Grammer-3

COM S 319

REVIEW-1 (grammer)

- Grammer is formally defined as follows. A grammer G is a four tuple { V, T, P, S} where V and T are finite sets of variables and terminals (or symbols). V and T are disjoint. P is a finite set of production rules. S is a special variable called the start symbol.
- Example:

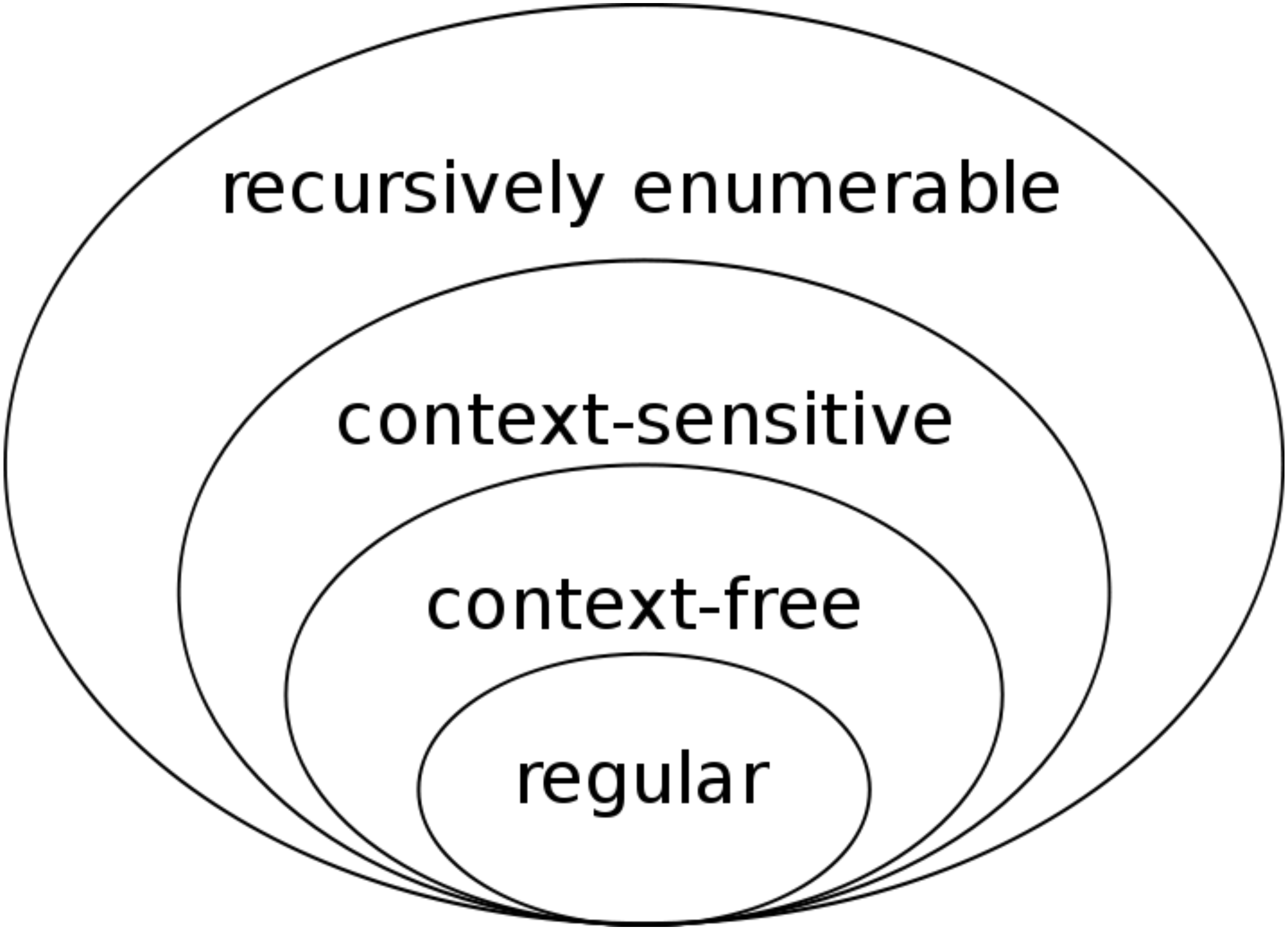
G1 = {V, T, P, S} where V = {E}, T = {+, -, (,), id},

S = E, P = rules below

(rule1) E → E + E, (rule2) E → E - E,

(rule3) E → (E), (rule4) E → id

REVIEW-2 (types of grammers)



REVIEW-3 Regular Grammer

- Production Rules have to have one of the forms
  1. A → a
  2. A → aB
  3. A → ε

where A and B stand for arbitrary variables and a stands for an arbitrary terminal (could also be empty). Epsilon is the empty string.

Note: There is an equivalent form, where middle rule is A → Ba

Review-4

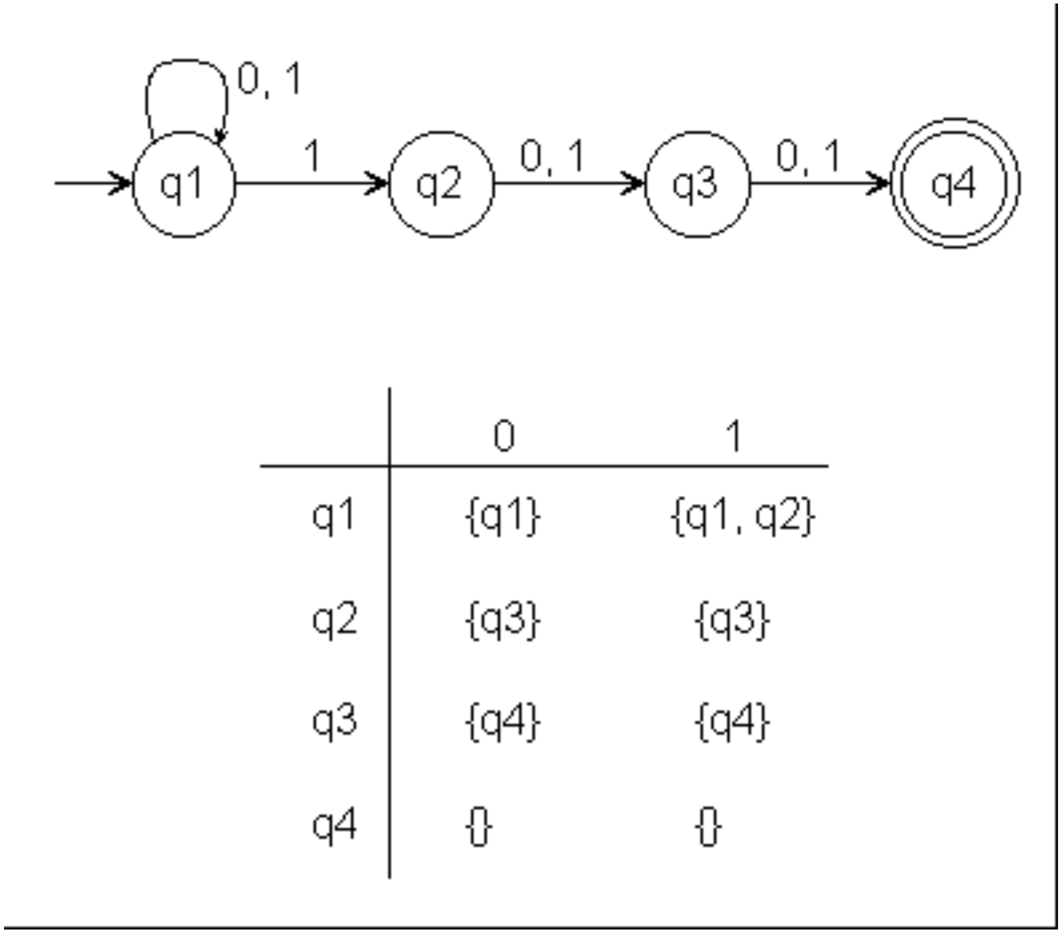
- Regular expressions express strings in regular language
- Regular grammer also expresses strings in regular language.
- Finite automaton is used to recognize regular expressions
- **RE, RG, FA are equivalent!**

Review-5 Non-deterministic FA

- A NFA is a 5-tuple (Q, A, T, S, F)
  - Q is a FINITE set of states
  - A is the alphabet
  - T is the transition function
    - Q x A+ε → P(Q) (i.e. state & alphabet gives state
  - S is the start state
  - F is set of final states
- NFA (non-deterministic finite automaton) can
  1. transition to multiple states on the same input and
  2. can also transition on epsilon

easier to express in NFA vs DFA (note DFA and NFA are equivalent)

Review-6: Transition table



	0	1
q1	{q1}	{q1, q2}
q2	{q3}	{q3}
q3	{q4}	{q4}
q4	{}	{}

LEX (LEXER OR LEXICAL ANALYSER)

lex & yacc

- describe rules for language in a file
- lex automatically generates lexical analyzer.
- yacc generates parser.

Format of lex rules file:

{definitions}

%%

{rules}

%%

{user subroutines}

Example Lex file

```
%{
#include <stdio.h>
%}

%%
[a-zA-Z][a-zA-Z0-9]*    printf("WORD ");
[a-zA-Z0-9\/.-]+        printf("FILENAME ");
\"                      printf("QUOTE ");
\{                      printf("OBRACE ");
\}                      printf("EBRACE ");
;                       printf("SEMICOLON ");
\n                      printf("\n");
[ \t]+                  /* ignore whitespace */;
%%
```

ANTLR

http://www.antlr.org

ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees.

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.

Example: ABC.g4

```
lexer grammar ABC;

options
{
  // antlr will generate java lexer and parser
  language = Java;  // JavaScript
}

// ***************** lexer rules
//the grammar must contain at least one lexer rule

SALUTATION: ('Hello world');
ENDSYMBOL: '!' ;
```

Using ANTLR

1. Auto-generate java/javascript etc code

   ```
   antlr4 ABC.g4          // which is short cut for
                   java -jar ../antlr-4.4-complete.jar ABC.g4
   ```

2. Compile java code

   ```
   javac ABC*.java
   ```

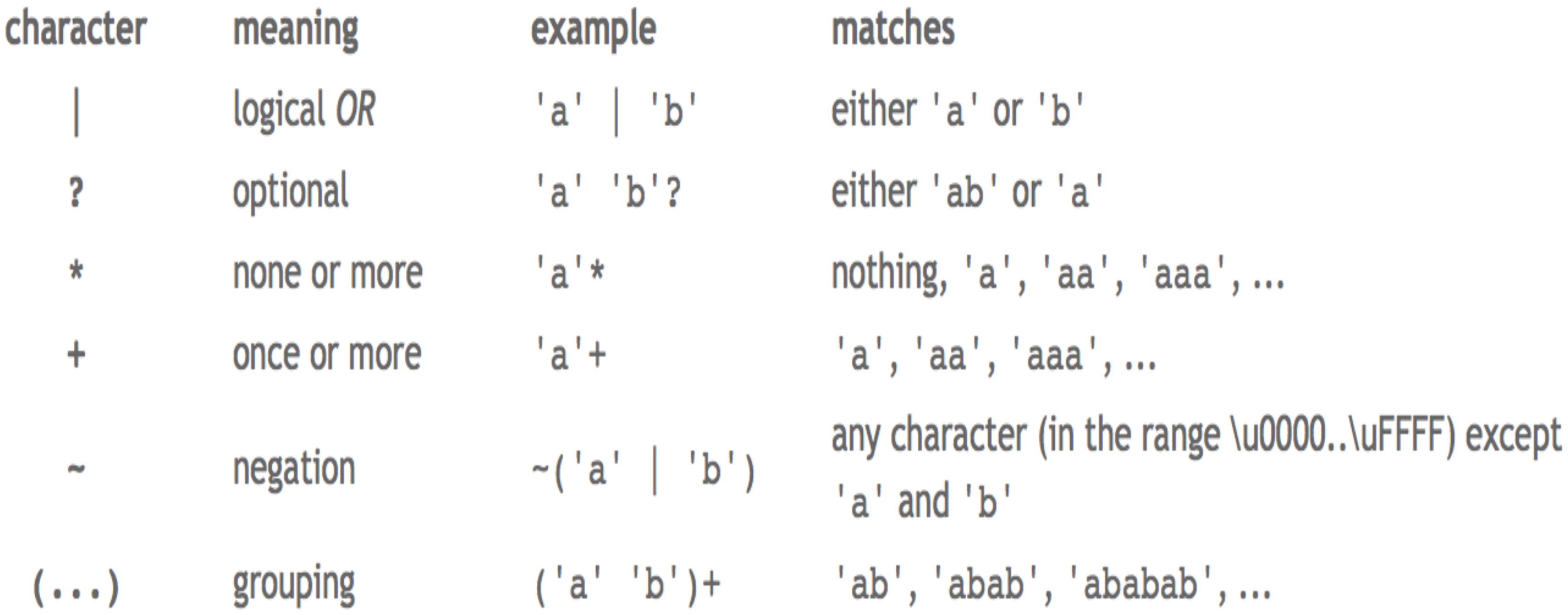
   (** do u know about path and classpath variables)

3. Run the built in test runner code

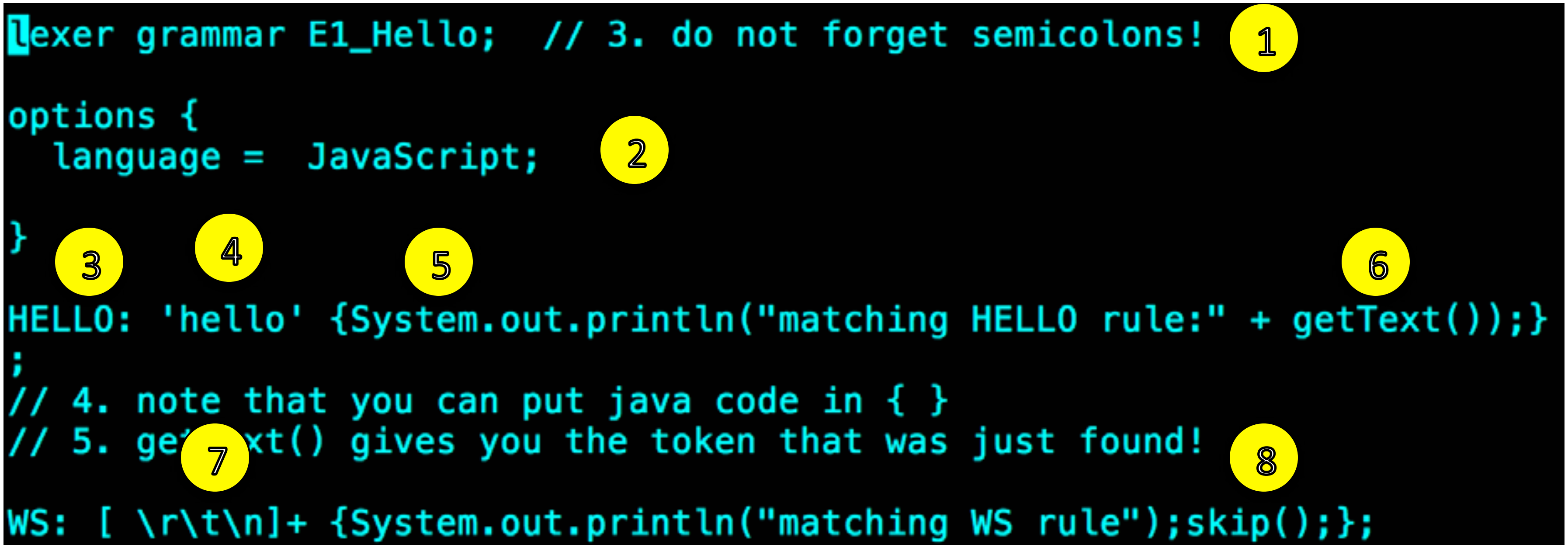
   ```
   grun ABC  tokens < input  // which is short cut for
   java org.antlr.v4.runtime.misc.TestRig ABC tokens < input
   ```

LEXER RULES

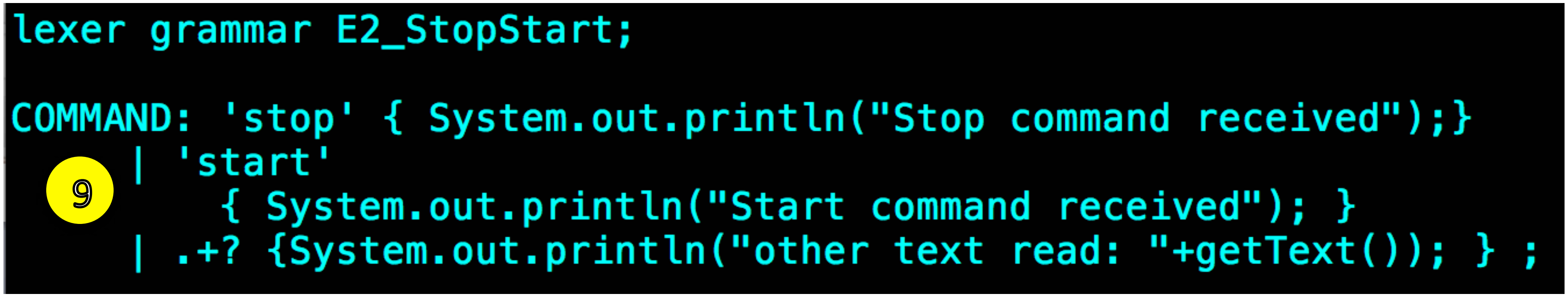
- RULE_NAME : RULE_CONTENTS ;



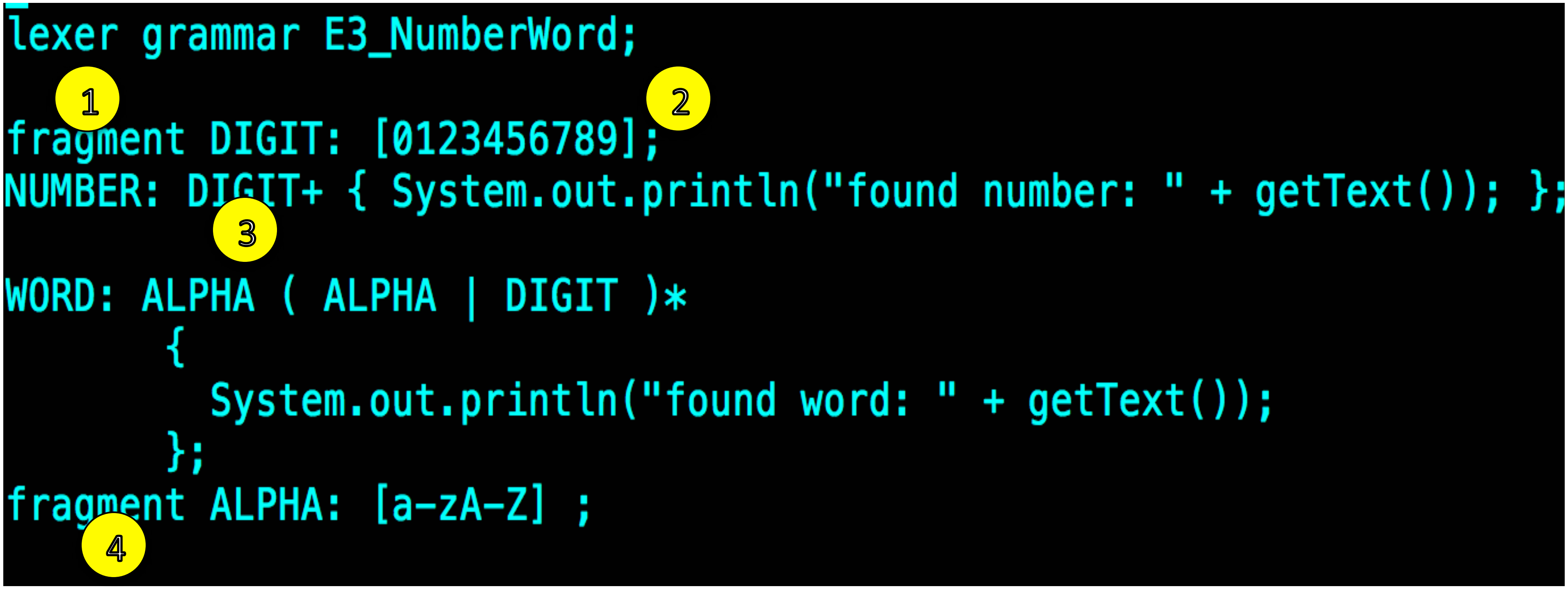
character	meaning	example	matches
|	logical *OR*	'a' | 'b'	either 'a' or 'b'
?	optional	'a' 'b'?	either 'ab' or 'a'
*	none or more	'a'*	nothing, 'a', 'aa', 'aaa', ...
+	once or more	'a'+	'a', 'aa', 'aaa', ...
~	negation	~('a' | 'b')	any character (in the range \u0000..\uFFFF) except 'a' and 'b'
(...)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



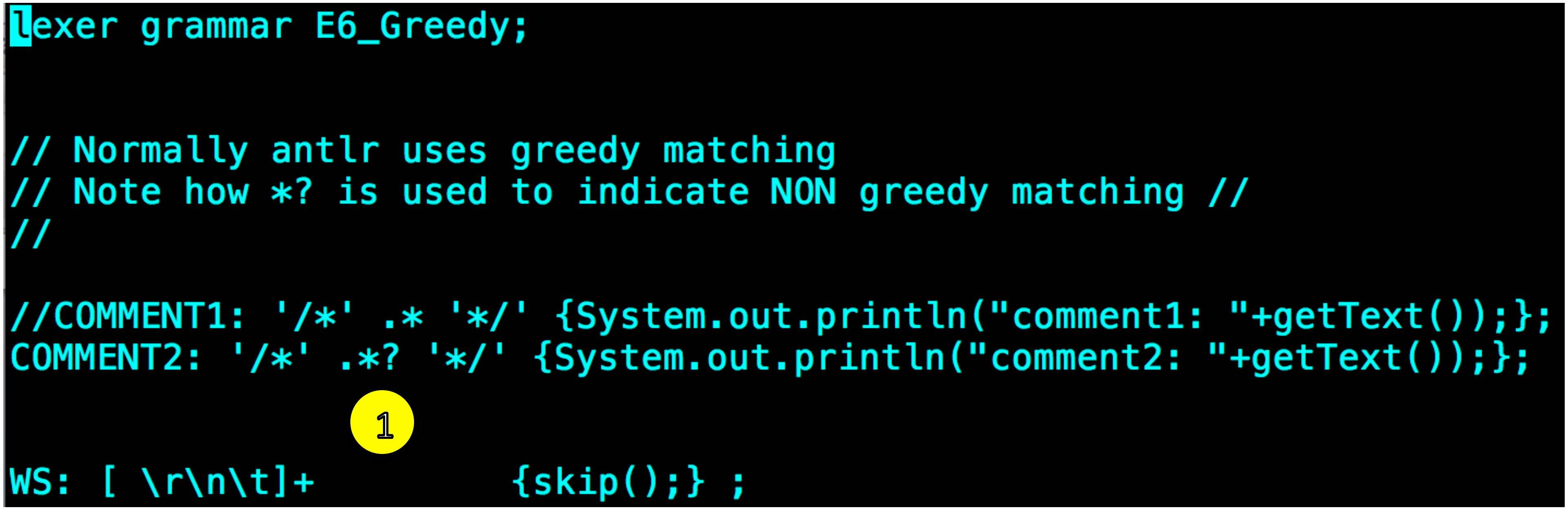
```
lexer grammar E1_Hello;  // 3. do not forget semicolons!

options {
  language =  JavaScript;

}

HELLO: 'hello' {System.out.println("matching HELLO rule:" + getText());}
;
// 4. note that you can put java code in { }
// 5. getText() gives you the token that was just found!

WS: [ \r\t\n]+ {System.out.println("matching WS rule");skip();};
```



```
lexer grammar E2_StopStart;

COMMAND: 'stop' { System.out.println("Stop command received");}
       | 'start'
           { System.out.println("Start command received"); }
       | .+? {System.out.println("other text read: "+getText()); } ;
```

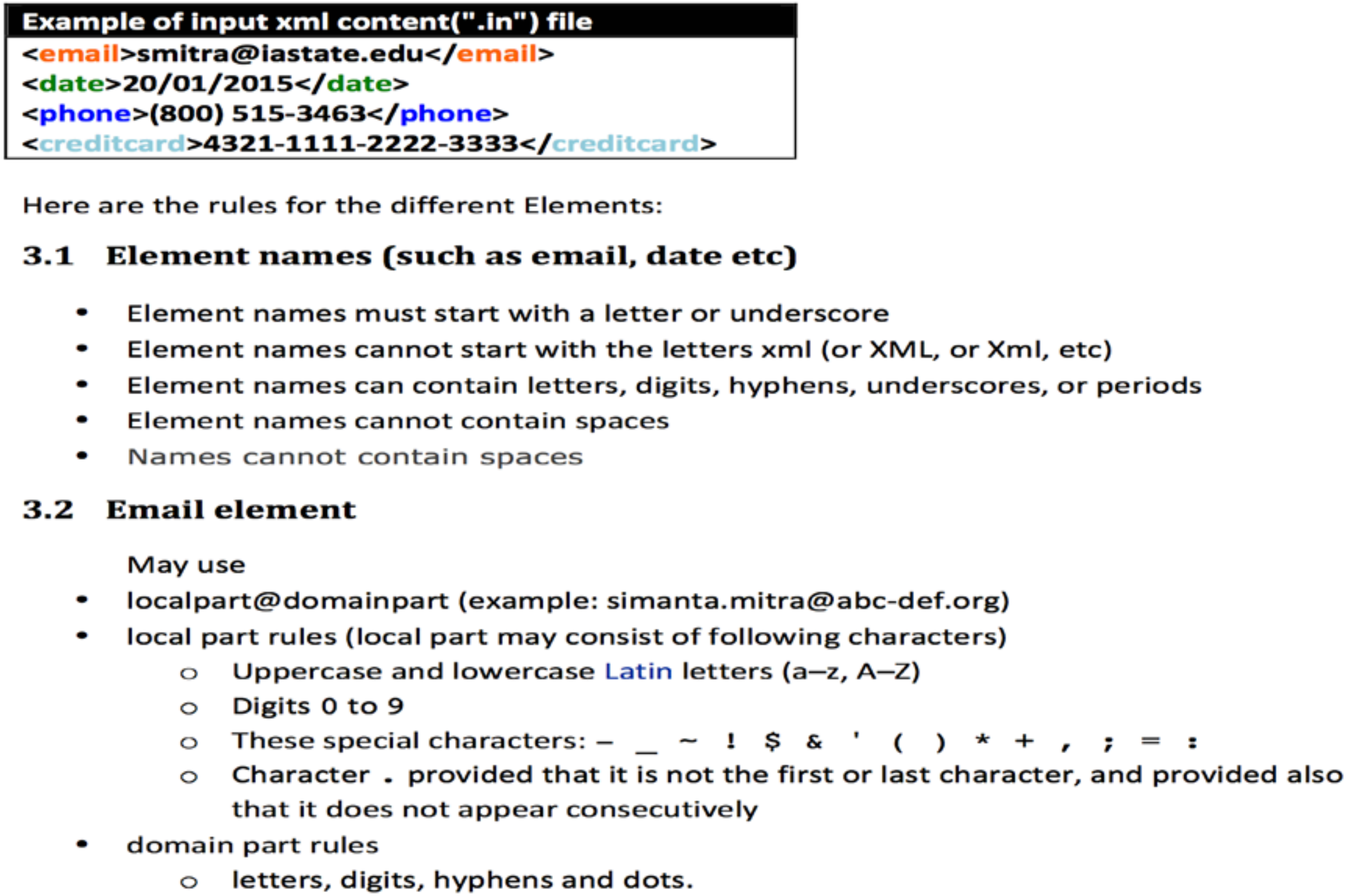


```
lexer grammar E3_NumberWord;

fragment DIGIT: [0123456789];
NUMBER: DIGIT+ { System.out.println("found number: " + getText()); };

WORD: ALPHA ( ALPHA | DIGIT )*
       {
         System.out.println("found word: " + getText());
       };
fragment ALPHA: [a-zA-Z] ;
```



```
lexer grammar E6_Greedy;


// Normally antlr uses greedy matching
// Note how *? is used to indicate NON greedy matching //
//

//COMMENT1: '/*' .* '*/' {System.out.println("comment1: "+getText());};
COMMENT2: '/*' .*? '*/' {System.out.println("comment2: "+getText());};


WS: [ \r\n\t]+          {skip();} ;
```

Self Check Write g4 for email.



Example of input xml content(".in") file

```
<email>smitra@iastate.edu</email>
<date>20/01/2015</date>
<phone>(800) 515-3463</phone>
<creditcard>4321-1111-2222-3333</creditcard>
```

Here are the rules for the different Elements:

3.1 Element names (such as email, date etc)

- Element names must start with a letter or underscore
- Element names cannot start with the letters xml (or XML, or Xml, etc)
- Element names can contain letters, digits, hyphens, underscores, or periods
- Element names cannot contain spaces
- Names cannot contain spaces

3.2 Email element

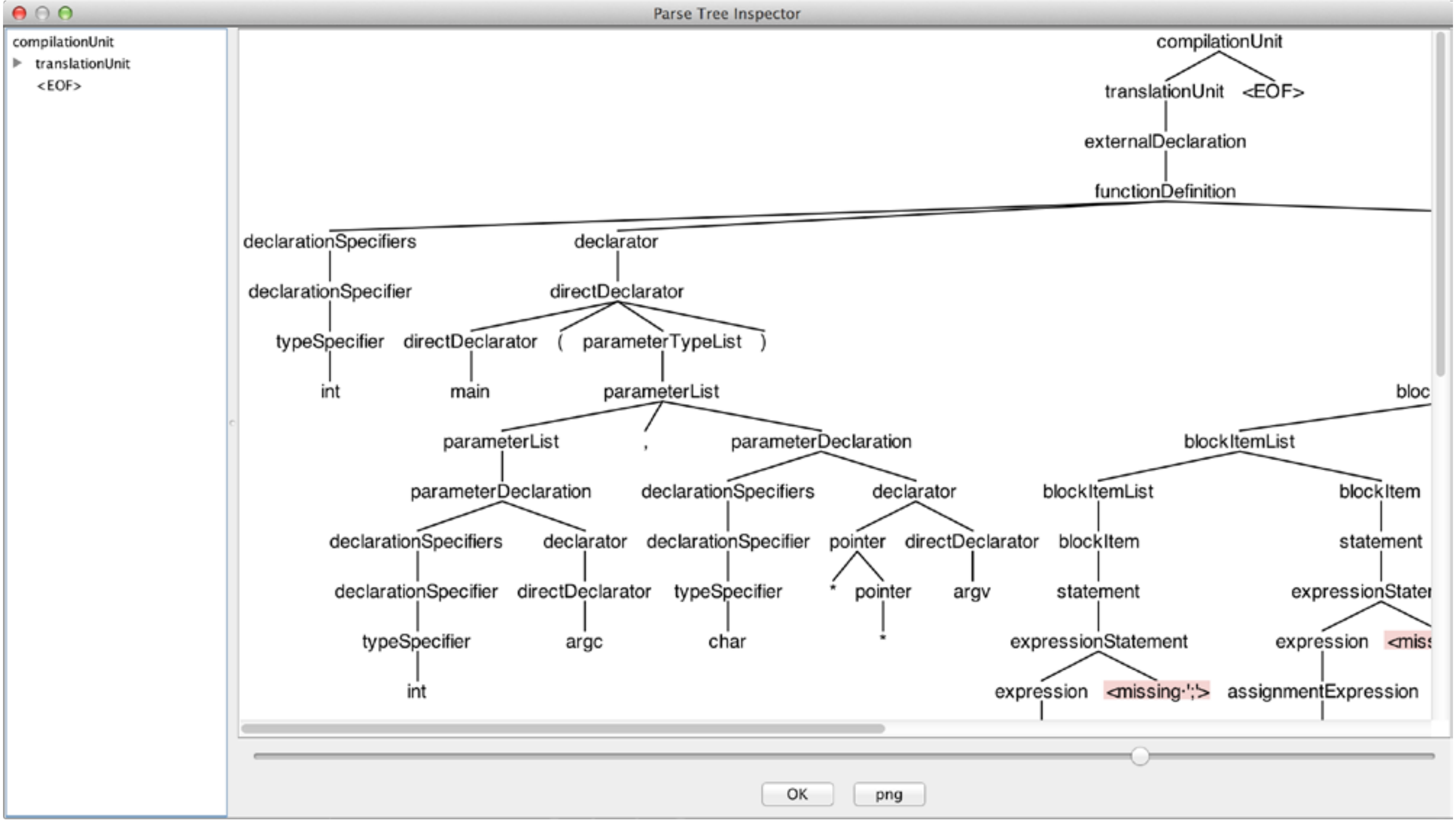
May use

- localpart@domainpart (example: simanta.mitra@abc-def.org)
- local part rules (local part may consist of following characters)
  - Uppercase and lowercase Latin letters (a–z, A–Z)
  - Digits 0 to 9
  - These special characters: – _ ~ ! $ & ' ( ) * + , ; = :
  - Character . provided that it is not the first or last character, and provided also that it does not appear consecutively
- domain part rules
  - letters, digits, hyphens and dots.

Parser rules: HelloWorld.g4

```
grammar HelloWorld;

options
{
  // antlr will generate java lexer and parser
  language = Java;
}

// ***************** lexer rules:
//the grammar must contain at least one lexer rule
SALUTATION: ('Hello world');
ENDSYMBOL: '!' ;

// ***************** parser rules:
//our grammer accepts only salutation followed by an end symbol
expression : SALUTATION ENDSYMBOL;
```

using –gui option


Parse Tree Inspector
compilationUnit
translationUnit
<EOF>
compilationUnit
translationUnit
<EOF>
externalDeclaration
functionDefinition
declarationSpecifiers
declarator
declarationSpecifier
directDeclarator
typeSpecifier
directDeclarator
(
parameterTypeList
)
int
main
parameterList
bloc
parameterList
,
parameterDeclaration
blockItemList
parameterDeclaration
declarationSpecifiers
declarator
blockItemList
blockItem
declarationSpecifiers
declarator
declarationSpecifier
pointer
directDeclarator
blockItem
statement
declarationSpecifier
directDeclarator
typeSpecifier
*
pointer
argv
statement
expressionStater
typeSpecifier
argc
char
*
expressionStatement
expression
<miss
int
expression
<missing ';'>
assignmentExpression
OK
png