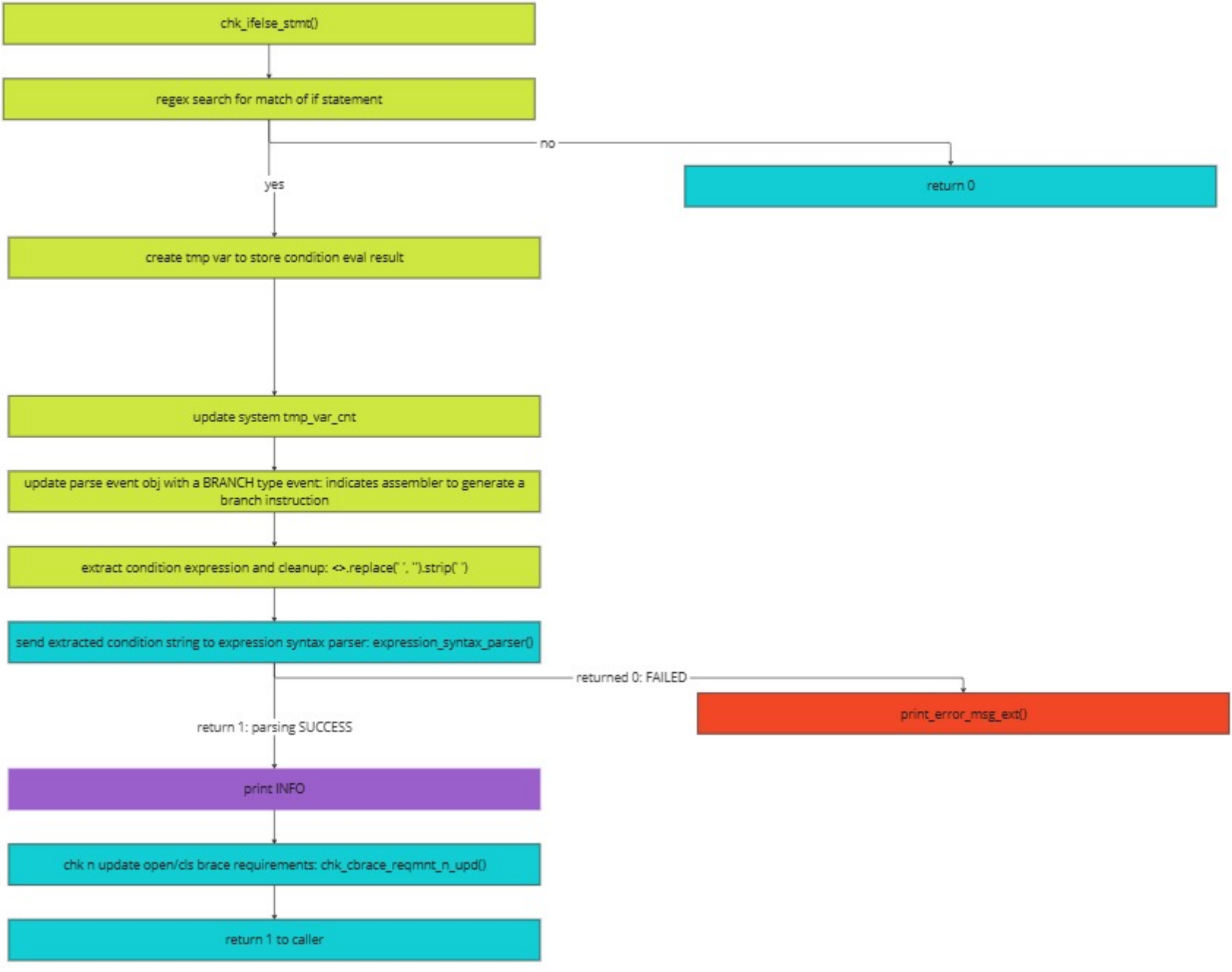
```
fsm_transition_events= [
    0, # got switch w/ valid expn
    1, # got (
    2, # got code stmt
    3, # got default
    4, # got )
    5  # got case
    ]
label 'ts[*]' in diagram
```

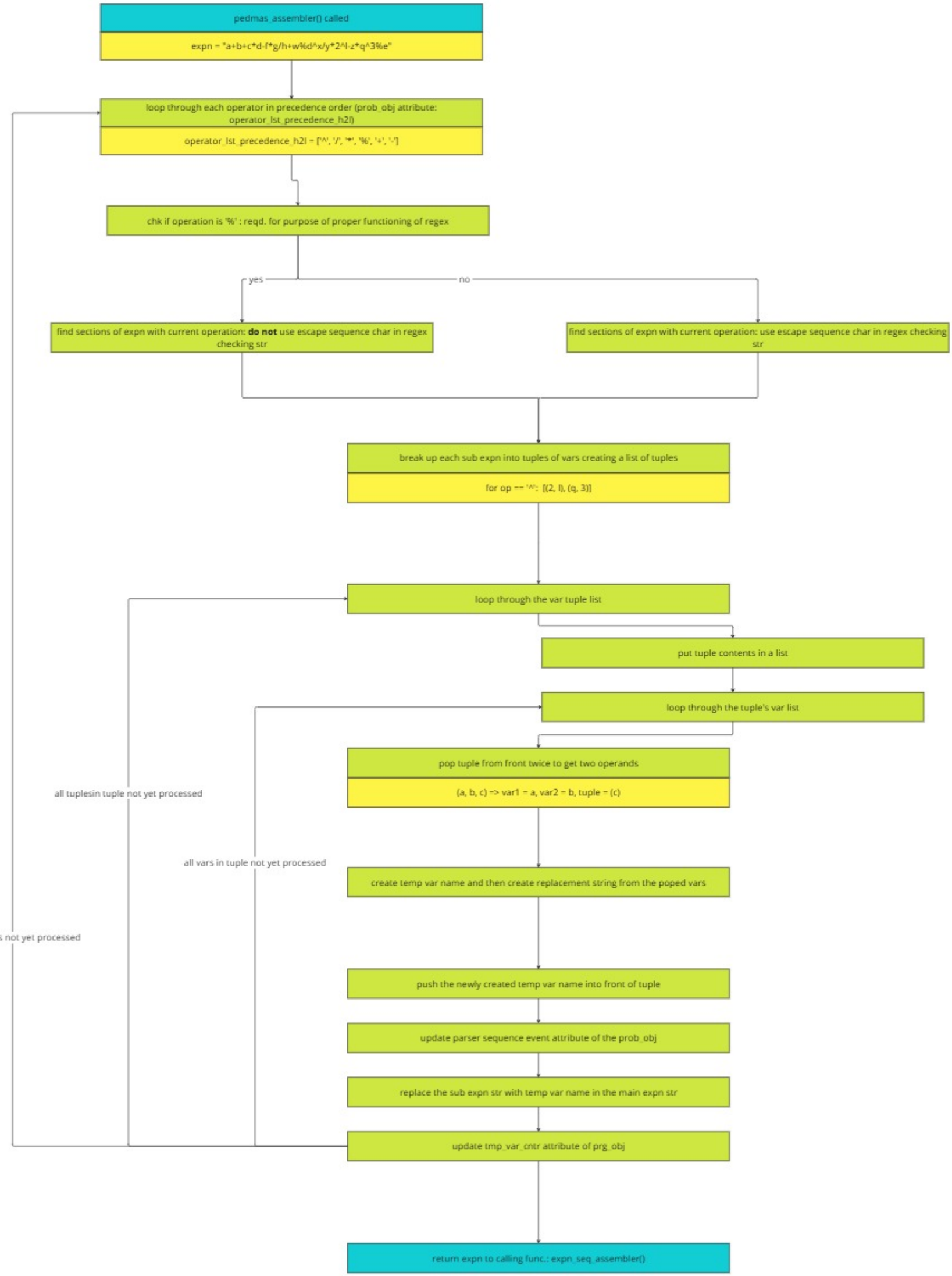
```
fsm_states = [
    0, # waiting for switch,
    1, # waiting for default,
    2, # waiting for switch statement closure,
    3, # error, exit,
    ]
label 'state[*]' in diagram
```

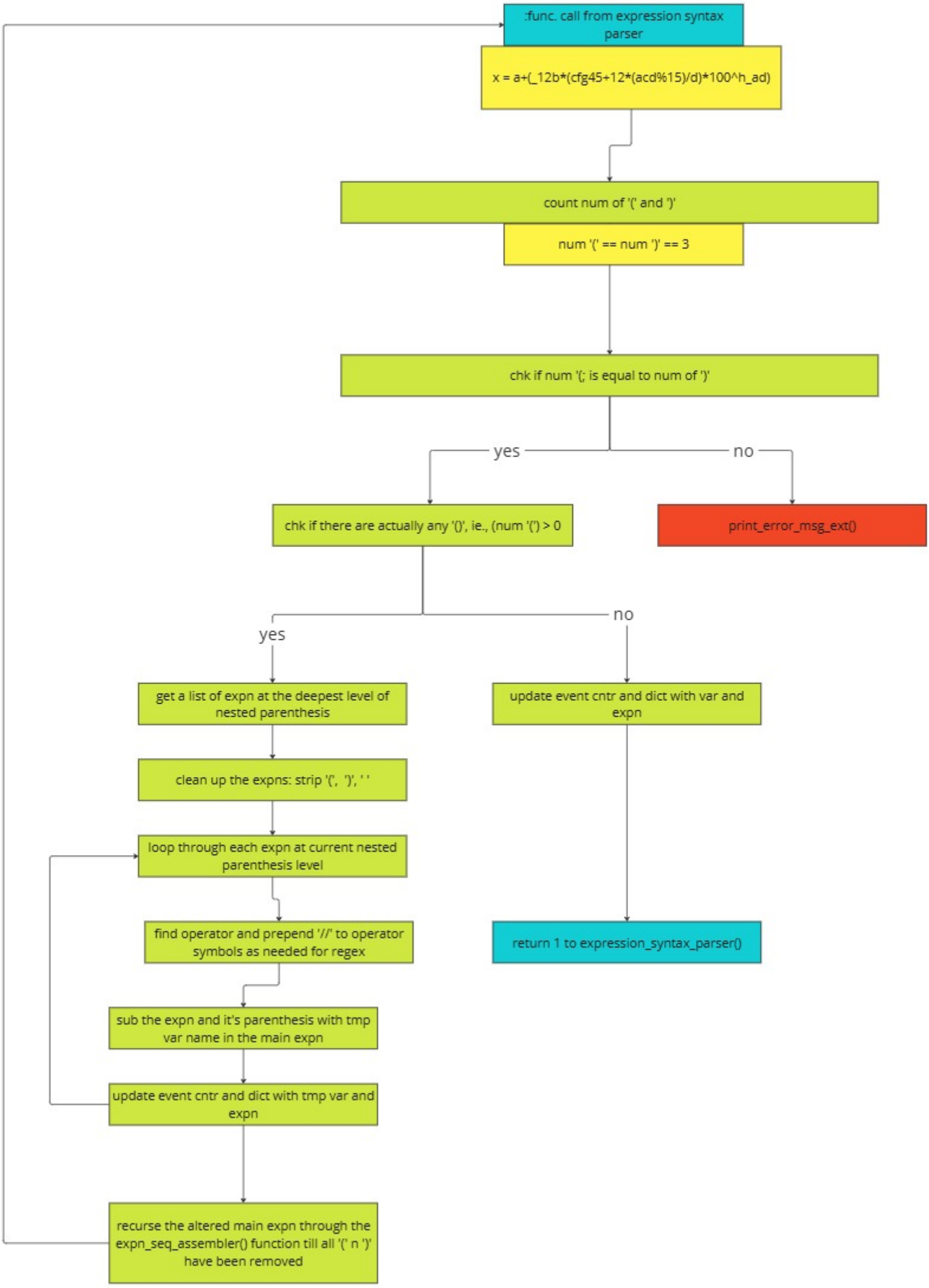


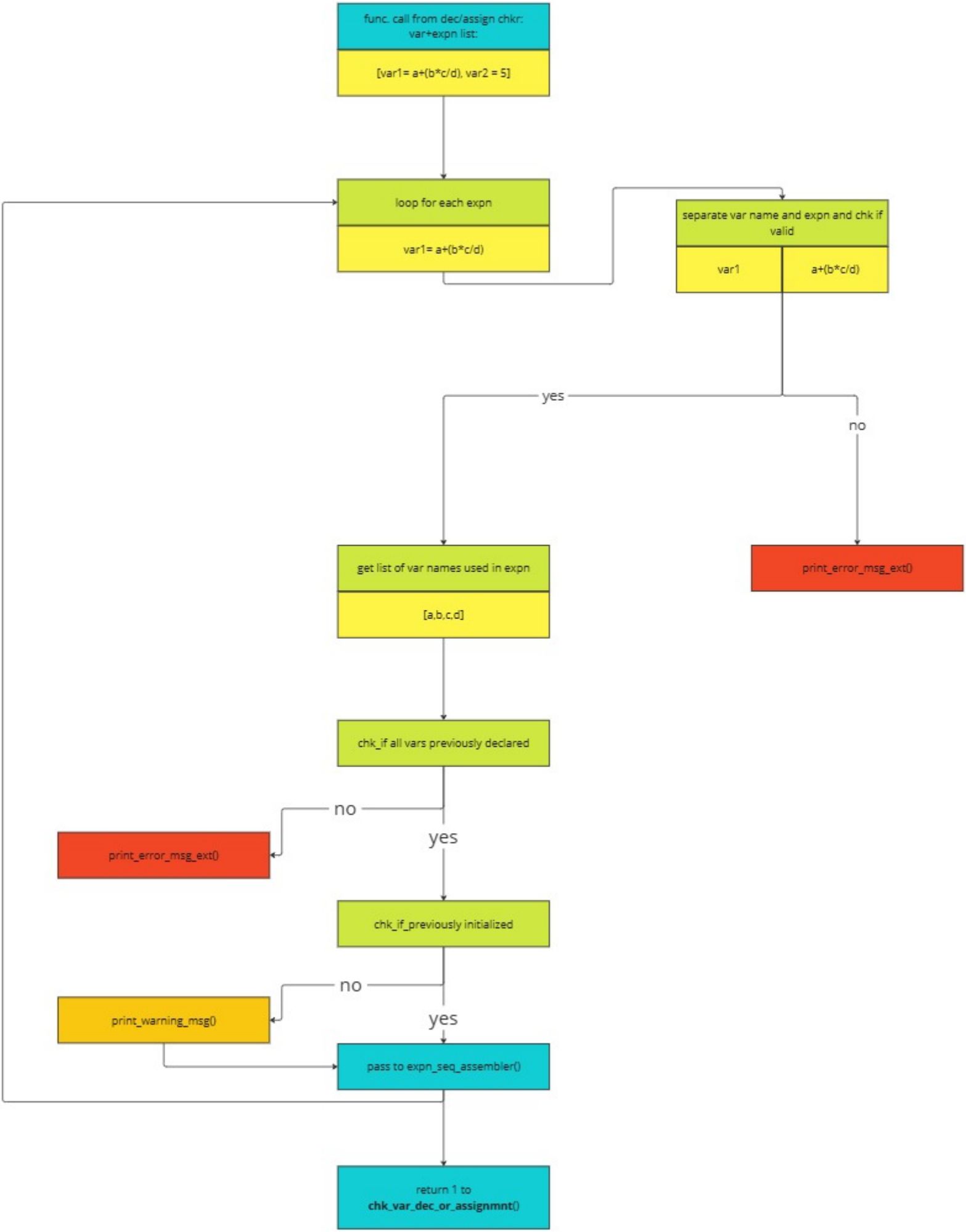
```
fsm_transition_events = [
    0, #'if with '
    1, #any code stmt
    2, #nested if
    3, #got '}'
    4, #got else if
    5  #got else
    ]
label in diagram: te[*]
```

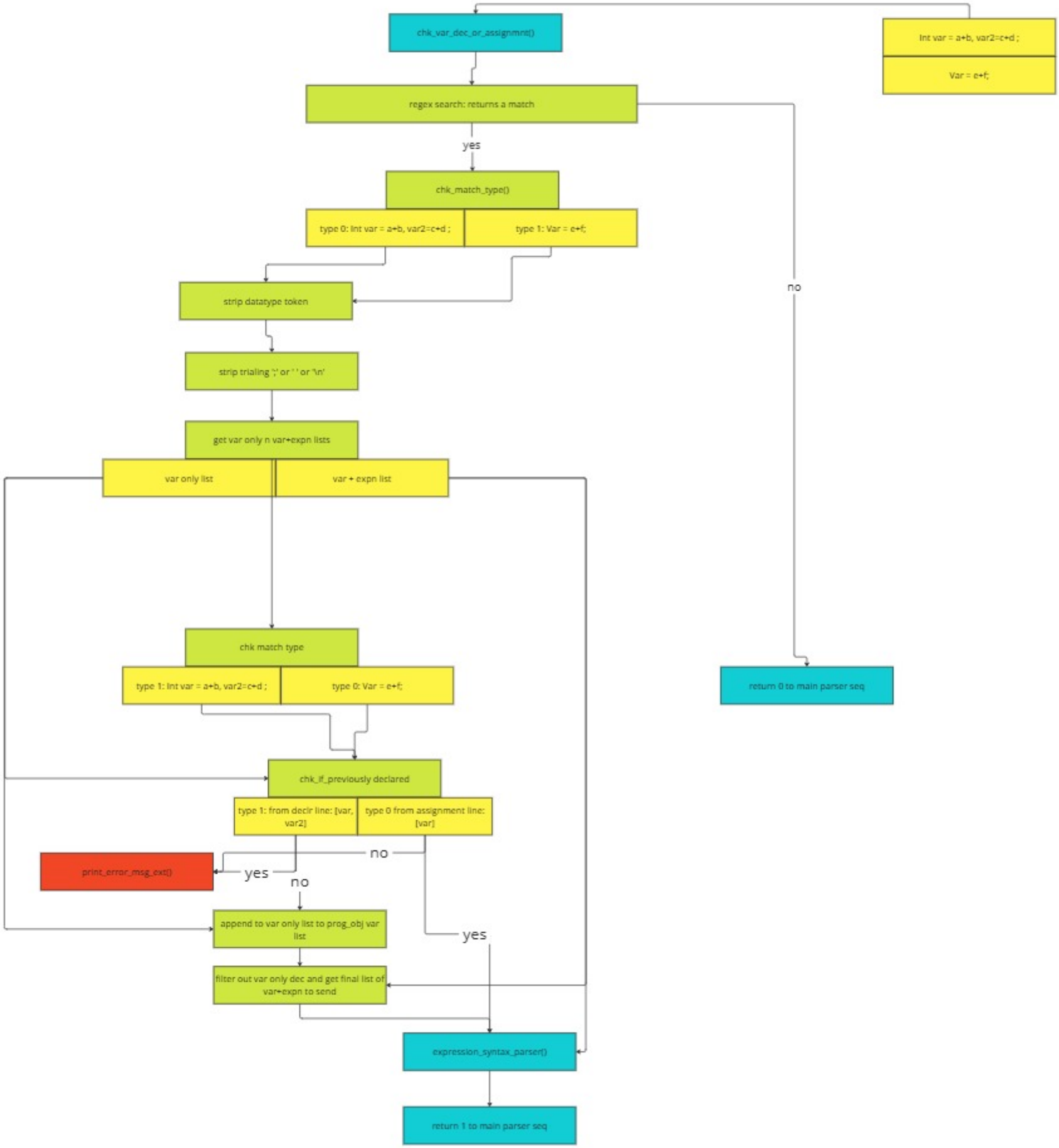
```
fsm_states = [0, #'waiting for  if ',
   1, #'nxt can be else or else if',
   2, #'waiting for else cls brace',
   3, #'waiting for else if cls brace',
   4, #'waiting for if cls brace',
   5  #'error, exit'
   ]
label in diagram: state[*]
```


chk_ifelse_stmt()
regex search for match of if statement
no
return 0
yes
create tmp var to store condition eval result
update system tmp_var_cnt
update parse event obj with a BRANCH type event: indicates assembler to generate a branch instruction
extract condition expression and cleanup: <>.replace(' ', '').strip(' ')
send extracted condition string to expression syntax parser: expression_syntax_parser()
returned 0: FAILED
print_error_msg_ext()
return 1: parsing SUCCESS
print INFO
chk n update open/cls brace requirements: chk_cbrace_reqmnt_n_upd()
return 1 to caller


pedmas_assembler() called
expn = "a+b+c*d-f*g/h+w%d^x/y*2^l-z*q^3%e"
loop through each operator in precedence order (prob_obj attribute: operator_lst_precedence_h2l)
operator_lst_precedence_h2l = ['^', '/', '*', '%', '+', '-']
chk if operation is '%' : reqd. for purpose of proper functioning of regex
yes
no
find sections of expn with current operation: do not use escape sequence char in regex checking str
find sections of expn with current operation: use escape sequence char in regex checking str
break up each sub expn into tuples of vars creating a list of tuples
for op == '^': [(2, l), (q, 3)]
loop through the var tuple list
put tuple contents in a list
loop through the tuple's var list
pop tuple from front twice to get two operands
(a, b, c) -> var1 = a, var2 = b, tuple = (c)
all tuplesin tuple not yet processed
all vars in tuple not yet processed
create temp var name and then create replacement string from the poped vars
s not yet processed
push the newly created temp var name into front of tuple
update parser sequence event attribute of the prob_obj
replace the sub expn str with temp var name in the main expn str
update tmp_var_cntr attribute of prg_obj
return expn to calling func.: expn_seq_assembler()


:func. call from expression syntax parser
x = a+(_12b*(cfg45+12*(acd%15)/d)*100^h_ad)
count num of '(' and ')'
num '(' == num ')' == 3
chk if num '(; is equal to num of ')'
yes
no
print_error_msg_ext()
chk if there are actually any '()', ie., (num '(') > 0
yes
no
get a list of expn at the deepest level of nested parenthesis
update event cntr and dict with var and expn
clean up the expns: strip '(', ')', ' '
loop through each expn at current nested parenthesis level
find operator and prepend '//' to operator symbols as needed for regex
return 1 to expression_syntax_parser()
sub the expn and it's parenthesis with tmp var name in the main expn
update event cntr and dict with tmp var and expn
recurse the altered main expn through the expn_seq_assembler() function till all '(' n ')' have been removed


func. call from dec/assign chkr: var+expn list:
[var1= a+(b*c/d), var2 = 5]
loop for each expn
var1= a+(b*c/d)
separate var name and expn and chk if valid
var1
a+(b*c/d)
yes
no
get list of var names used in expn
[a,b,c,d]
print_error_msg_ext()
chk_if all vars previously declared
no
yes
print_error_msg_ext()
chk_if_previously initialized
no
yes
print_warning_msg()
pass to expn_seq_assembler()
return 1 to chk_var_dec_or_assignmnt()


chk_var_dec_or_assignmnt()
Int var = a+b, var2=c+d ;
Var = e+f;
regex search: returns a match
yes
no
chk_match_type()
type 0: Int var = a+b, var2=c+d ;
type 1: Var = e+f;
strip datatype token
strip trialing ';' or ' ' or '\n'
get var only n var+expn lists
var only list
var + expn list
chk match type
type 1: Int var = a+b, var2=c+d ;
type 0: Var = e+f;
return 0 to main parser seq
chk_if_previously declared
type 1: from declr line: [var, var2]
type 0 from assignment line: [var]
no
print_error_msg_ext()
yes
no
append to var only list to prog_obj var list
yes
filter out var only dec and get final list of var+expn to send
expression_syntax_parser()
return 1 to main parser seq