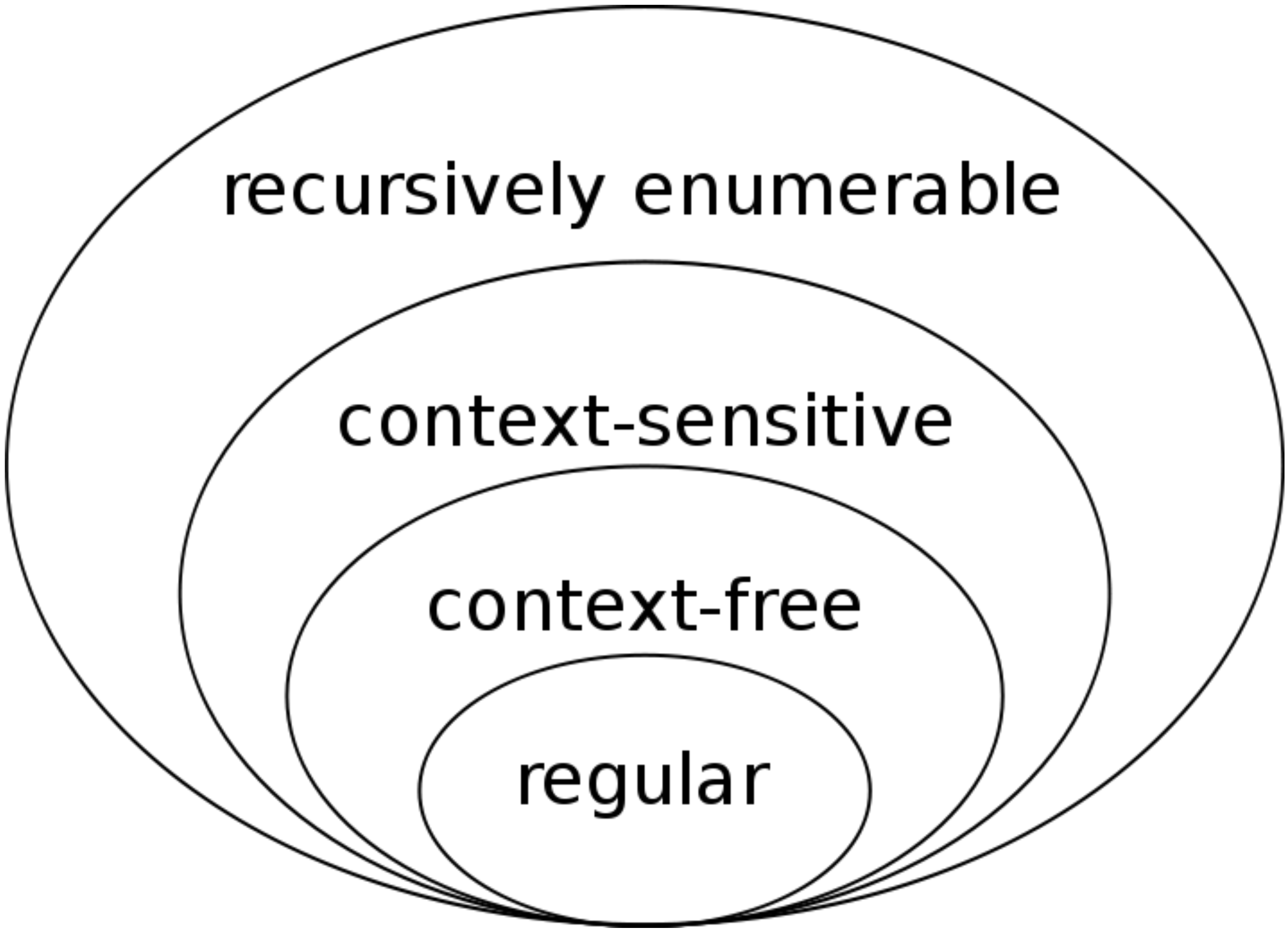
Grammer-4

COM S 319

REVIEW-1 (types of grammers)



Review-2

- Regular expressions express strings in regular language
- Regular grammer also expresses strings in regular language.
- Finite automaton is used to recognize regular expressions
- **RE, RG, FA are equivalent!**

ANTLR

- Antlr
  – tool to generate lexer and parser
  – we have seen how to build a lexer

- Antlr Commands

  antlr4 Expr.g4 // generates java code for lexer/ parser

  javac Expr*.java // compile the code

  grun Expr prog –gui (or –tree or –tokens)

  100+2*34 ^D

CONTEXT FREE GRAMMER

Restriction for Regular Grammer

- We cannot express $a^n b c^n$ using regular expressions or regular grammer i.e. the number of a's and c's must match.
- We cannot use just a finite automaton to accept $a^n b c^n$. Because, there is no way to store the "n" when we use a finite automaton. We cannot restrict it to match the a's and c's.

Do remember that regular grammer/expression/FA are identical in what they can generate/express/accept.

Context Free Grammer

Production Rules have to be of the form

$$A \rightarrow \alpha$$

where A stand for an arbitrary variable and **α** stands for a sequence of variables and terminals.

Example:

S-> aSc | abc   This generates $a^n b c^n$

- CFGs can accept $a^n b c^n$
- They can accept all regular expressions (REs).
- Not all CFGs can be accepted by REs.
- They can accept matching parentheses and braces etc. They can accept programming languages!

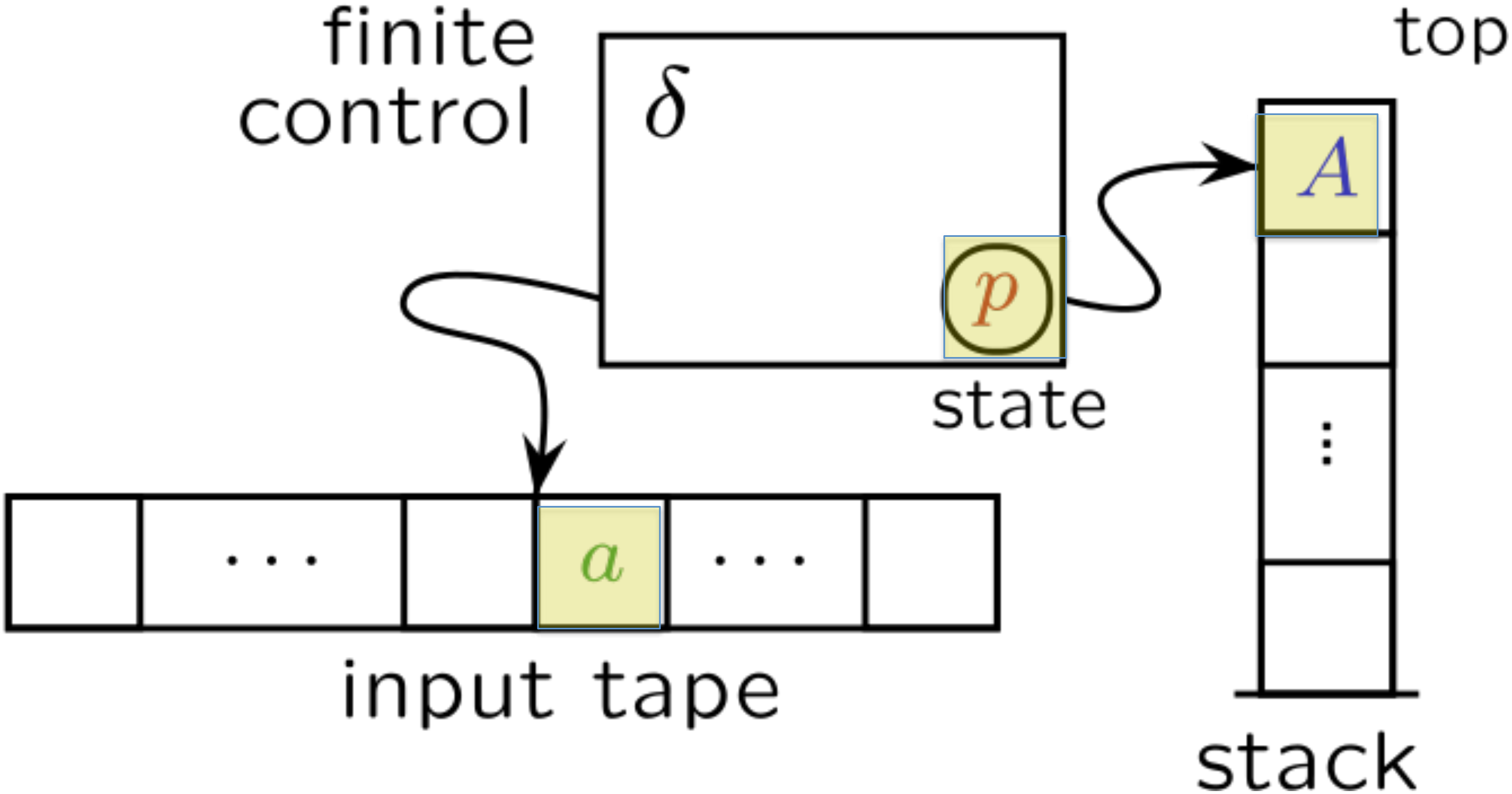
CFG: PUSH DOWN AUTOMATON

Finite Automaton has states and transitions.

Next state depends on current state and input

Push Down Automatons

Basic idea is that a stack is used to keep track of already seen open items that need to be closed later (say open parenthesis)



In addition to STATE and TRANSITION, Push Down Automaton has a STACK.

Next state depends on current state, input, AND top of stack (all highlighted in yellow)

Pushdown Automaton (PDA)

A PDA is formally defined as a 7-tuple:

- $M = (Q, A, A_S, T, S, I_S, F)$
- where
- Q is a finite set of *states*
- A is a finite set which is called the *input alphabet*
- A_S is a finite set which is called the *stack alphabet*
- T is the *transition relation*
- S is the *start state*
- I_S is the *initial stack symbol*
- F is the set of *accepting states*

Example for 0^n1^n

GRAMMAR: S-> 0S1 | ε

PUSHDOWN AUTOMATA:

Q = { p, q, r }, A = { 0, 1}, A_S = {A, Z}, S = p, I_S = Z, F = {r}

Transitions are:

<state, input, stackstate> gives <new state, new stack state>

p, 0, Z gives p, AZ (push A on stack)

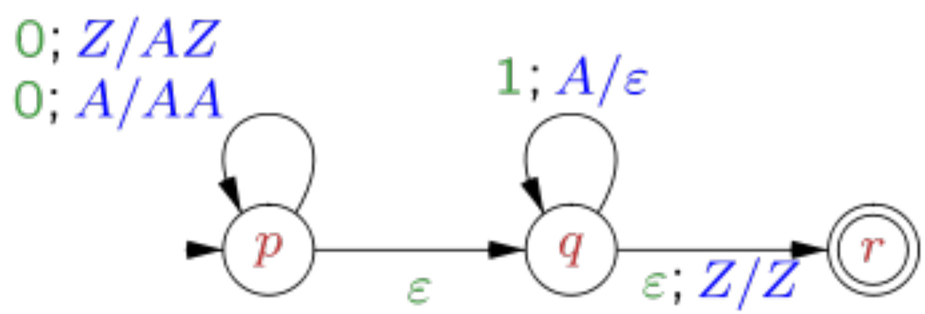
p, 0, A gives p, AA (push A on stack)

p, ε, Z gives q, Z (move state)

p, ε, A gives q, A (move state)

q, 1, A gives q, ε (pop stack)

q, ε, Z gives r, Z (move state)



In the diagram 0; Z/AZ in state p is same as p, 0, Z gives p, AZ (push A on stack)

CFG: HOW TO ACCEPT CFGS

Push Down Automatons

How to convert Context free grammar to PDA?

Q = {p,q,r}, A = { terminals of CFG}, A_S = {variables and terminals of CFG and Z}, S = p, I_S = Z, F = {r}

Transitions are:

0) <p, ε, Z> gives <q, startvariableZ>
<q, ε, Z> gives <r, Z> (accept)

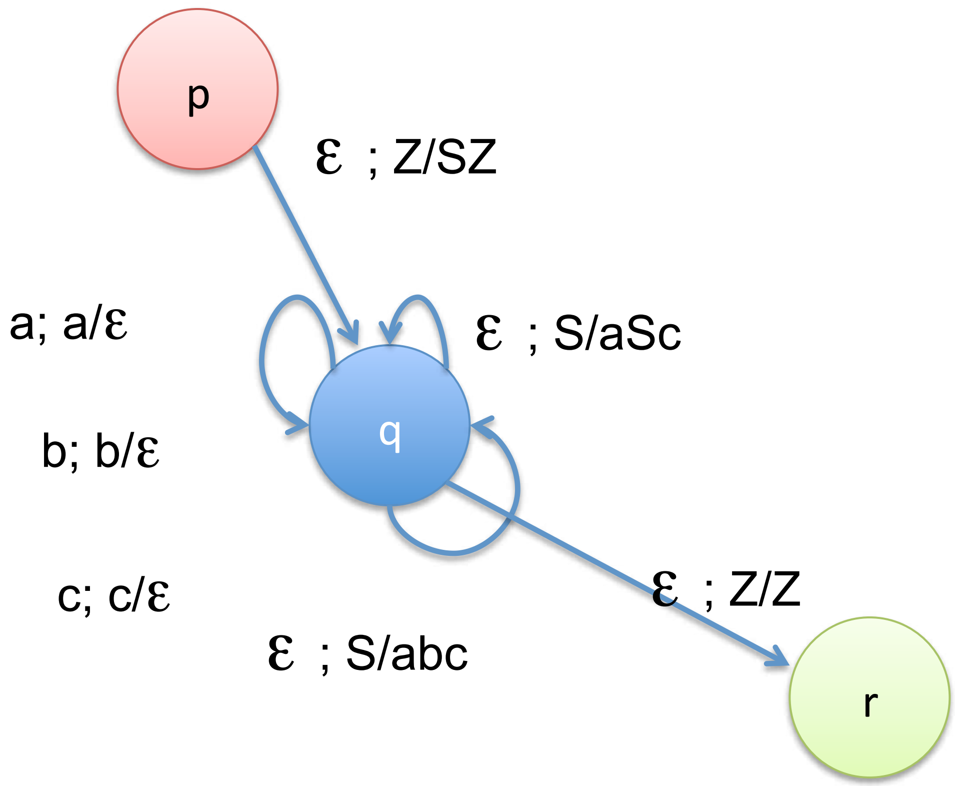
1) Expand variable rule: A → α
<q, ε, A> gives <q, α> (i.e. pop A and push α)

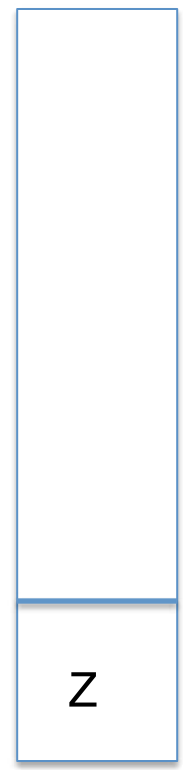
2) Match terminal rule: (all terminals)
<q, a, a> gives q, ε (i.e. pop) for each terminal a

Example

GRAMMAR: S -> aAc | abc

Pushdown Automata: (shown below)





Is aabcc in language?

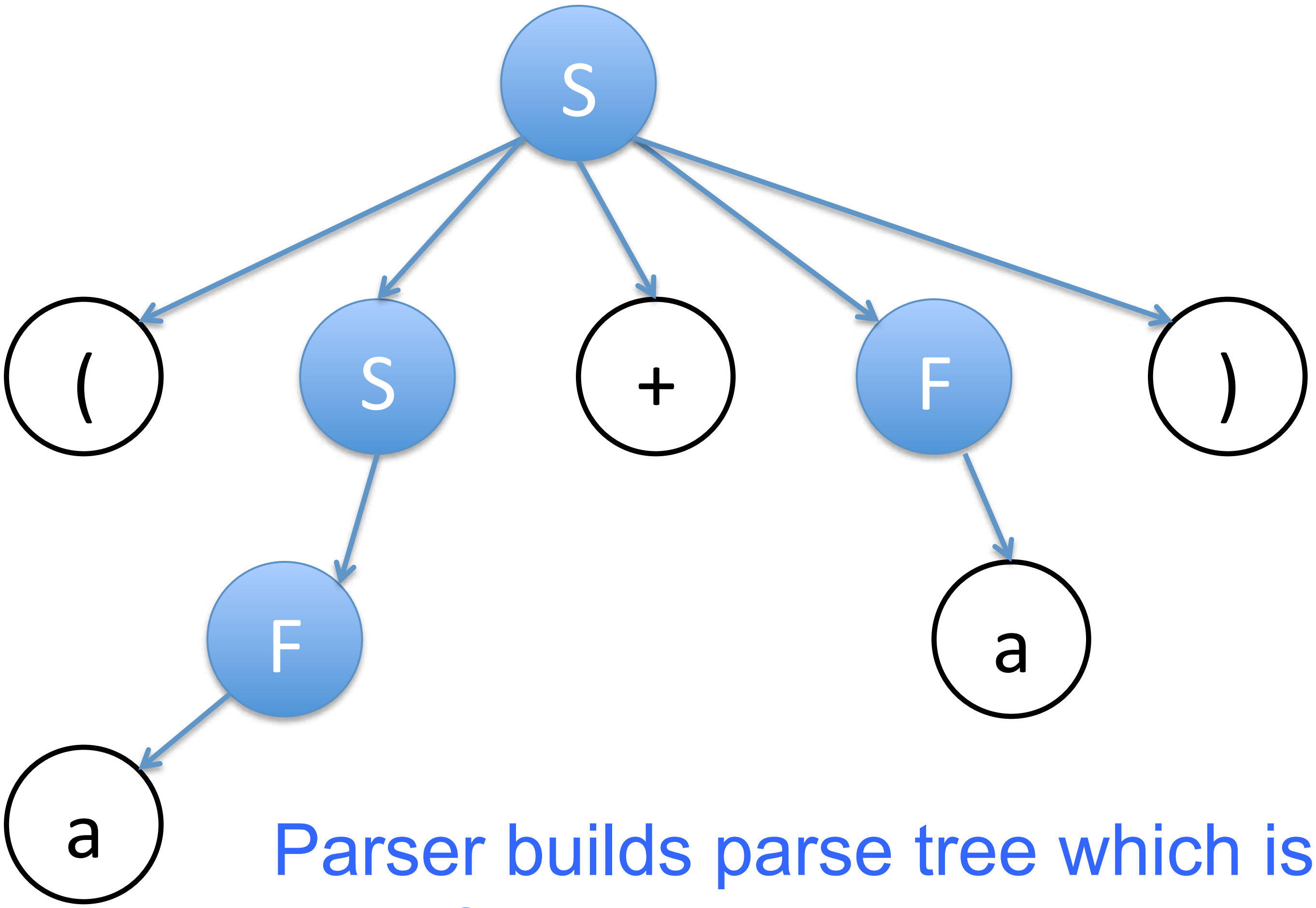
Pushdown Automaton and Context free grammers are equivalent.

i.e. CFG can generate and PDA can accept same language.

1. Create CFG and PDA for matching parentheses. Example strings: (()), (()(()()))

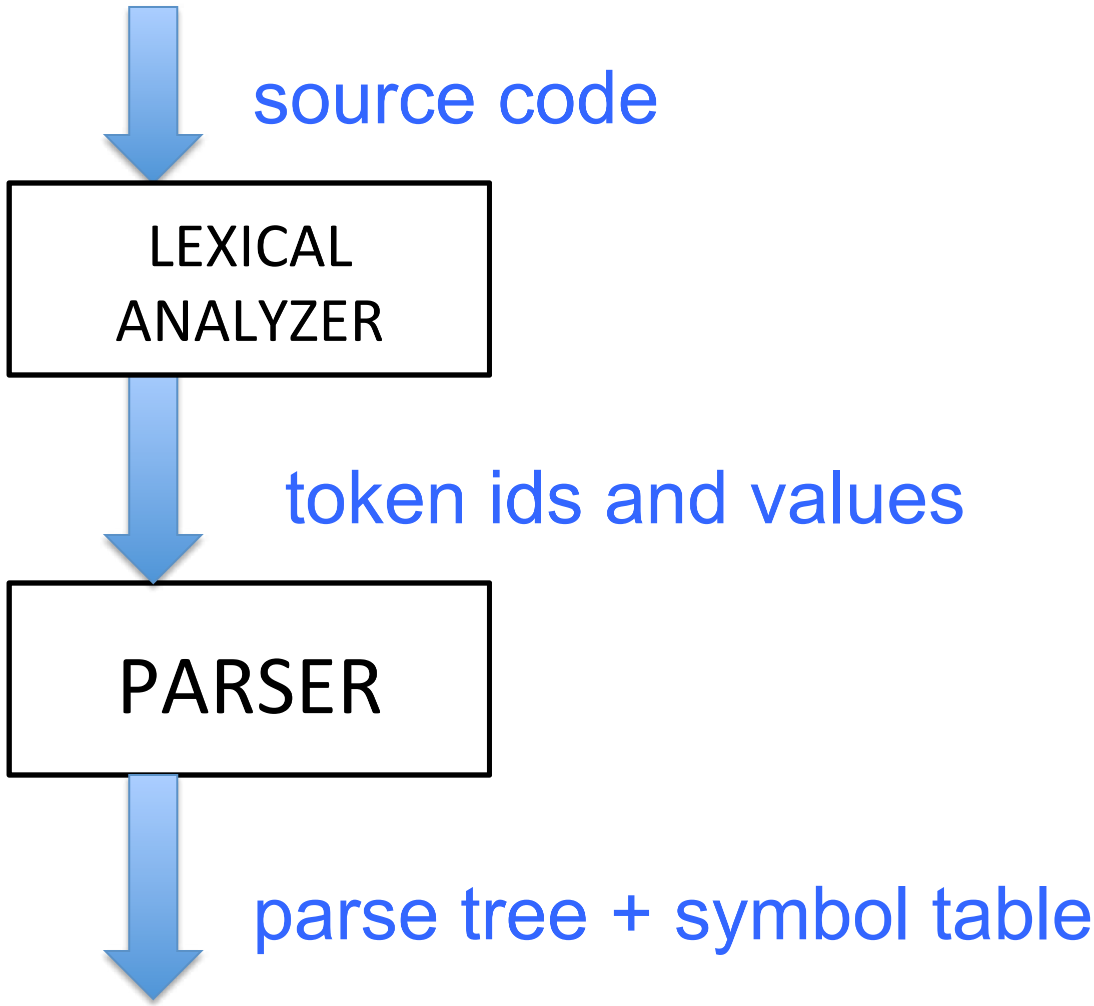
2. Given $S \rightarrow F$; $S \rightarrow (S + F)$; $F \rightarrow a$, create PDA

parse tree



Parser builds parse tree which is used for analysis by the compiler and then to generate code

first stages of a compiler



ANTLR

ANTLR

- lex and yacc – standard unix utilities to build lexer and parser (to build compilers).
  - c code
  - lexer and parser rules kept in separate files

- ANTLR
  - completely Java code.
  - Both lexer and parser rules are specified in one file.

- Antlr Commands

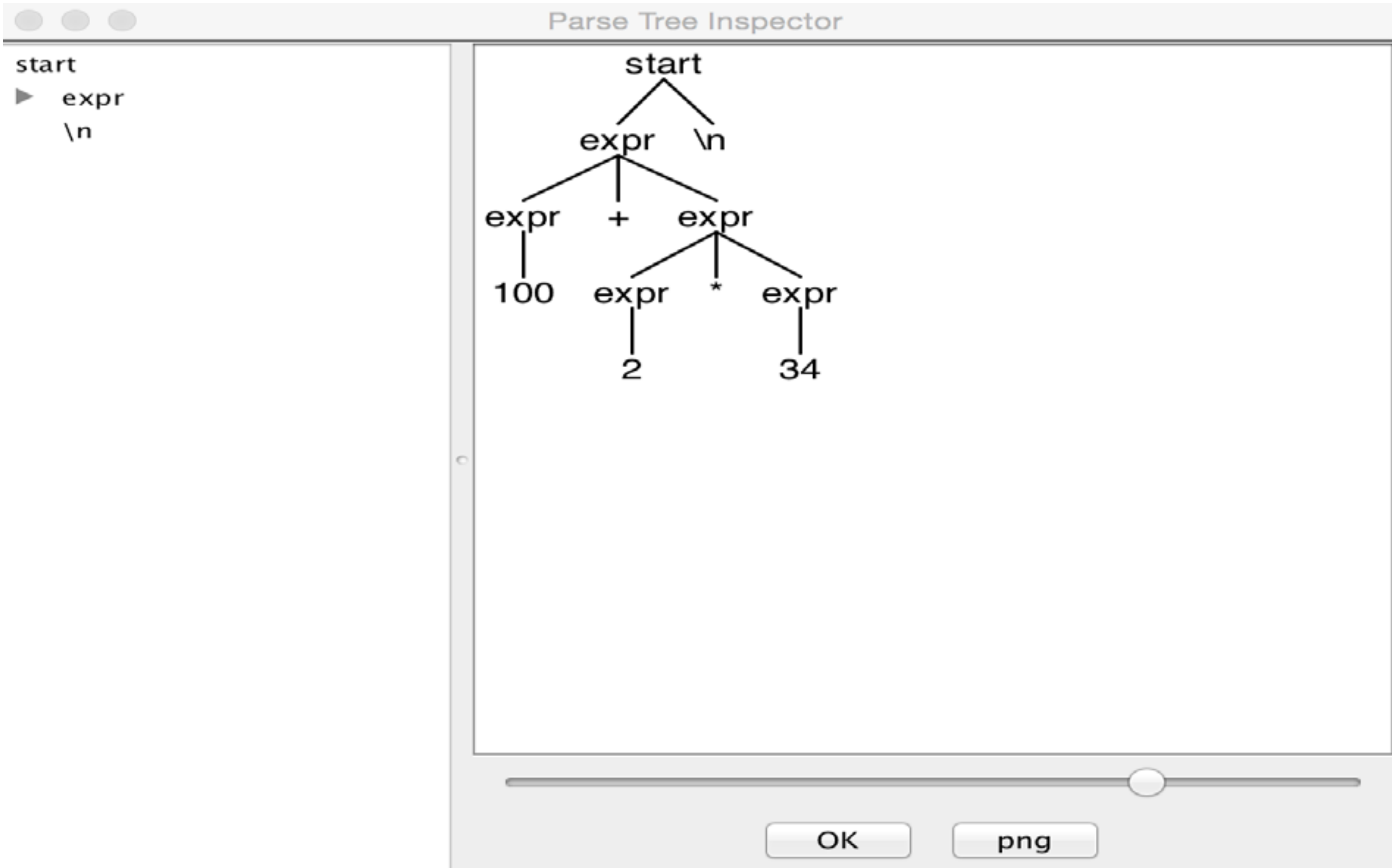
  antlr4 Expr.g4 // generates java code for lexer/parser

  javac Expr*.java // compile the code

  grun Expr prog **–gui** (or **–tree** or –tokens)

  100+2*34 ^D

using –gui option



recap – lexer rules

lexer grammar ABC;

// ***************** lexer rules

//the grammar must contain at least one lexer rule

fragment OKCHAR: [a-zA-Z0-9-.] ;

SALUTATION: 'Hello world';

ENDSYMBOL: '!' ;

recap LEXER REGEX

character	meaning	example	matches
|	logical *OR*	'a' | 'b'	either 'a' or 'b'
?	optional	'a' 'b'?	either 'ab' or 'a'
*	none or more	'a'*	nothing, 'a', 'aa', 'aaa', ...
+	once or more	'a'+	'a', 'aa', 'aaa', ...
~	negation	~('a' | 'b')	any character (in the range \u0000..\uFFFF) except 'a' and 'b'
(...)	grouping	('a' 'b')+	'ab', 'abab', 'ababab', ...

Parser rules: HelloWorld.g4

```
grammar HelloWorld;  // notice NO lexer keyword

// ***************** lexer rules:
SALUTATION: 'Hello world';
ENDSYMBOL: '!' ;

// ***************** parser rules:
//our grammer accepts only salutation followed by an
//end symbol.  non-terminal is in lowercase.
expression : SALUTATION ENDSYMBOL;
```

```
grammar Count;

@members {
        int count = 0;
}

// PARSE RULE
start
        @after {System.out.println("Total ints Count is
" + count);}
        : INT {count++; } ( ',' INT   {count++;} )*
        ;

// LEXER RULE
INT : [0-9]+ ;
WS : [ \r\t\n]+ {skip();} ;
```

NOTE CODE SEGMENTS ALLOWED