# Tokens Description

1. Every token consists of two parts.
   1. Constant token part
   2. Lexeme part

2. INT, CHAR, WHILE, IF, ELSE, RET, IN, OUT, ‘+’, ‘-’, ‘\*’, ‘/’, ‘{’, ‘}’, ‘(’, ‘)’, ‘,’, ‘;’, ‘:=’, ‘!’ all

have similar token i.e. (self,null) e.g. (INT,null)

1. Token of identifier is (ID,name) where ID is constant and name is the name of identifier.
2. Token of number is (NUM,num) where NUM is constant and num is the actual number

e.g. (NUM,123).

1. Token of character literal is (CL,char) where CL is constant and char is the character literal e.g. (CL,’a’).
2. Token of string literal is (STR, str) where STR is constant and str is the string literal e.g. (STR,”abc def”).
3. Token of relational operator is (RO,type) where RO is constant and type is variable
   1. Type can be LT, LE, GT, GE, EQ, NE for <, <=, >, >=, ==, != respectively
   2. Example: (RO,LT).

# Regular Expressions

## ID

letter(letter | digit) \*

## INT

digit+

## CHAR

(‘)anything(‘)

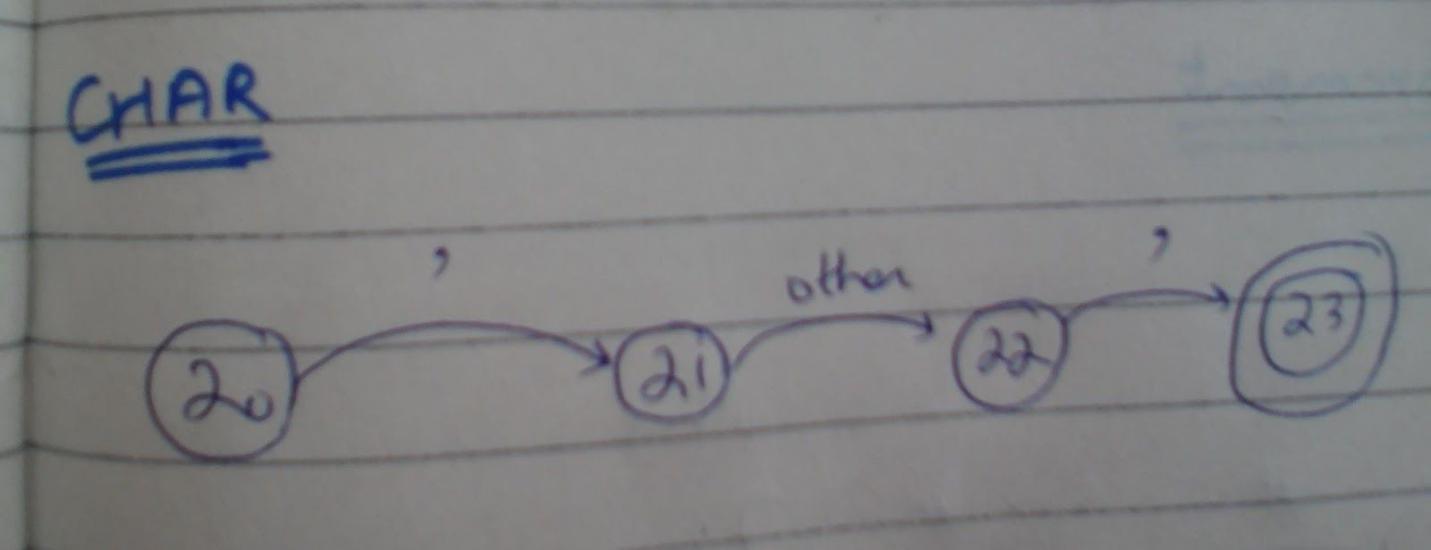
## RO

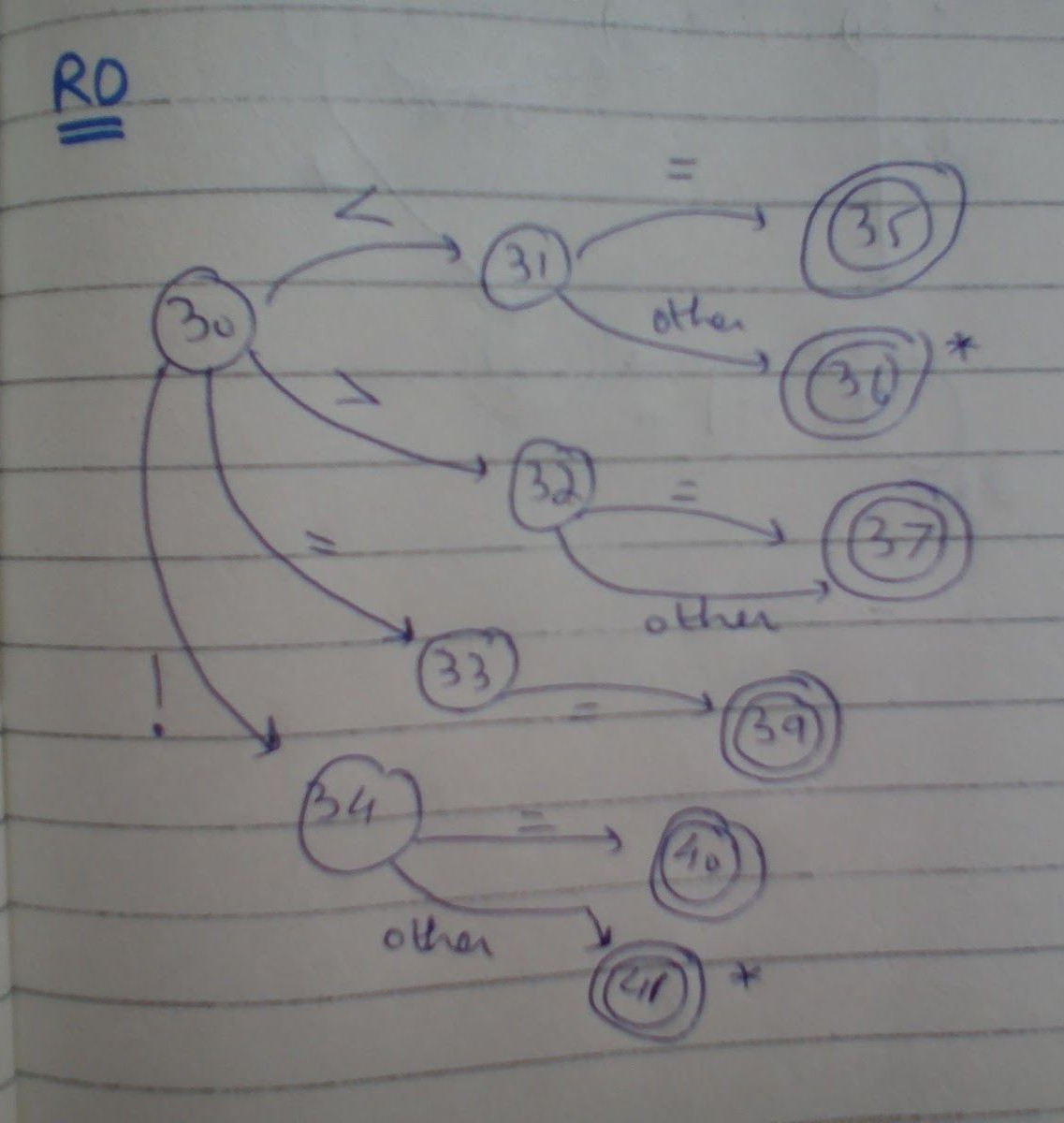
<(= | ^) + >(= | ^) + == + !(= | ^)

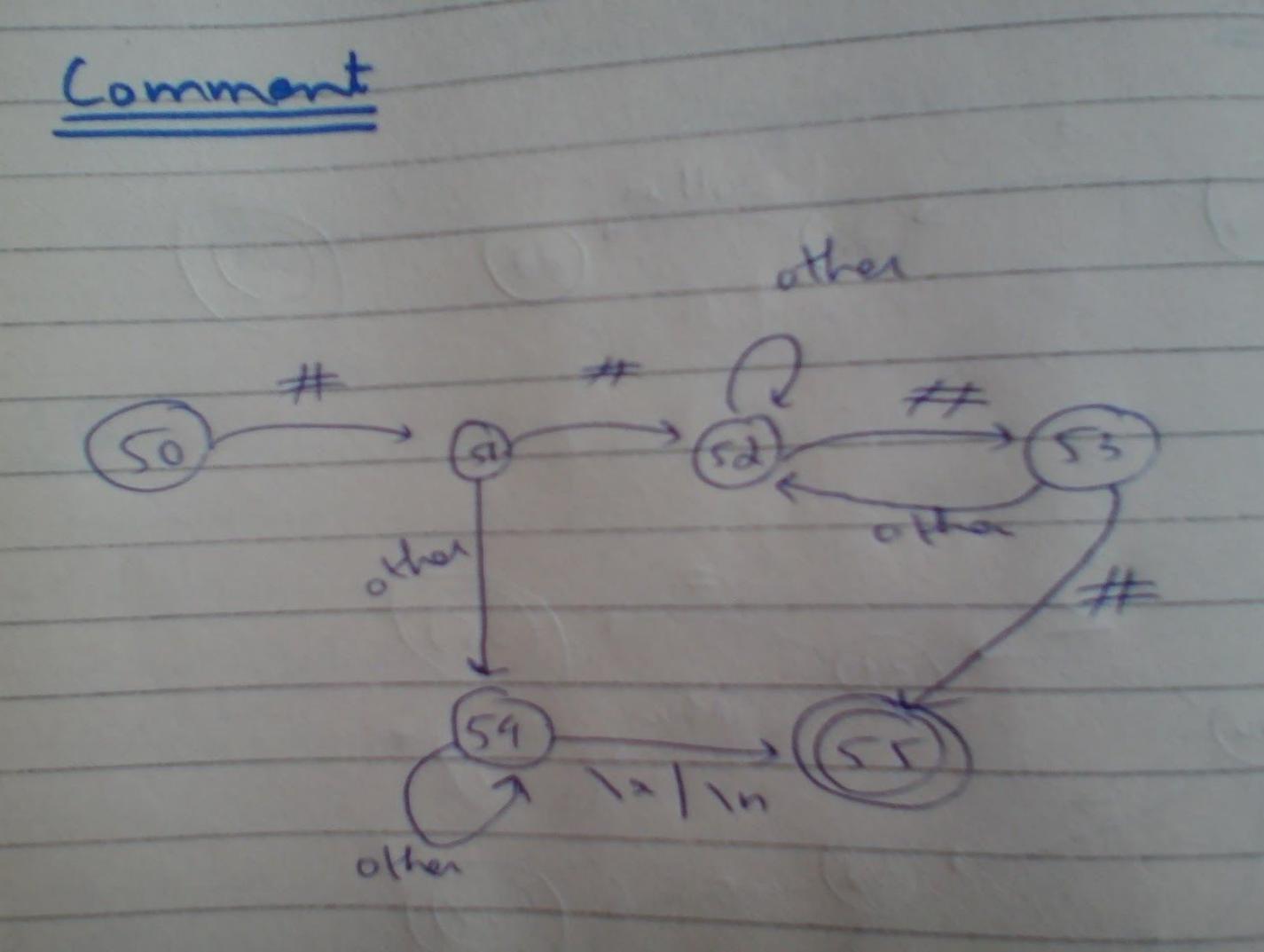
## Comment

##other\*## + #other\*( \r | \n )

# Transition Diagrams







# Context Free Grammar

S -> D S | L S | C S | I S | O S | A S

S’ -> D’’ S | L S | C S | R S | I S | O S | A S | ^

B -> { S’ }

Expression:

E -> T [E’.p = T.n]

E’ [E.n = E’.n]

T -> F [T’.p = F.n]

T’ [T.n = T’.n]

E’ -> + T [E’1.p = newTmp(); emit(E’1.p, “=”, E’.p, “+”, T.n)]

E’1  [E’.n = E’1.n]

E’ -> - T [E’1.p = newTmp(); emit(E’1.p, “=”, E’.p, “-”, T.n)]

E’1  [E’.n = E’1.n]

E’ -> ^ [E’.n = E’.p]

T’ -> \* F [T’1.p = newTmp(); emit(T’1.p, “=”, T’.p, “\*”, F.n)]

T’1  [T’.n = T’1.n]

T’ -> / F [T’1.p = newTmp(); emit(T’1.p, “=”, T’.p, “/”, F.n)]

T’1 [T’.n = T’1.n]

T’ -> ^ [T’.n = T’.p]

F -> ID [F.n = ID.lex]

F -> NUM [F.n = NUM.lex]

F -> CL [F.n = CL.lex]

F -> ( E ) [F.n = E.n]

F -> [ E ] [F.n = E.n]

In:

I -> in ID ; [emit(“in”, ID.lex)]

Out:

O -> out R ;

R -> STR [emit(“out”, STR.lex)]

R -> E [emit(“out”, E.n)]

Return:

R -> ret E ; [emit(“return”, E.n)]

Condition:

C’ -> ( E1 RO E2 ) [C’.t = n; emit(“if”, E1.n, RO.lex, E2.n, “goto”); C’.f = n, emit(“goto”)]

Conditional:

C -> if C’ [backPatch(C’.t, n); O’.f = C’.f]

B

O’

O’ -> else [O’.next = n; emit(“goto”); backPatch(O’.f, n)]

B [backPatch(O’.next, n)]

O’ -> ^ [backPatch(O’.f, n)]

Loop:

L -> while C’ [backPatch(C’.t, n)]

B [emit(“goto”, C’.t); backPatch(C’.f, n)]

Declaration Outside Function;

D -> D’ [R’’.size = D’.size] R’’

D’ -> T ID [D’.size = T.size]

T -> INT [T.size = 4]

T -> CHAR [T.size = 1]

R’’ -> ( PRD ) B

R’’ -> [O’’.size = R’’.size] O’’

O’’ -> , ID [O’’1.size = O’’.size] O’’1

O’’ -> ;

PRD -> D’ R’ | ^

R’ -> , D’ R’ | ^

Declaration Inside Function:

D’’ -> D’ [O’’.size = D’.size] O’’

Assignment or function call:

A -> ID [R’’’.p = ID.lex]

R’’’;

R’’’ -> EQ A’ [emit(R’’’.p, “=”, A’.n)]

R’’’ -> ( PRS ) [tmp = newTmp(); emit(“call”, R’’’.p, “,”, PRS.c, “,”, tmp)]

A’ -> ID ( PRS ) [A’.n = newTmp(); emit(“call”, ID.lex, “,”, PRS.c, “,”, A’.n)]

A’ -> E [A’.n = E.n]

PRS -> PR [PRS.c = PR.c]

PRS -> ^ [PRS.c = 0]

PR -> E [emit(“parm”, E.n)]

PR’ [PR.c = PR’.c + 1]

PR’ -> , PR [PR’.c = PR.c]

PR’ -> ^ [PR’.c = 0]

Terms to understand:

ID: identifier

STR: string literal

CL: character literal

RO: relational operator

EQ: equal operator