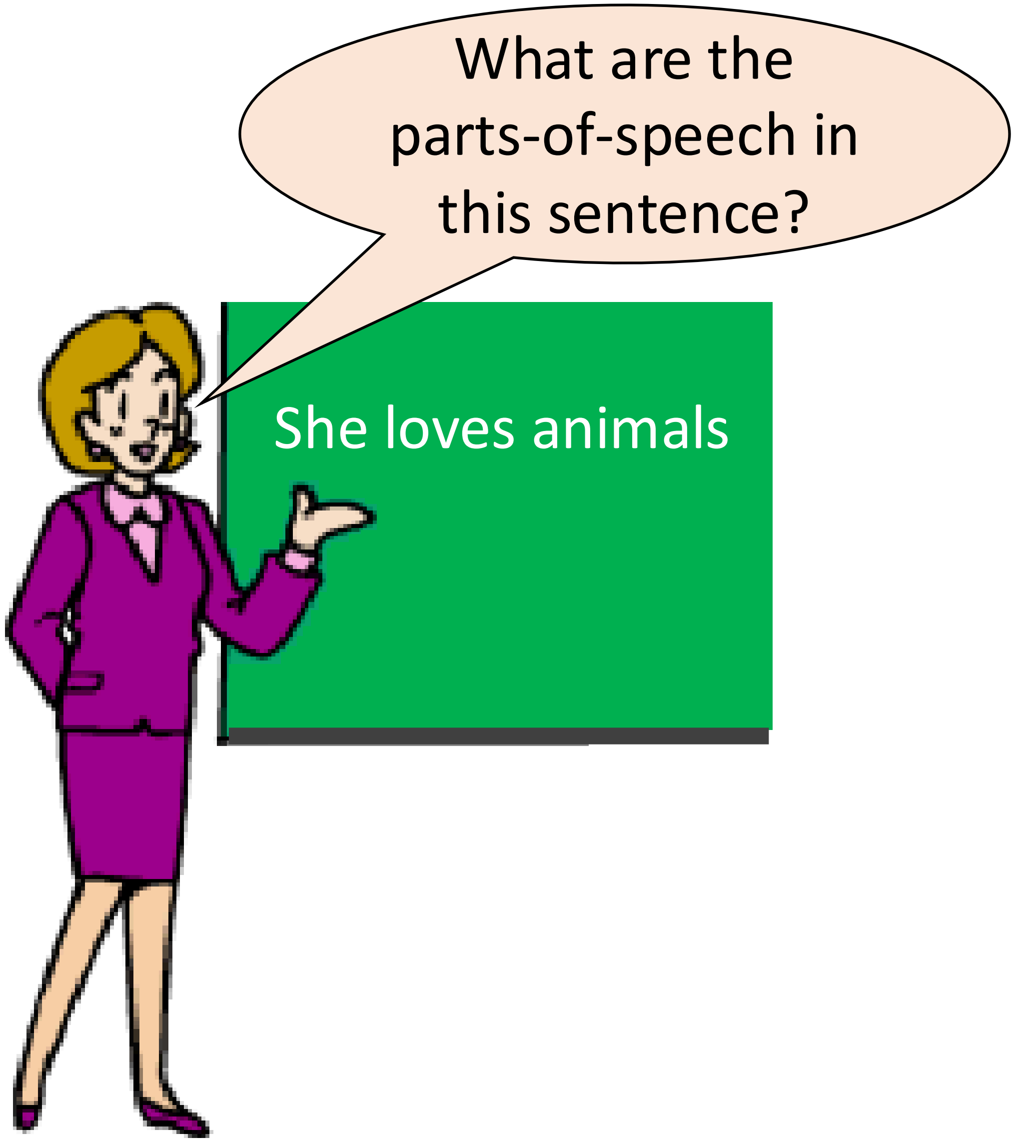
ANTLR

(ANother Tool for Language Recognition)

Approved for Public Release; Distribution Unlimited. Case Number 15-2978. The author's affiliation with The MITRE Corporation is provided for identification purposes only, and is not intended to convey or imply MITRE's concurrence with, or support for, the positions, opinions or viewpoints expressed by the author

Roger L. Costello
January 9, 2016

Remember grammar classes in grade school?



First, you broke up the sentence into parts (words)

She loves animals

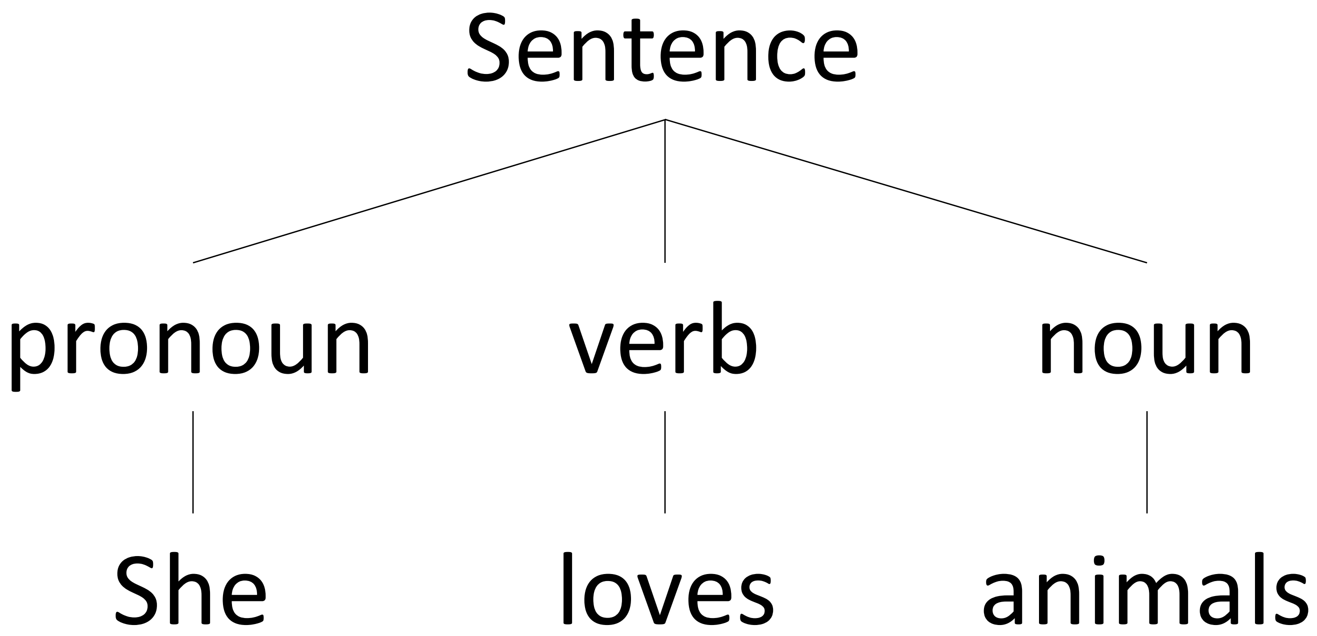
whitespace demarcates the parts

Second, you identified each part's type

She loves animals

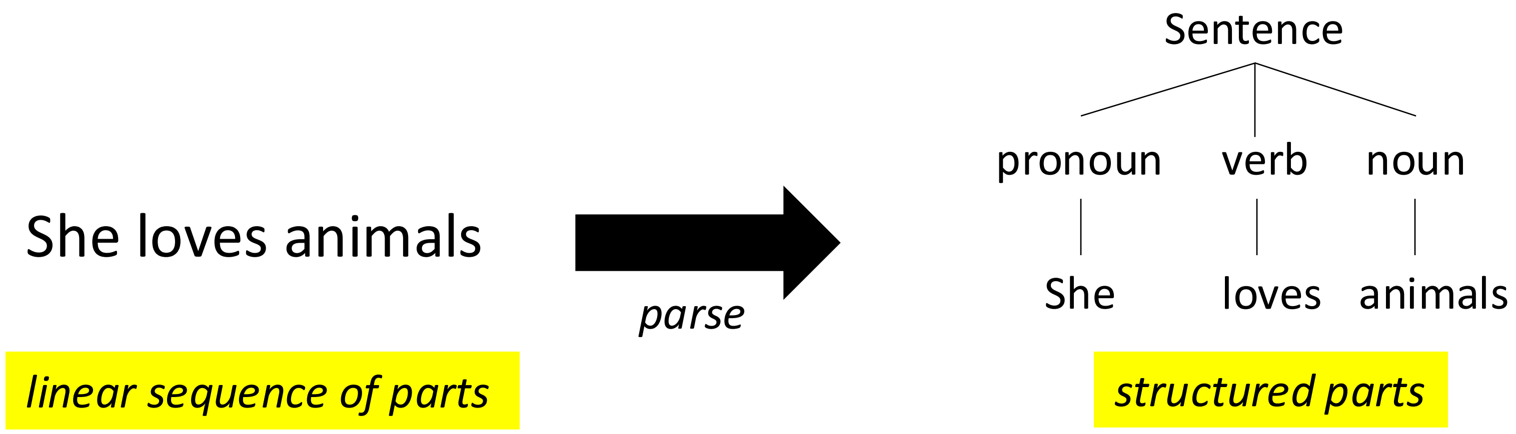
pronoun verb noun

Third, you diagrammed the sentence



That's parsing!

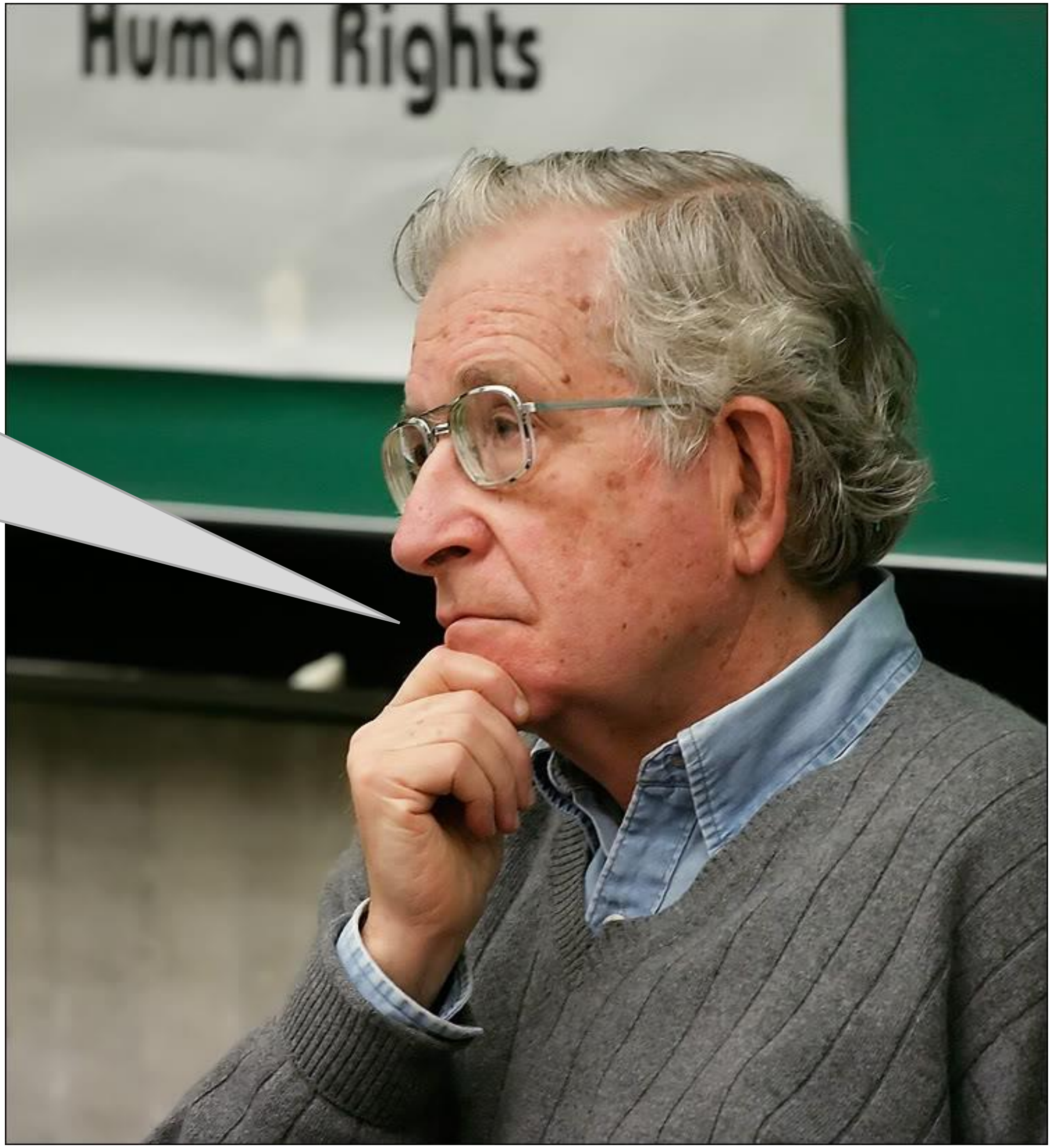
Parsing is nothing but structuring a linear sequence of parts



Parsing is nothing but structuring a linear sequence of parts

But don't think that is trivial or unimportant

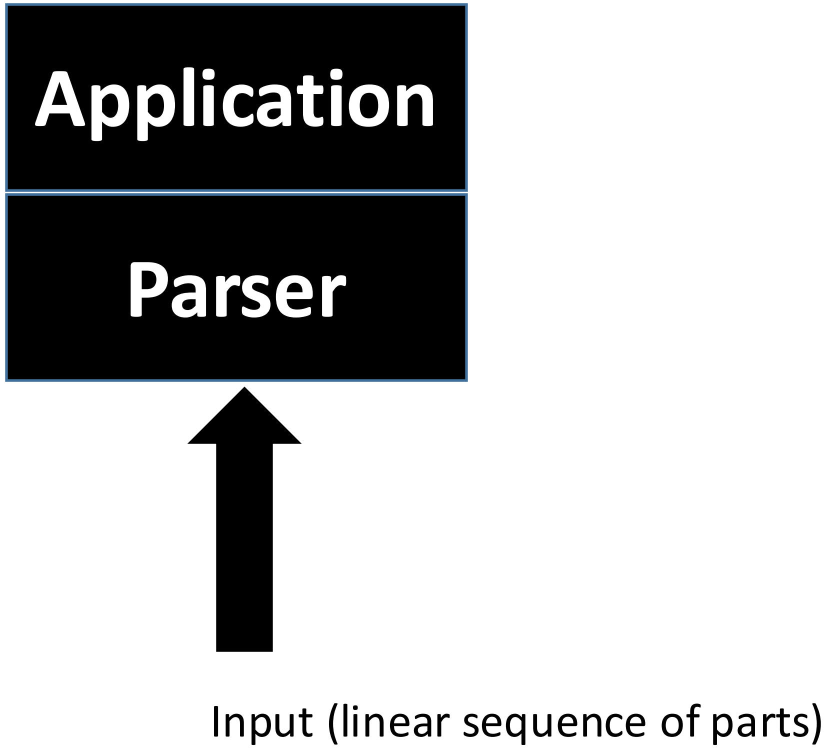
In our brain we automatically convert a linear sequence of parts into a parse tree in order to understand.

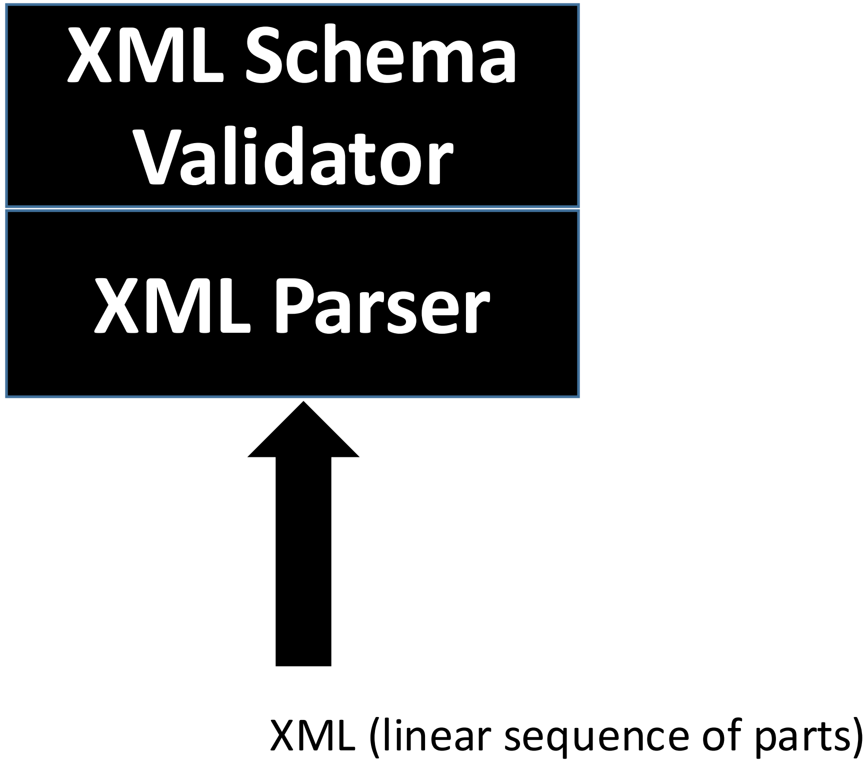


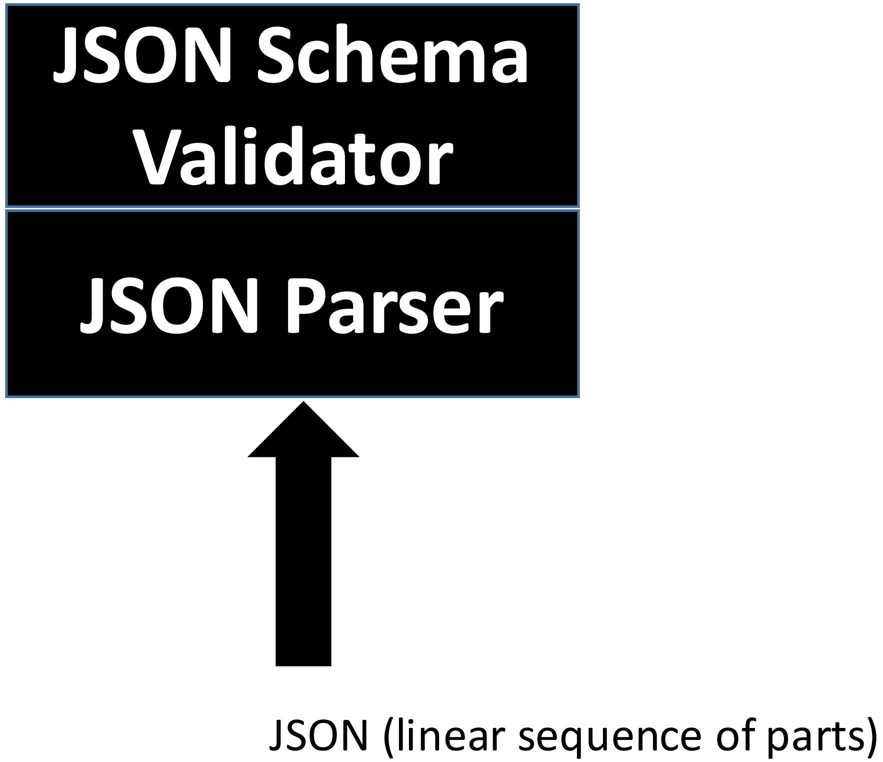
Noam Chomsky (linguist)

I am paraphrasing Chomsky.

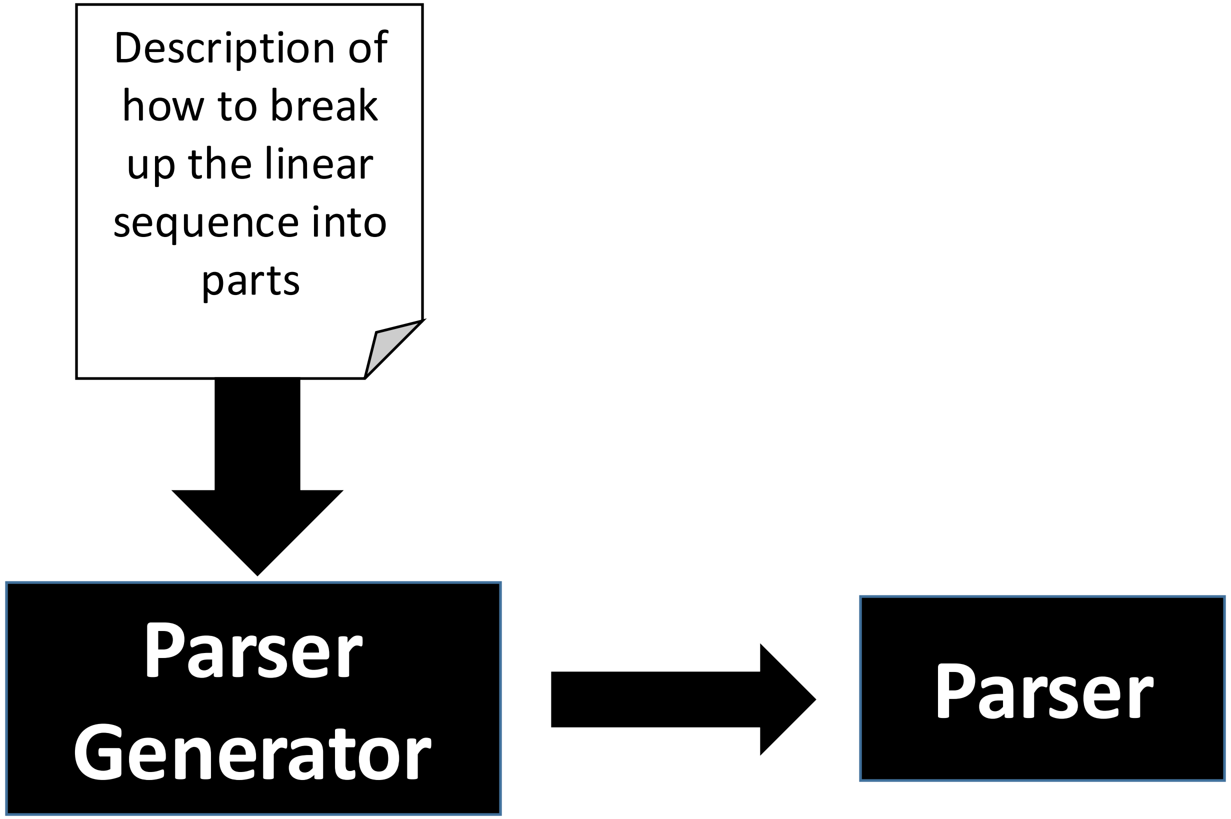

She loves animals
linear sequence of parts
Parser
Sentence
pronoun
verb
noun
She
loves
animals
structured parts

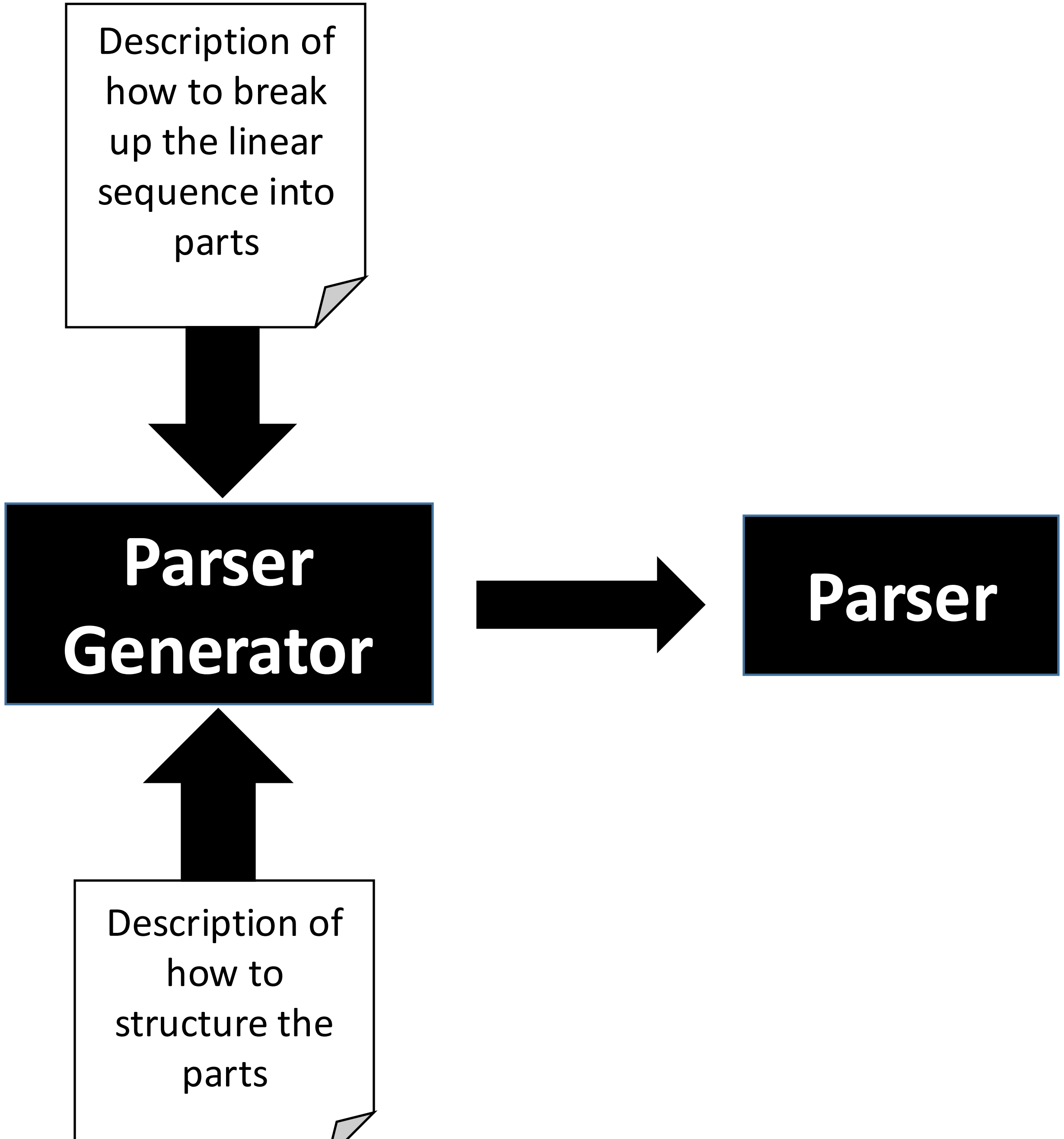

Application
Parser
Input (linear sequence of parts)

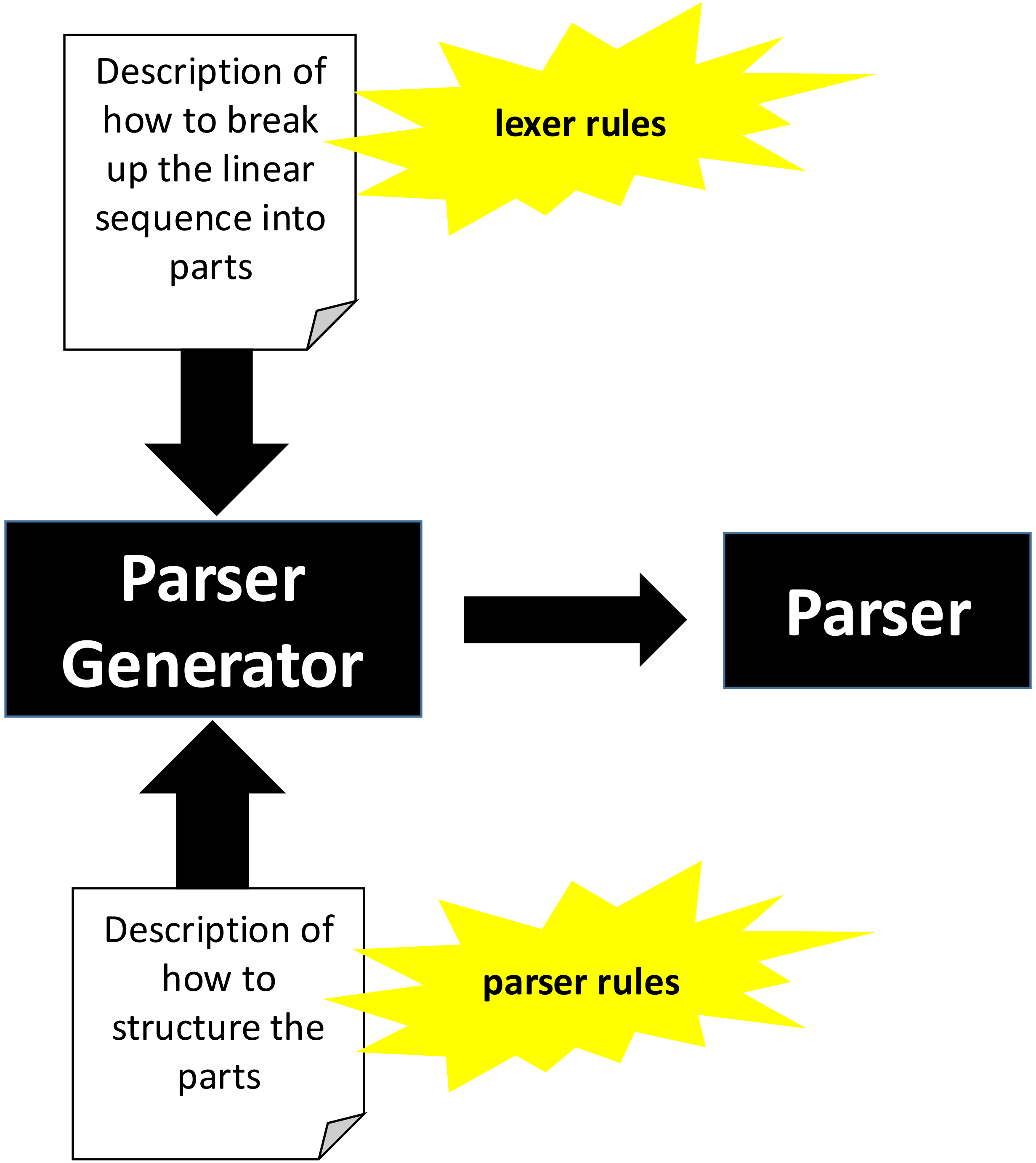

XML Schema
Validator
XML Parser
XML (linear sequence of parts)


JSON Schema Validator
JSON Parser
JSON (linear sequence of parts)


Parser Generator
Parser

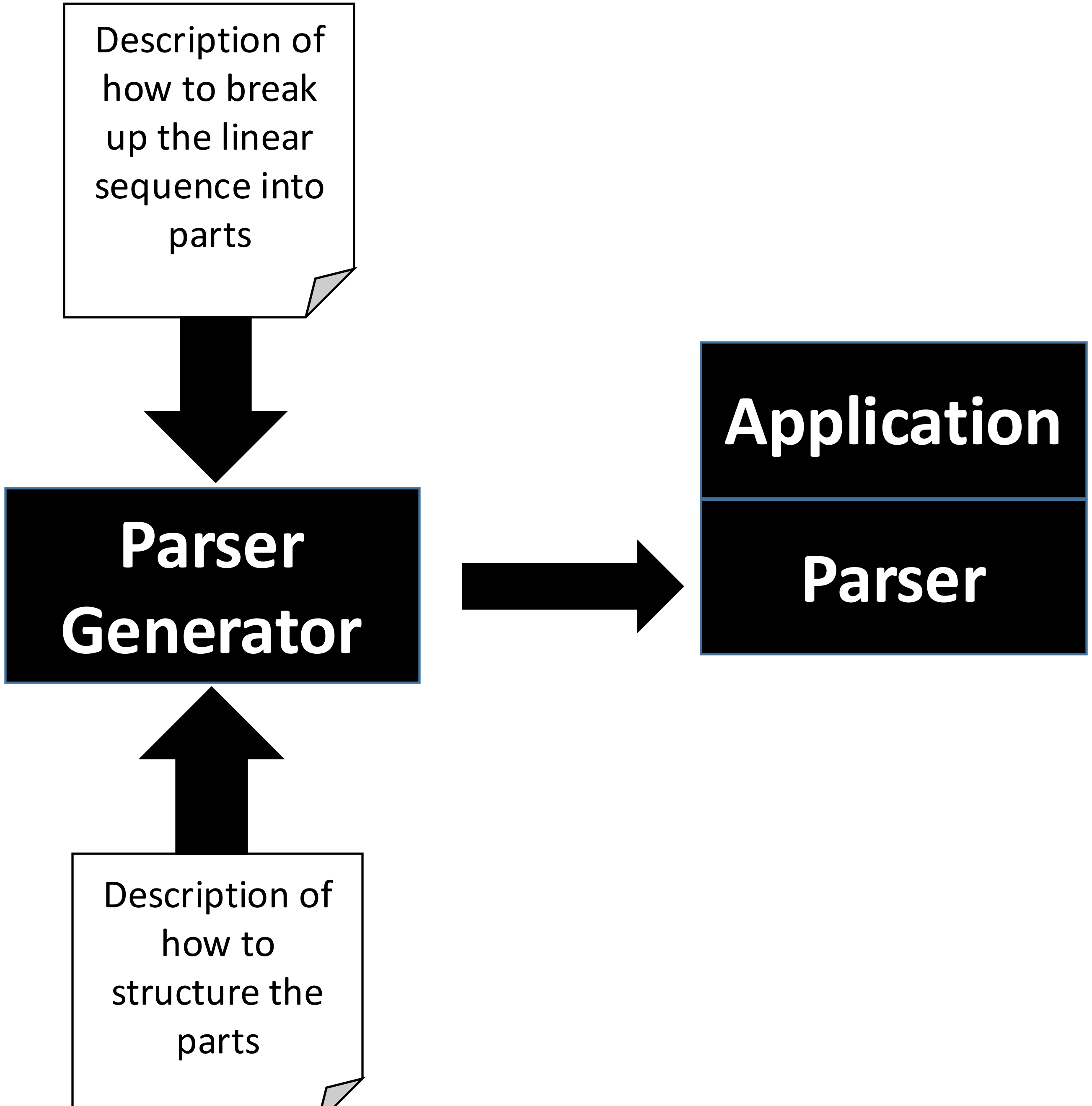

Description of how to break up the linear sequence into parts
Parser Generator
Parser

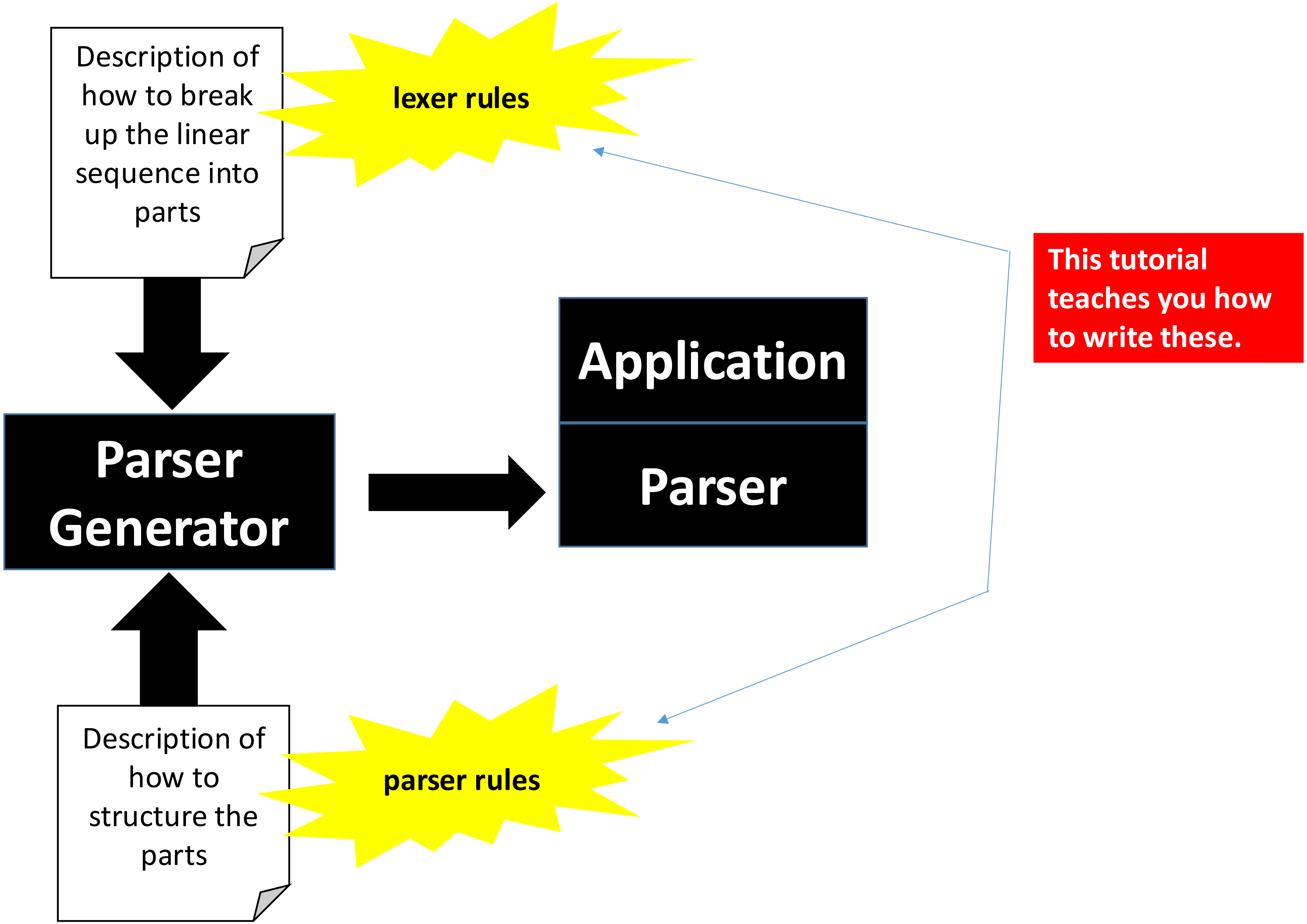

Description of how to break up the linear sequence into parts
Parser Generator
Parser
Description of how to structure the parts


Description of how to break up the linear sequence into parts
lexer rules
Parser Generator
Parser
Description of how to structure the parts
parser rules

Parser Generator (e.g. ANTLR)

Other popular parser generators: Bison, Yacc


Description of how to break up the linear sequence into parts
Parser Generator
Description of how to structure the parts
Application
Parser


Description of how to break up the linear sequence into parts
lexer rules
Parser Generator
Application
Parser
This tutorial teaches you how to write these.
Description of how to structure the parts
parser rules

Who uses ANTLR?

- **Twitter**: Uses ANTLR for parsing queries, over 2 billion queries a day
- **NetBeans IDE**: Uses ANTLR to parse C++
- **Oracle**: Uses ANTLR within their SQL Developer IDE
- **Hive and Pig Languages**: the data warehouse and analysis systems for Hadoop
- **Apple**: Uses ANTLR for their expression evaluator in Numbers (spreadsheet).

Acknowledgment: slide from Tim McNevin

ANTLR is all about generating parsers that can rip apart data files so that you can analyze their parts.

Table of Contents

1. Introduction to ANTLR, grammars, and parsing.
2. How to write grammars that are not tied to any particular programming language, i.e., how to write grammars which generate parsers that can be processed by different programming languages.
3. How to write grammars for binary data files.
4. How to insert programming language code into grammars. The resulting grammars generate parsers that can only be processed by that particular language.

Question: Is there an environment/IDE that you recommend for use in the tutorial?

Answer: This tutorial is focused on learning how to writing lexer and parser grammar rules. For that, any text editor is fine. We won't be writing much code (Java or Python), although we will be auto-generating code using ANTLR.

Introduction to ANTLR, grammars, and parsing

ANTLR Mailing List

https://groups.google.com/forum/#!forum/antlr-discussion

ANTLR book examples (Java version)

https://pragprog.com/titles/tpantlr2/source_code

ANTLR book examples (Python version)

https://github.com/jszheng/py3antlr4book

ANTLR Jar file, if you want to generate Java parsers

http://www.antlr.org/download/

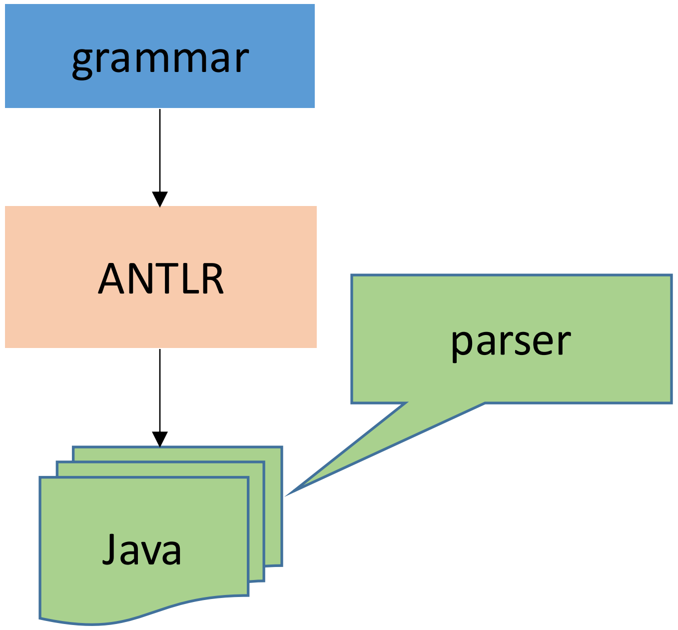
ANTLR Python runtime, if you want to generate Python parsers

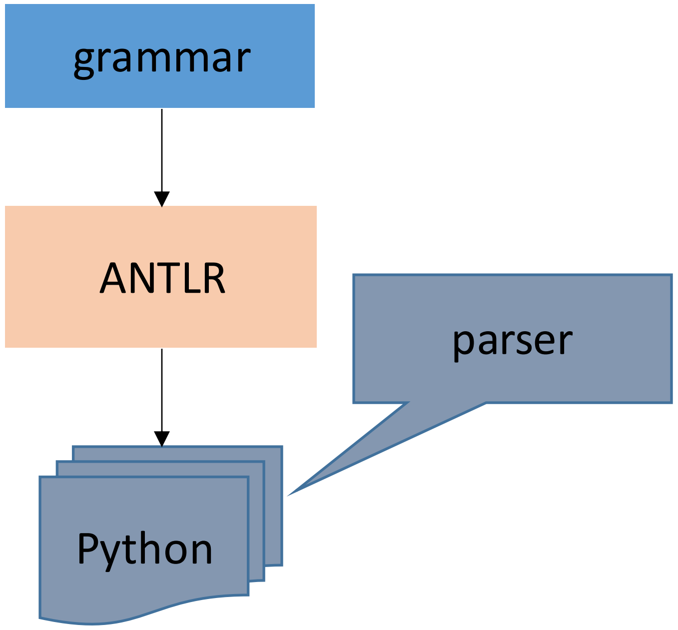
https://theantlrguy.atlassian.net/wiki/display/ANTLR4/Python+Target

Note: Python runtimes are better fetched using PyPI

What is ANTLR?

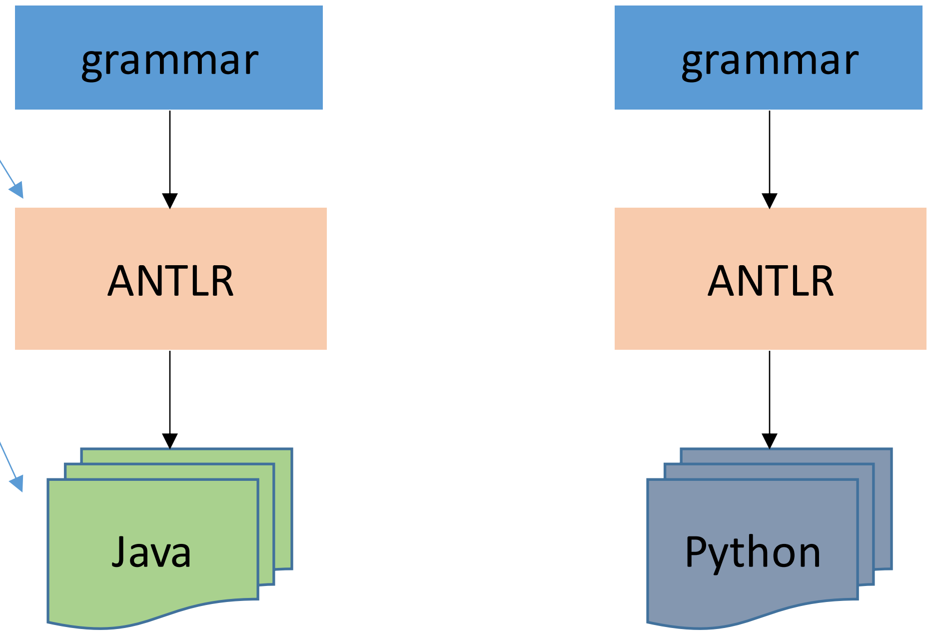
ANTLR is a tool (program) that converts a grammar into a parser:



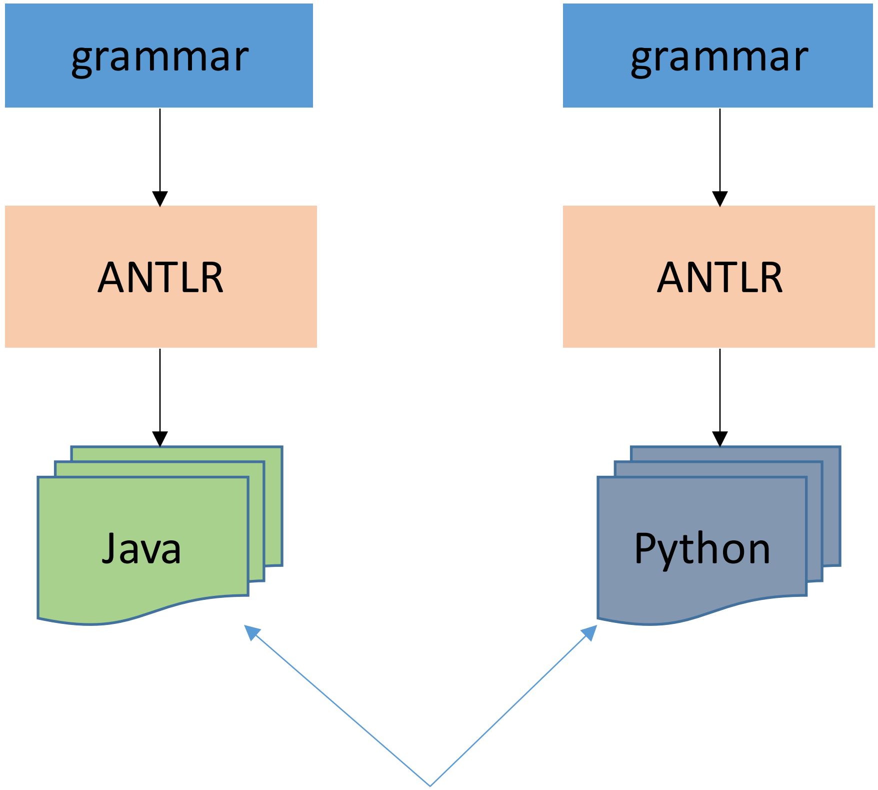


What is ANTLR? (cont.)

ANTLR is a program that generates another program (a parser). ANTLR is a parser generator!

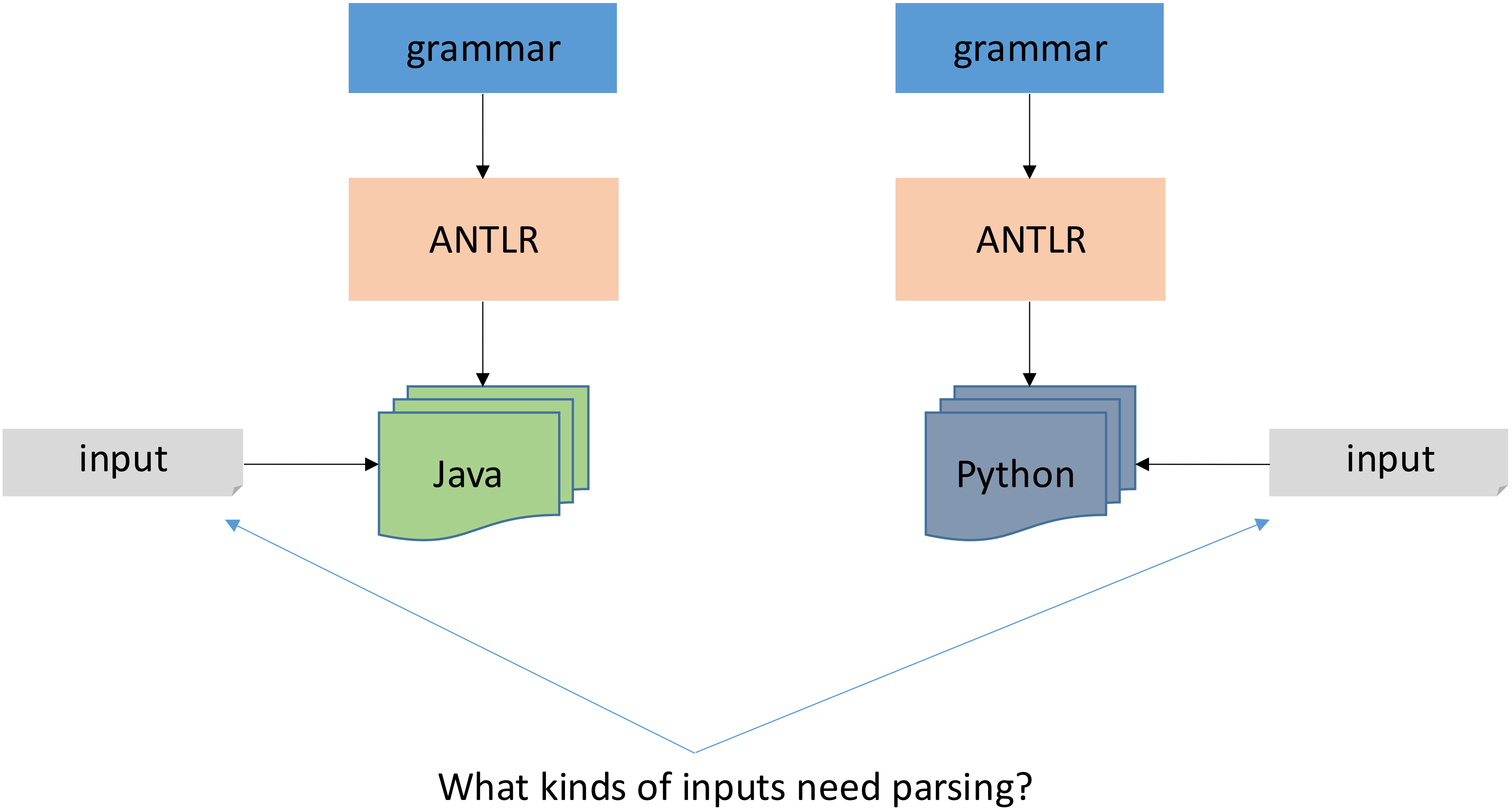


ANTLR Supports Multiple Target Languages



ANTLR can also generate C# and JavaScript and (future) C++ parsers

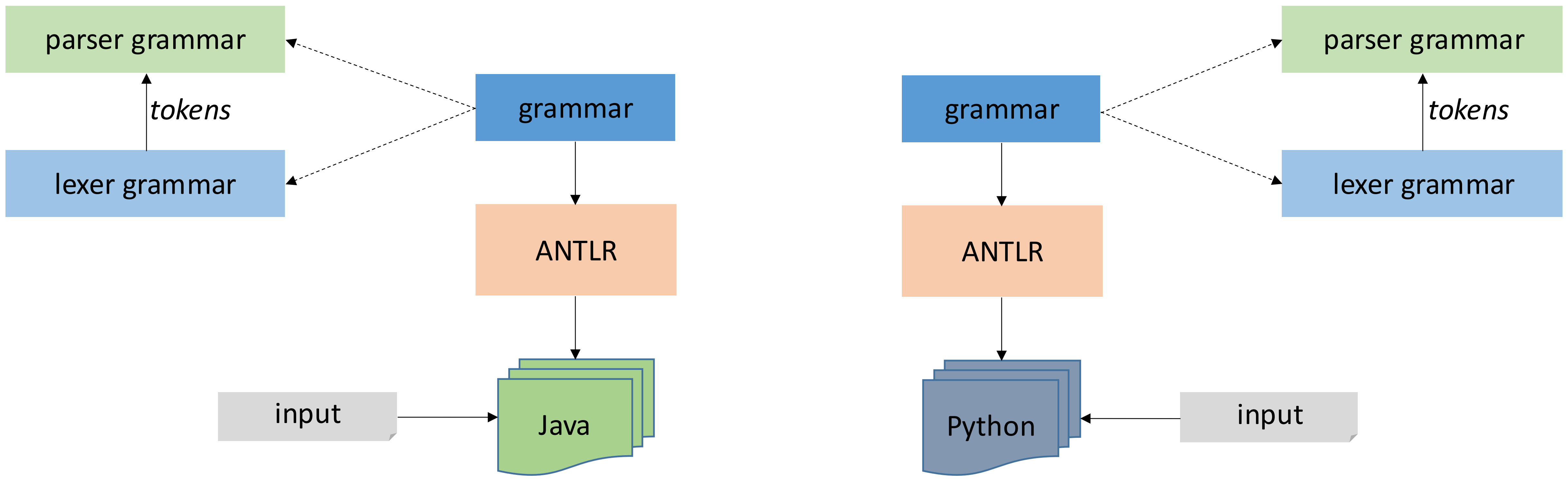
What kinds of inputs need parsing?



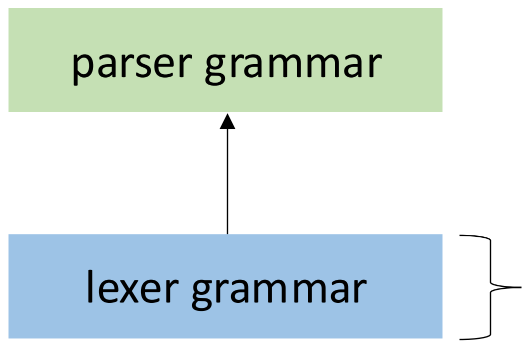
Kinds of inputs that need parsing

- **Data formats**: there are thousands of different data formats. There is a good chance that you will need to write an application to process data that is in a data format. You need ANTLR!
  - Examples of popular data formats: XML, JSON, Comma-Separated-Values (CSV), Key-Value pairs
- **Programming language**: there are hundreds of different programming languages. It is possible that you will need to write an application to process a program written in a programming language. You need ANTLR!
  - Example of popular programming languages: Java, Python, C++, C#, C

Two grammars

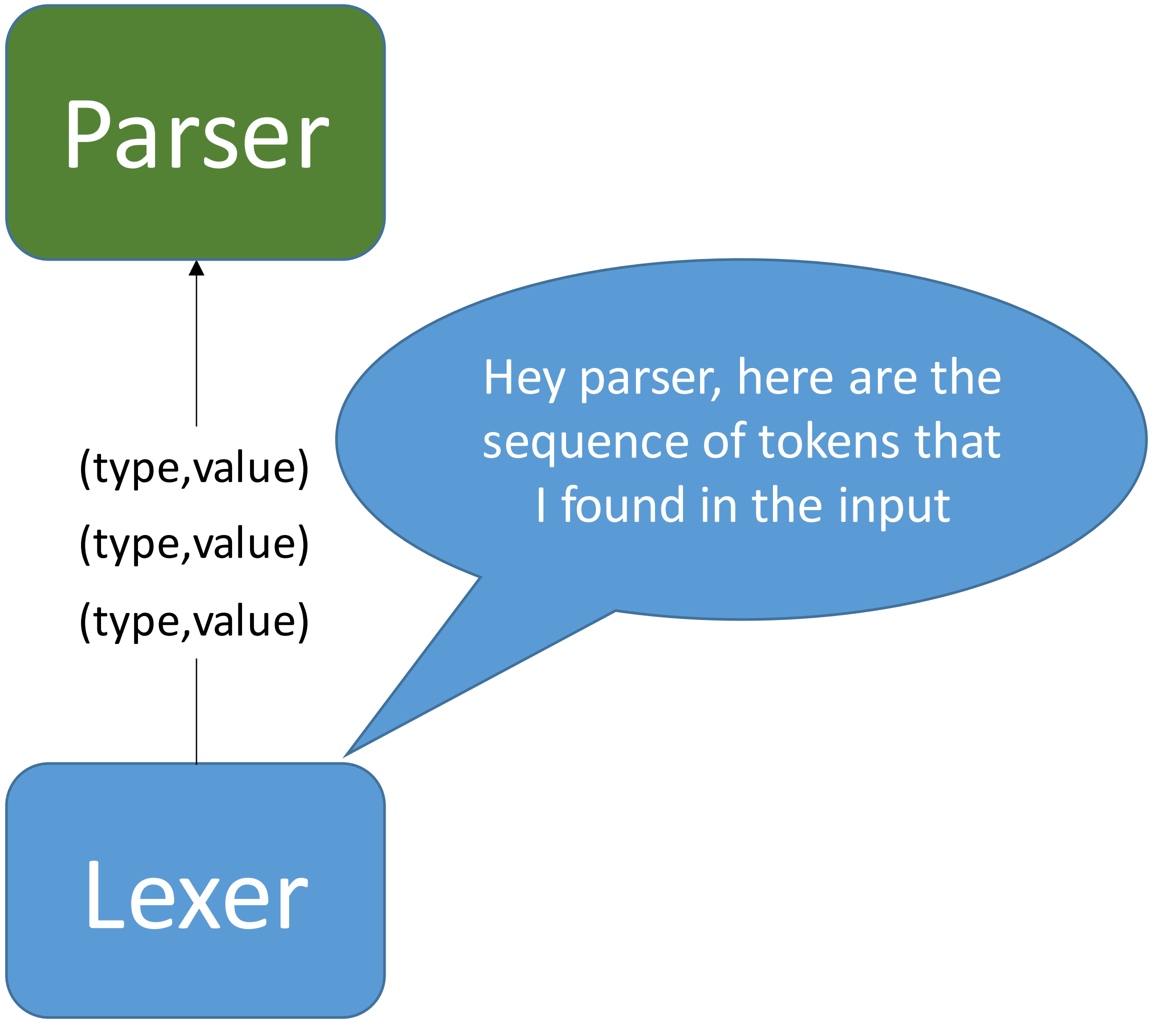


Lexer grammar

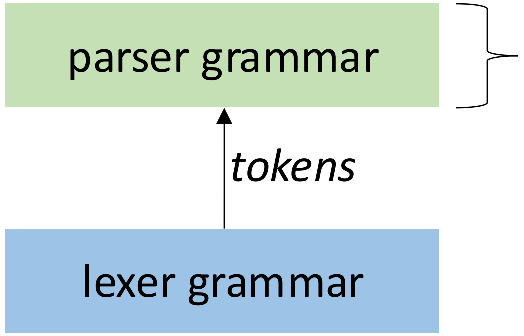


A lexer grammar describes how the input should be broken up into parts (tokens). This description is accomplished using lexer rules. The form of a token is expressed using regular expressions.

Lexer

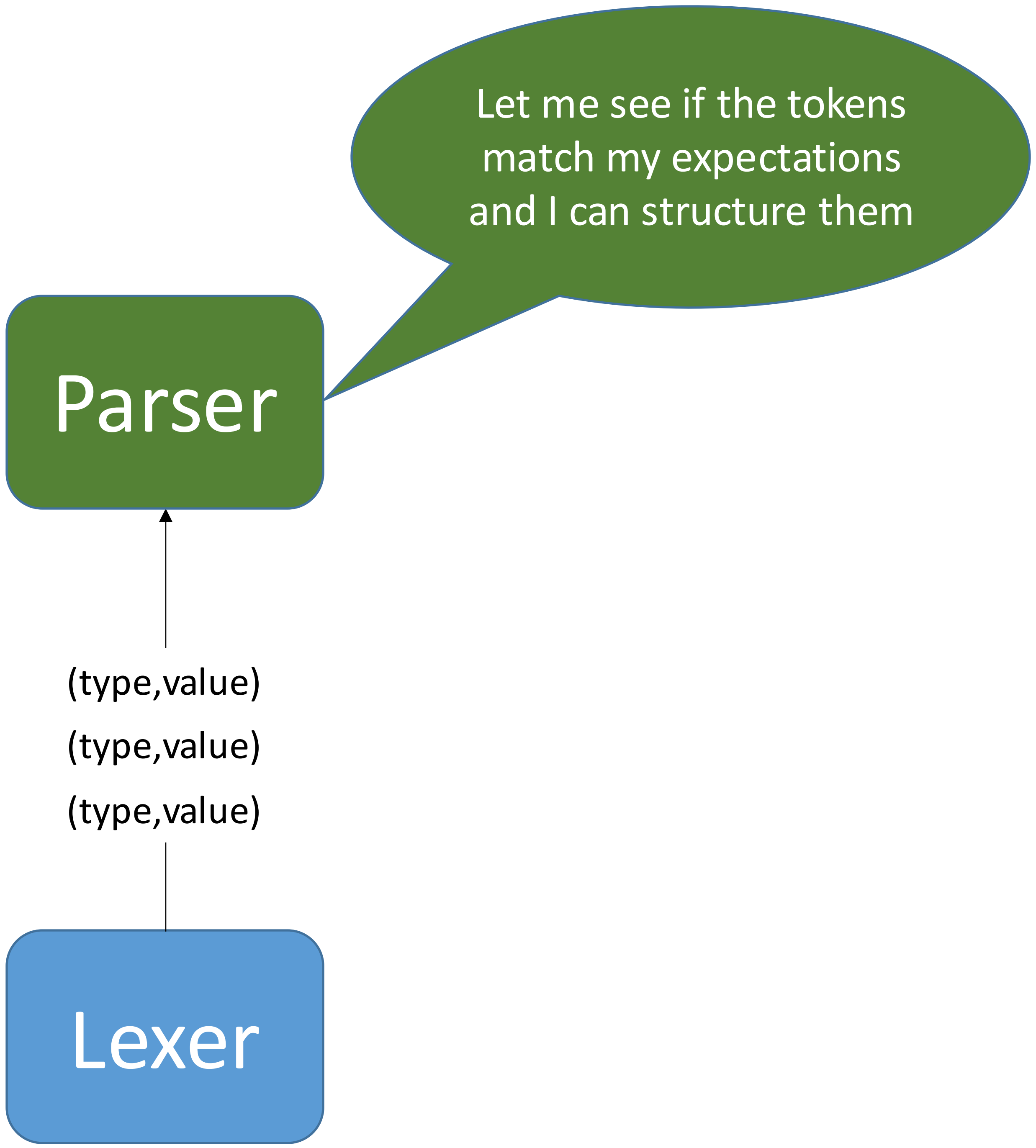

Parser
(type,value)
(type,value)
(type,value)
Hey parser, here are the sequence of tokens that I found in the input
Lexer

Parser grammar



A parser grammar describes how to structure the tokens received from the lexer. This description is accomplished using parser rules.

Parser


Let me see if the tokens match my expectations and I can structure them
Parser
(type,value)
(type,value)
(type,value)
Lexer

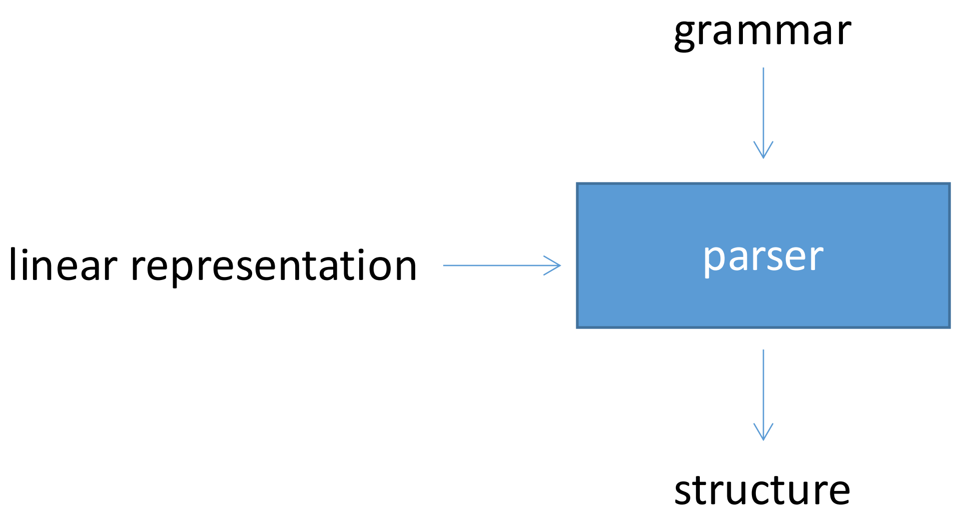
Parsing

- Definition: Parsing is the process of structuring a linear representation in accordance with a given grammar.
- This definition has been kept abstract on purpose to allow as wide an interpretation as possible.
- The “linear representation” may be:
    - a sentence
    - a computer program
    - a knitting pattern
    - a sequence of geological strata
    - a piece of music
    - actions of ritual behavior

  In short, any linear sequence in which the preceding elements in some way restrict the next element.
- For some of the examples the grammar is well known, for some it is an object of research, and for some our notion of a grammar is only just beginning to take shape.

Acknowledgment: the next few slides contain info from the (excellent) book [Parsing Techniques] by Dick Grune

Parsing



Parsing is the process of structuring a linear representation in accordance with a given grammar. A “linear representation” is any linear sequence in which the preceding elements in some way restrict the next element.

Grammar

- For each grammar, there are generally an infinite number of linear representations (“sentences”) that can be structured with it.
- That is, a finite-sized grammar can supply structure to an infinite number of sentences.
- This is the main strength of the grammar paradigm and indeed the main source of the importance of grammars: they summarize succinctly the structure of an infinite number of objects of a certain class.

Reasons for parsing

There are several reasons to perform this structuring process called parsing.

1. One reason derives from the fact that the obtained structure helps us to process the object further. When we know that a certain segment of a sentence is the subject, that information helps in understanding or translating the sentence. Once the structure of a document has been brought to the surface, it can be processed more easily.
2. A second reason is related to the fact that the grammar in a sense represents our understanding of the observed sentences: *the better a grammar we can give for the movement of bees, the deeper our understanding of them*.
3. A third lies in the completion of missing information that parsers, and especially error-repairing parsers, can provide. Given a reasonable grammar of the language, an error-repairing parser can suggest possible word classes for missing or unknown words on clay tablets.

The science of parsing

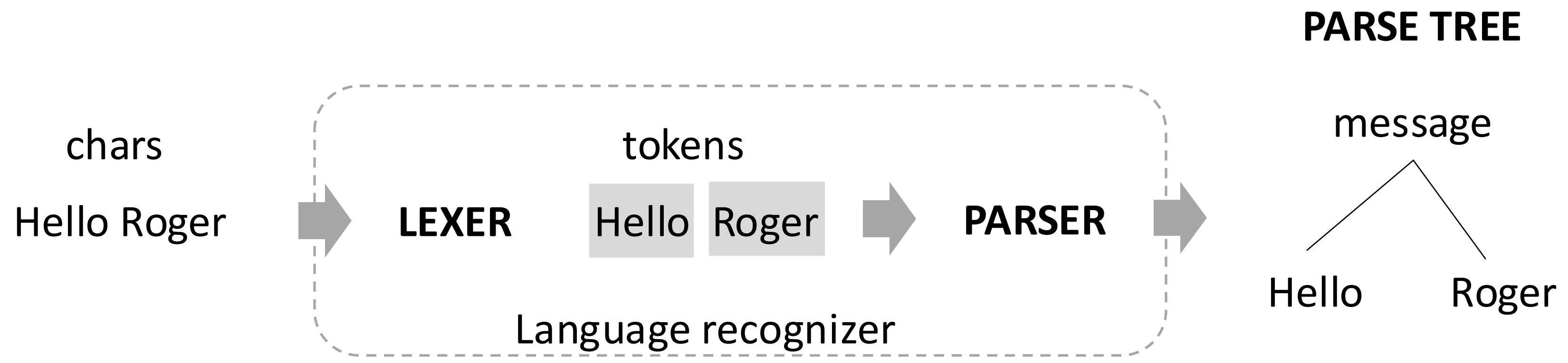
- Parsing is no longer an arcane art.
- In the 1970s Aho, Ullman, Knuth, and many others put parsing techniques solidly on their theoretical feet.

Many uses for parsing

Parsing is for anyone who has parsing to do:

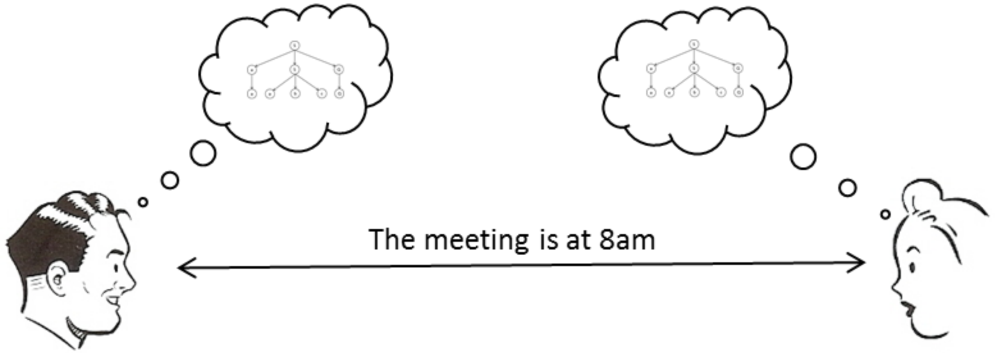
- The compiler writer
- The linguist
- The database interface writer
- The geologist who wants to test grammatical descriptions of a sequence of geological strata
- The musicologist who wants to test grammatical descriptions of a music piece

Data flow of a language recognizer



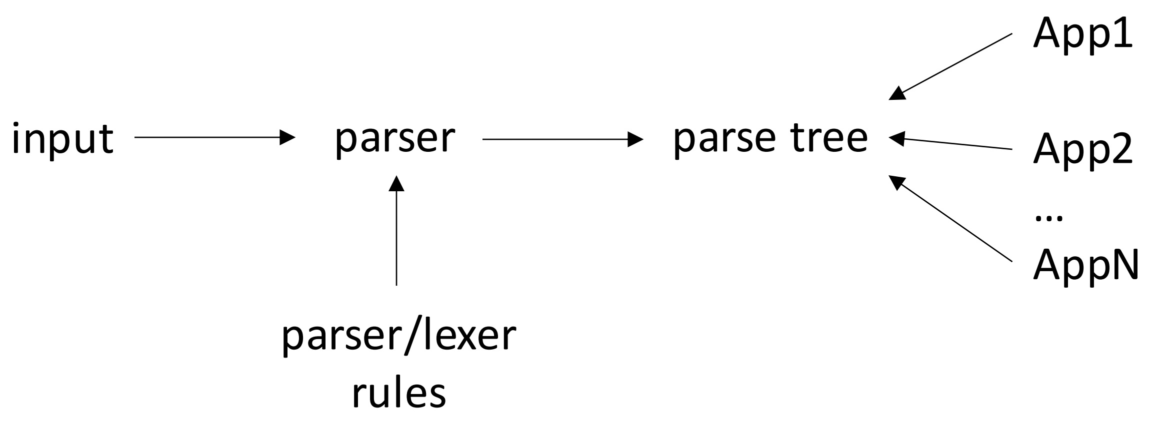
Humans, sentences, parse trees

Sentences (linear sequences of symbols) are really just serializations of parse trees that we humans understand. To get an idea across to someone we have to conjure up the same parse tree in their heads using a word stream.



According to Chomsky, sentences in a language, natural or artificial, are constructed according to a grammar. While being generated they obtain a structure, the generation tree. This structure encodes the meaning. When the sentence is spoken or written the terminal symbols (words) alone are transferred to the listener or reader, losing the structure (linearized). Since the meaning is attached to that structure the listener or reader will have to reconstruct the generation tree, now called the parse tree, to retrieve the meaning. That's why we need parsing.

Multiple applications may operate on a parse tree



Creating language-independent grammars

Let's create our first parser

Let's create a lexer and parser grammar for input that consists of a greeting ('Hello' or 'Greetings') followed by the name of a person. Here are two valid input strings:

```
Hello Roger
Greetings Sally
```

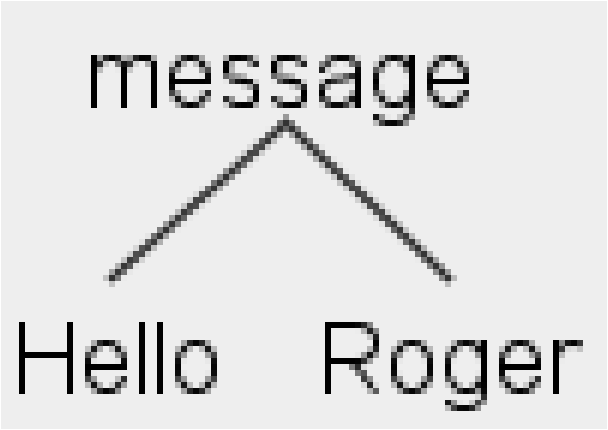
Our first parser (cont.)

The lexer grammar describes how to break the input into two tokens:

(1) The greeting (Hello or Greetings), and

(2) The name of the person.

The parser grammar describes how to structure the tokens as shown by the below graphic (the input is: Hello Roger):



See:

java-examples/example01

python-examples/example01

The lexer grammar

```
lexer grammar MyLexer ;

GREETING    : ('Hello' | 'Greetings') ;
ID          : [a-zA-Z]+ ;
WS          : [ \t\r\n]+ -> skip ;
```

MyLexer.g4

```
lexer grammar MyLexer ;

GREETING     : ('Hello' | 'Greetings') ;
ID           : [a-zA-Z]+ ;
WS           : [ \t\r\n]+ -> skip ;
```

MyLexer.g4

filename, must have this suffix: g4

This is a lexer grammar (not a parser grammar)

```
lexer grammar MyLexer ;

GREETING      : ('Hello' | 'Greetings') ;
ID            : [a-zA-Z]+ ;
WS            : [ \t\r\n]+ -> skip ;
```

MyLexer.g4

Name of the lexer

```
lexer grammar MyLexer ;

GREETING     : ('Hello' | 'Greetings') ;
ID           : [a-zA-Z]+ ;
WS           : [ \t\r\n]+ -> skip ;
```

MyLexer.g4

These two must be the same (i.e., the filename and the lexer name must be the same)

```
lexer grammar MyLexer ;

GREETING     : ('Hello' | 'Greetings') ;
ID           : [a-zA-Z]+ ;
WS           : [ \t\r\n]+ -> skip ;
```

MyLexer.g4

```
lexer grammar MyLexer ;

GREETING      : ('Hello' | 'Greetings') ;
ID            : [a-zA-Z]+ ;
WS            : [ \t\r\n]+ -> skip ;
```

MyLexer.g4

Any string in the input that matches 'Hello' or 'Greetings' is to be treated as a GREETING.

GREETING is a token type.

```
lexer grammar MyLexer ;

GREETING      : ('Hello' | 'Greetings') ;
ID            : [a-zA-Z]+ ;
WS            : [ \t\r\n]+ -> skip ;
```

MyLexer.g4

'Hello' and 'Greetings' are token values (string literals).

```
lexer grammar MyLexer ;

GREETING     : ('Hello' | 'Greetings') ;
ID           : [a-zA-Z]+ ;
WS           : [ \t\r\n]+ -> skip ;
```

MyLexer.g4

Delimiter of string literals

string literals are always delimited by <u>single</u> quotes (not double quotes)

```
lexer grammar MyLexer ;

GREETING      : ('Hello' | 'Greetings') ;
ID            : [a-zA-Z]+ ;
WS            : [ \t\r\n]+ -> skip ;
```

MyLexer.g4

```
lexer grammar MyLexer ;

GREETING      : ('Hello' | 'Greetings') ;
ID            : [a-zA-Z]+ ;
WS            : [ \t\r\n]+ -> skip ;
```

MyLexer.g4

This is a token rule (lexer rule).

```
lexer grammar MyLexer ;

GREETING      : ('Hello' | 'Greetings') ;
ID            : [a-zA-Z]+ ;
WS            : [ \t\r\n]+ -> skip ;
```

MyLexer.g4

Any string in the input that matches this regular expression is to be treated as an ID. The plus symbol (+) means "one or more".

```
lexer grammar MyLexer ;

GREETING      : ('Hello' | 'Greetings') ;
ID            : [a-zA-Z]+ ;
WS            : [ \t\r\n]+ -> skip ;
```

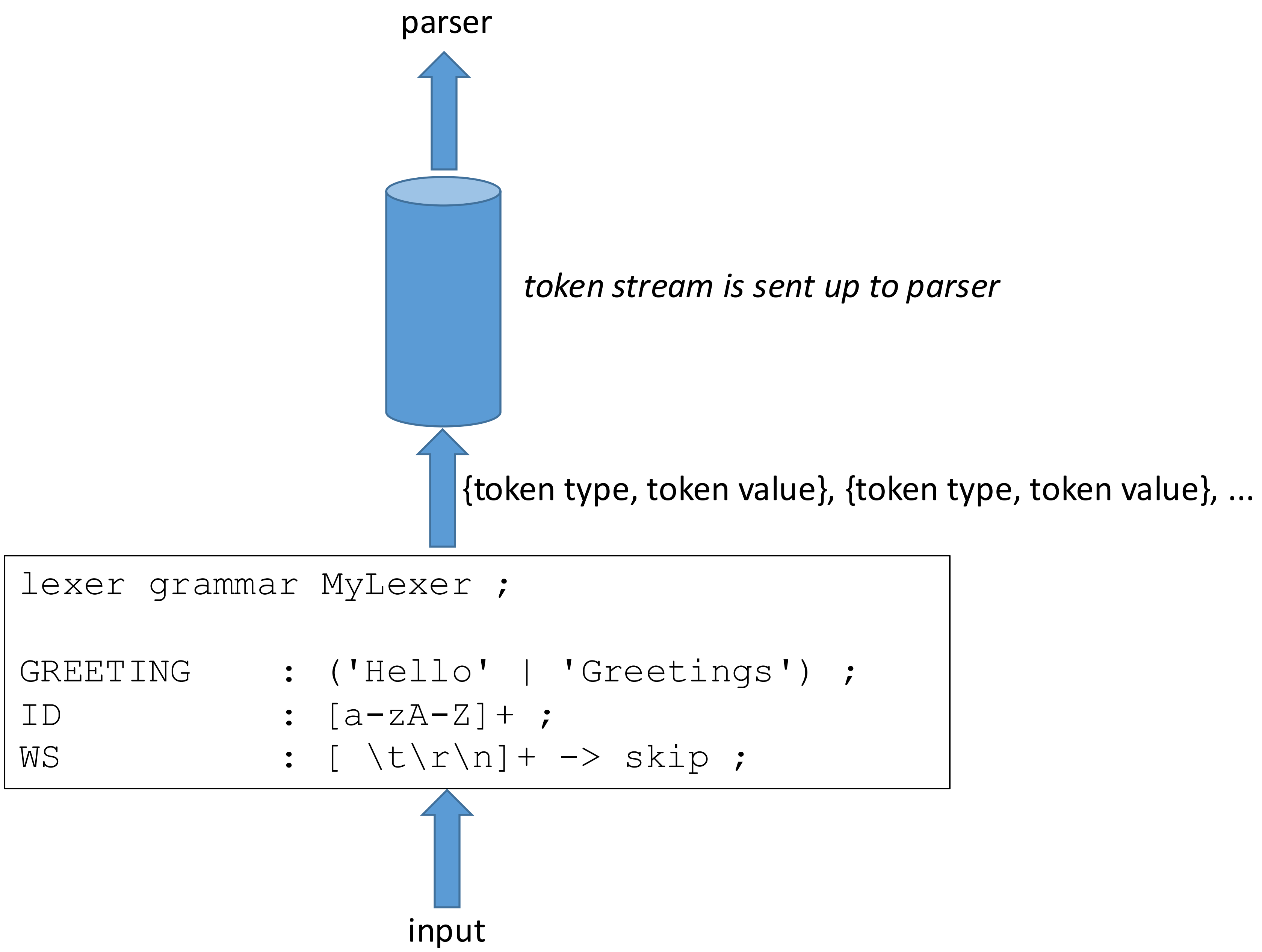
MyLexer.g4

Any whitespace (space, tab, carriage return, newline) in the input is to be treated as a WS token. The parser doesn't need these tokens, so we want the lexer to discard them (not send them up to the parser).

Lexer command

```
lexer grammar MyLexer ;

GREETING    : ('Hello' | 'Greetings') ;
ID          : [a-zA-Z]+ ;
WS          : [ \t\r\n]+ -> skip ;
```

Lexer command appears at the end of a lexer rule. The "skip" lexer command tells the lexer to collect the token and then discard it (i.e., don't send the token up to the parser).


parser
token stream is sent up to parser
{token type, token value}, {token type, token value}, ...
lexer grammar MyLexer ;
GREETING : ('Hello' | 'Greetings') ;
ID : [a-zA-Z]+ ;
WS : [\t\r\n]+ -> skip ;
input

Token names must begin with an Upper-Case letter

Must begin with a capital letter

```
lexer grammar MyLexer ;

GREETING     : ('Hello' | 'Greetings') ;
ID           : [a-zA-Z]+ ;
WS           : [ \t\r\n]+ -> skip ;
```

Token name: Capital letter (from any alphabet) followed by zero or more digit (0-9), underscore, upper- or lower-case letter (from any alphabet).

The parser grammar

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

message : GREETING ID;
```

MyParser.g4

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

message : GREETING ID;
```

MyParser.g4

filename, must have this suffix: g4

Lexer and parser filenames must have the same prefix

```
lexer grammar MyLexer ;

GREETING     : ('Hello' | 'Greetings') ;
ID           : [a-zA-Z]+ ;
WS           : [ \t\r\n]+ -> skip ;
```

MyLexer.g4

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

message : GREETING ID;
```

MyParser.g4

The filename prefixes for the lexer and parser must be the same.

Filename recommendation

Always name your lexer: **MyLexer.g4**

Always name your parser: **MyParser.g4**

This is a parser grammar (not a lexer grammar)

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

message : GREETING ID;
```

MyParser.g4

Name of the parser

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

message : GREETING ID;
```

MyParser.g4

These two must be the same (i.e., the filename and the parser name must be the same)

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

message : GREETING ID;
```

MyParser.g4

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

message : GREETING ID;
```

MyParser.g4

This parser will use the tokens generated by MyLexer

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

message : GREETING ID;
```

MyParser.g4

If the input is valid, it must contain a token of type GREETING followed by a token of type ID.

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

message : GREETING ID;
```

MyParser.g4

This is a parser rule.

Parser names must begin with a lower-case letter

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

message : GREETING ID;
```

Must begin with a lower-case letter

Parser name: lower-case letter (from any alphabet) followed by zero or more digit (0-9), underscore, upper- or lower-case letter (from any alphabet).

Parser versus Lexer

Rule for defining the structure of tokens received from the lexer.

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

message : GREETING ID;
```

Rules for defining the vocabulary symbols (tokens) of the input.

```
lexer grammar MyLexer ;

GREETING     : ('Hello' | 'Greetings') ;
ID           : [a-zA-Z]+ ;
WS           : [ \t\r\n]+ -> skip ;
```

A lexer rule for every character

Every possible input character must be matched by at least one lexical rule.

These lexer rules must be able to tokenize all of the input.

```
lexer grammar MyLexer ;

GREETING     : ('Hello' | 'Greetings') ;
ID           : [a-zA-Z]+ ;
WS           : [ \t\r\n]+ -> skip ;
```

ANTLR internals: parser rules are converted to Java/Python functions

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

message : GREETING ID;
```

ANTLR

```
message(...)
{
   ...
}
```

Consequently, your parser rule names must not conflict with Java/Python keywords. For example, do not have a parser rule like this:

char : CHAR ;

illegal since "char" is a keyword in Java

Java keywords

Your parser rule names and your lexer rule names must not be any of these:

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Python keywords

Your parser rule names and your lexer rule names must not be any of these:

and	del	from	not	while
as	elif	global	or	with
assert	else	if	pass	yield
break	except	import	print	
class	exec	in	raise	
continue	finally	is	return	
def	for	lambda	try	

ANTLR reserved words

Your parser rule names and your lexer rule names must not be any of these:

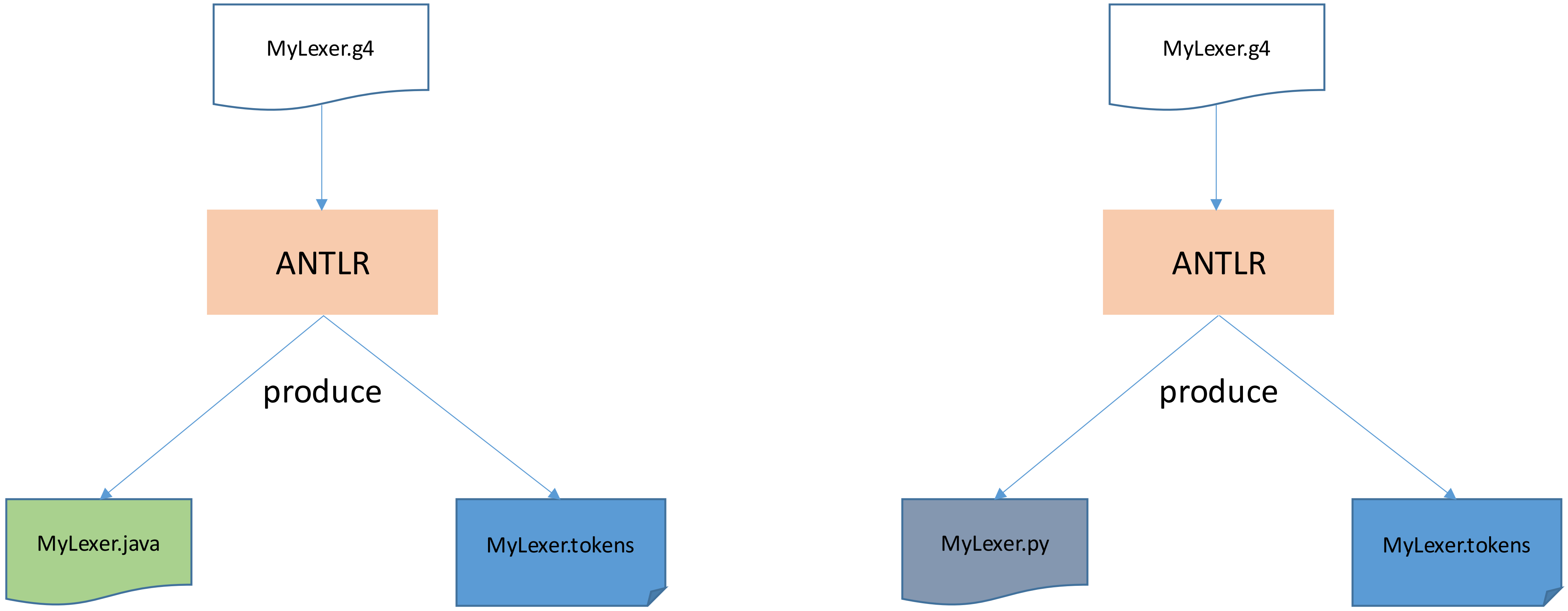
- import
- fragment
- lexer
- parser
- grammar
- returns
- locals
- throws
- catch
- finally
- mode
- options
- tokens
- rule

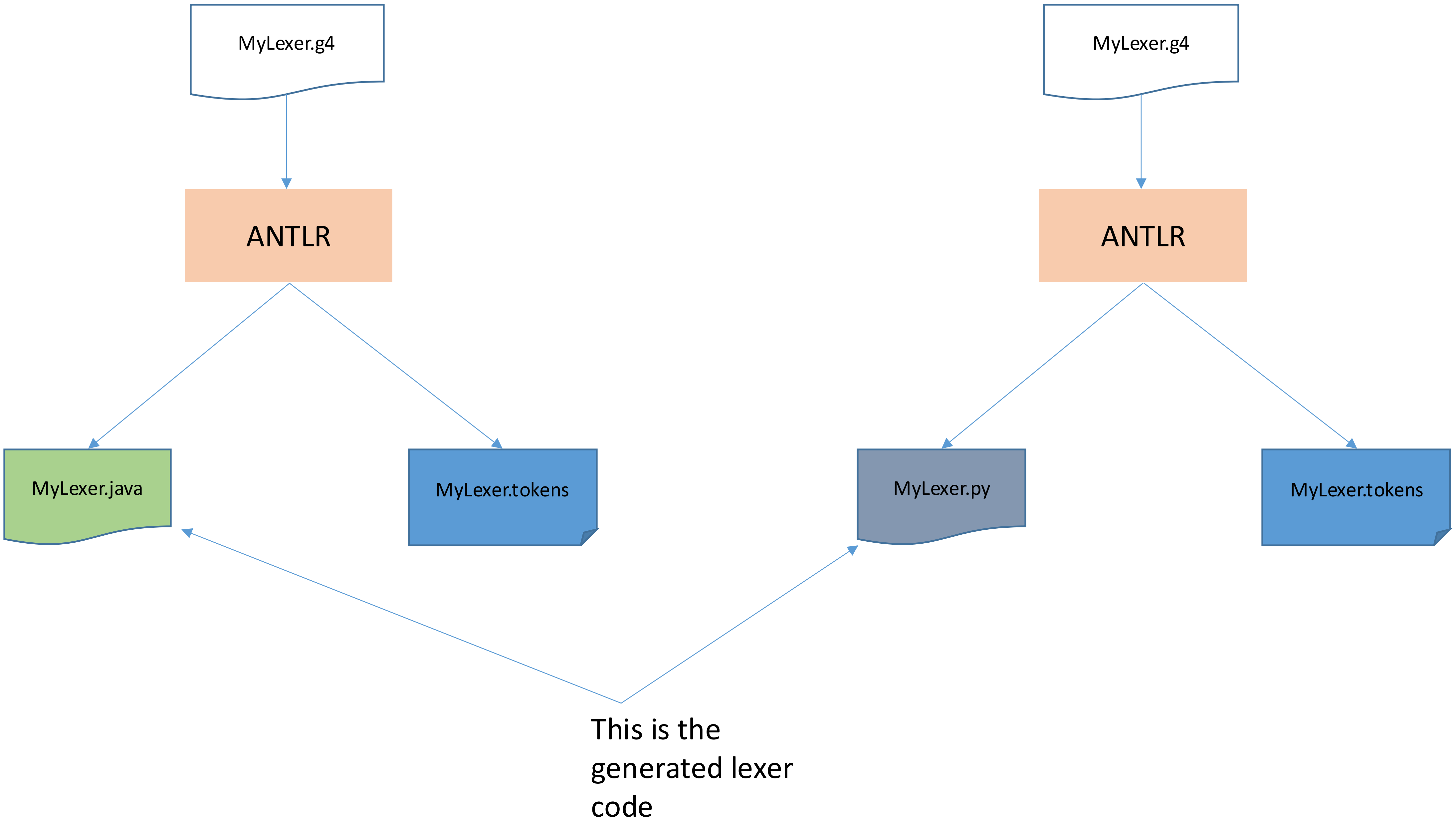
antlr4.bat

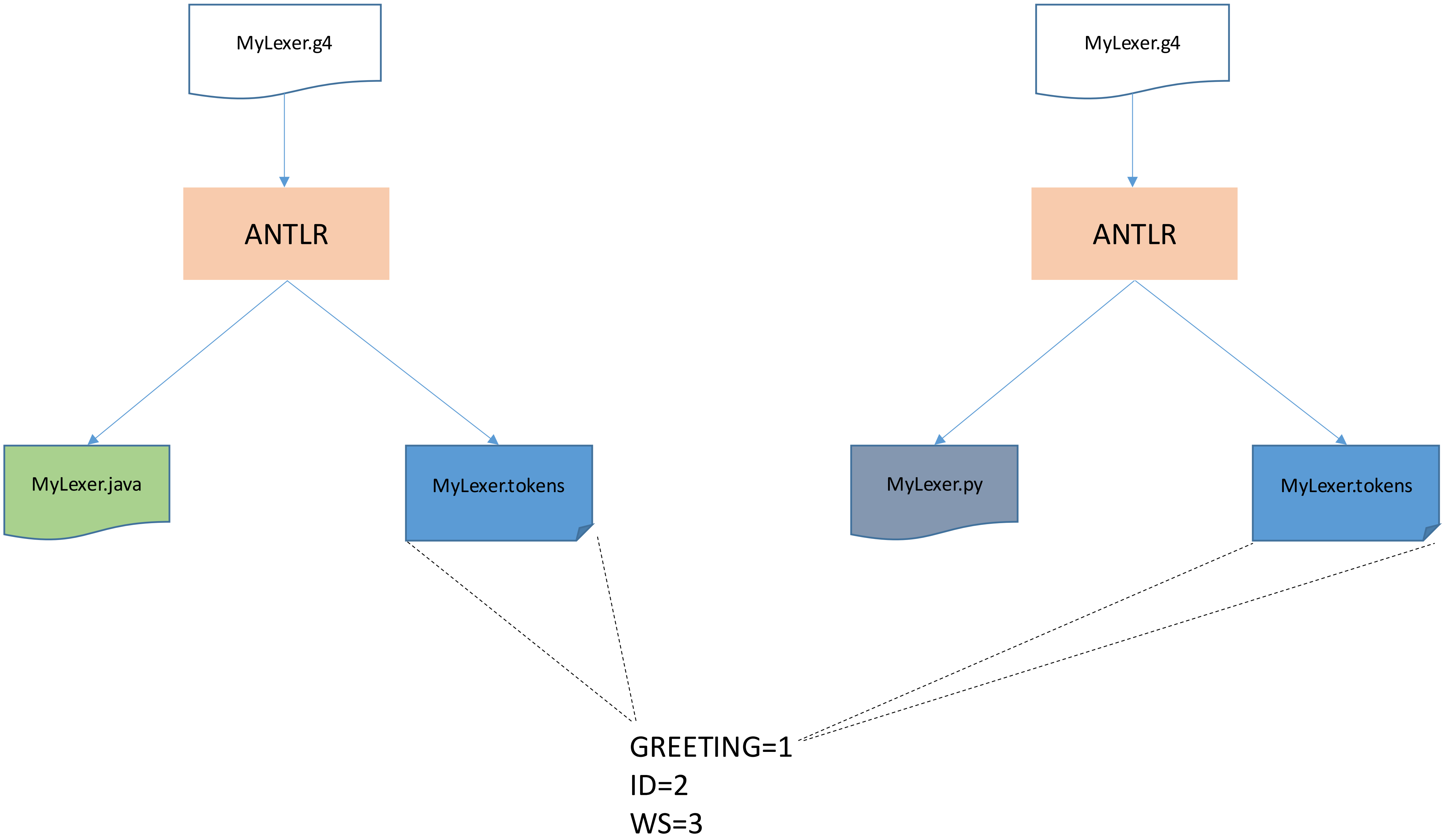
- We will run things from a command window.
- I created a batch file, antlr4.bat
- It invokes the ANTLR tool.
- When you invoke it, provide it the name of either the lexer or parser.
- You *must* run ANTLR on the lexer before the parser.

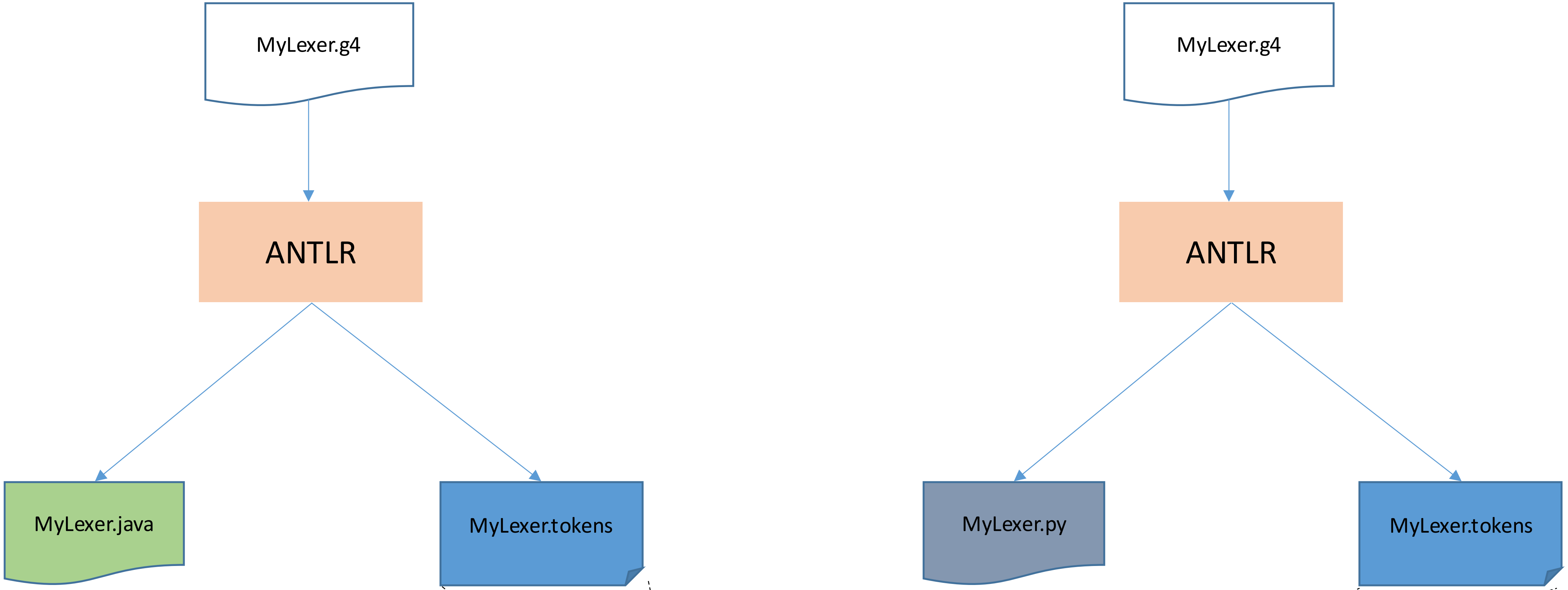
```
antlr4 MyLexer.g4
antlr4 MyParser.g4
```

First run ANTLR on the lexer grammar



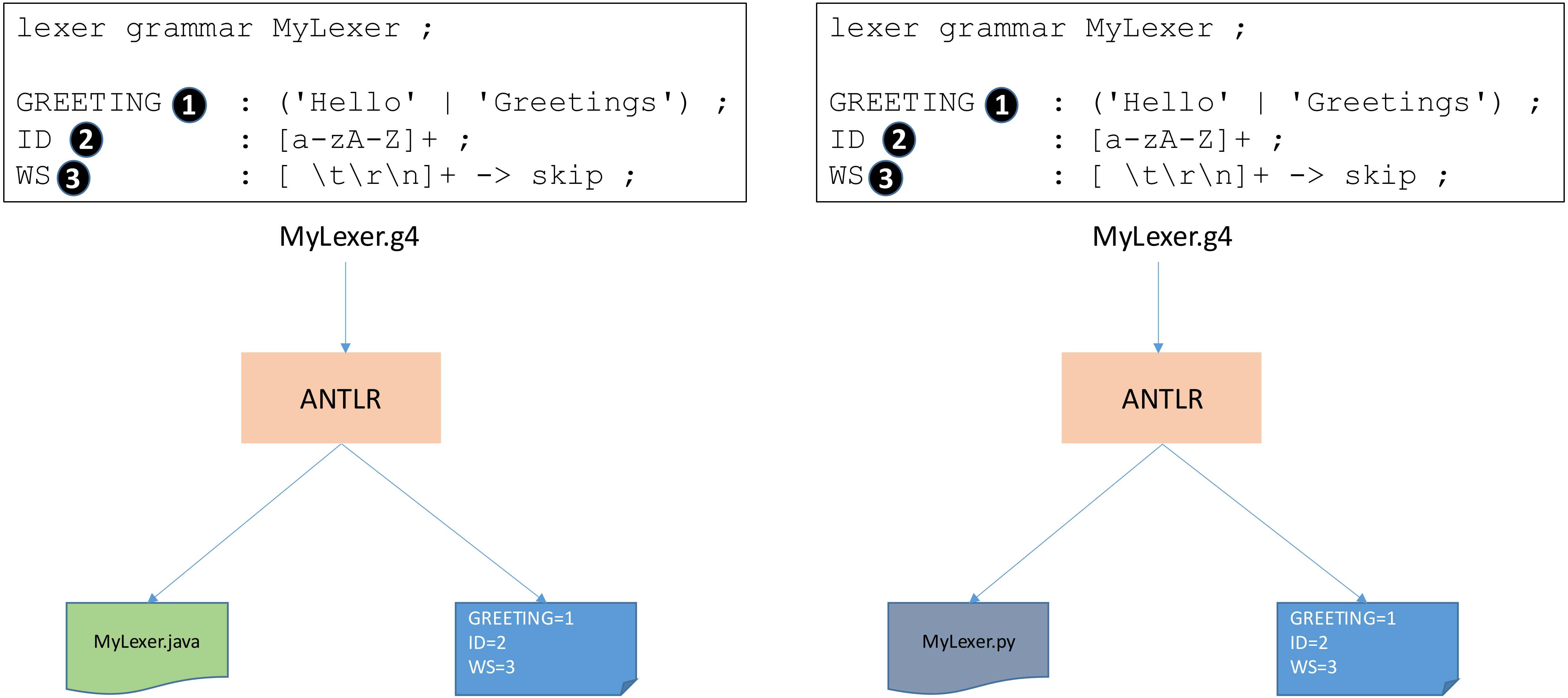

MyLexer.g4
ANTLR
MyLexer.java
MyLexer.tokens
MyLexer.g4
ANTLR
MyLexer.py
MyLexer.tokens
This is the generated lexer code


MyLexer.g4
ANTLR
MyLexer.java
MyLexer.tokens
MyLexer.g4
ANTLR
MyLexer.py
MyLexer.tokens
GREETING=1
ID=2
WS=3

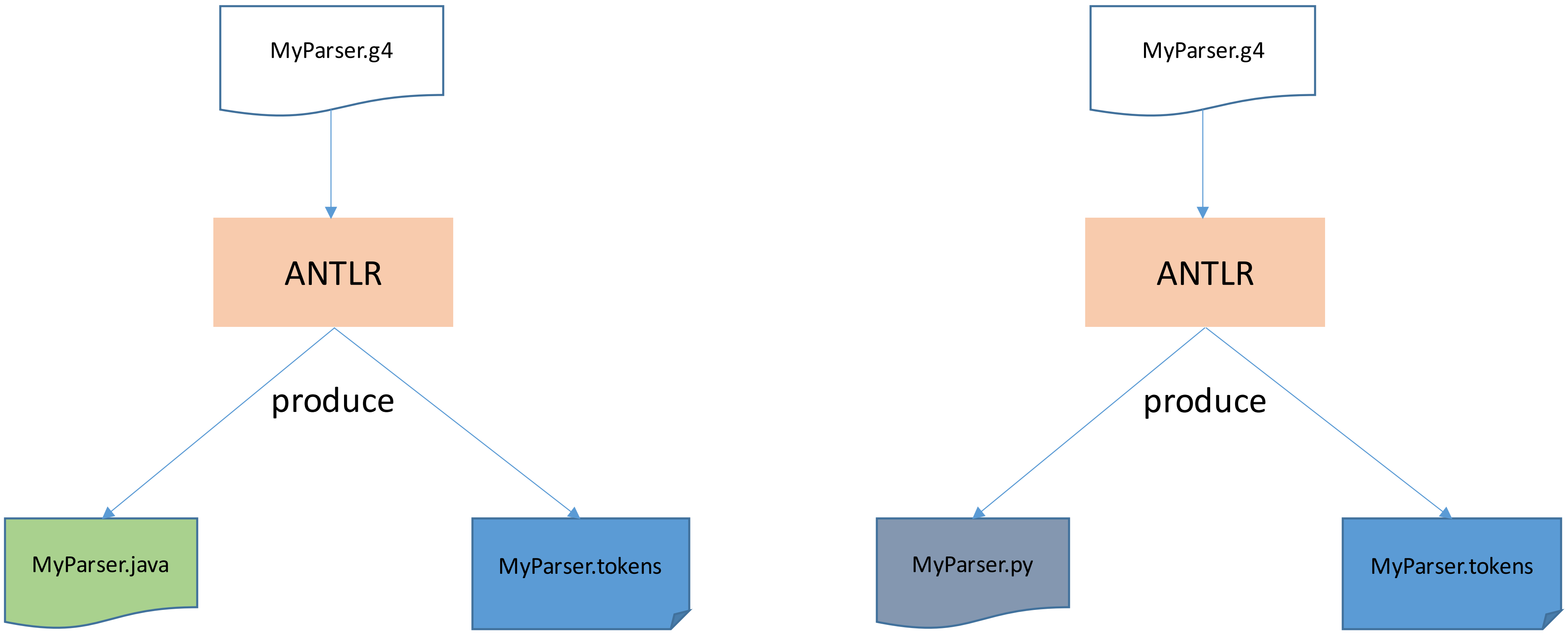


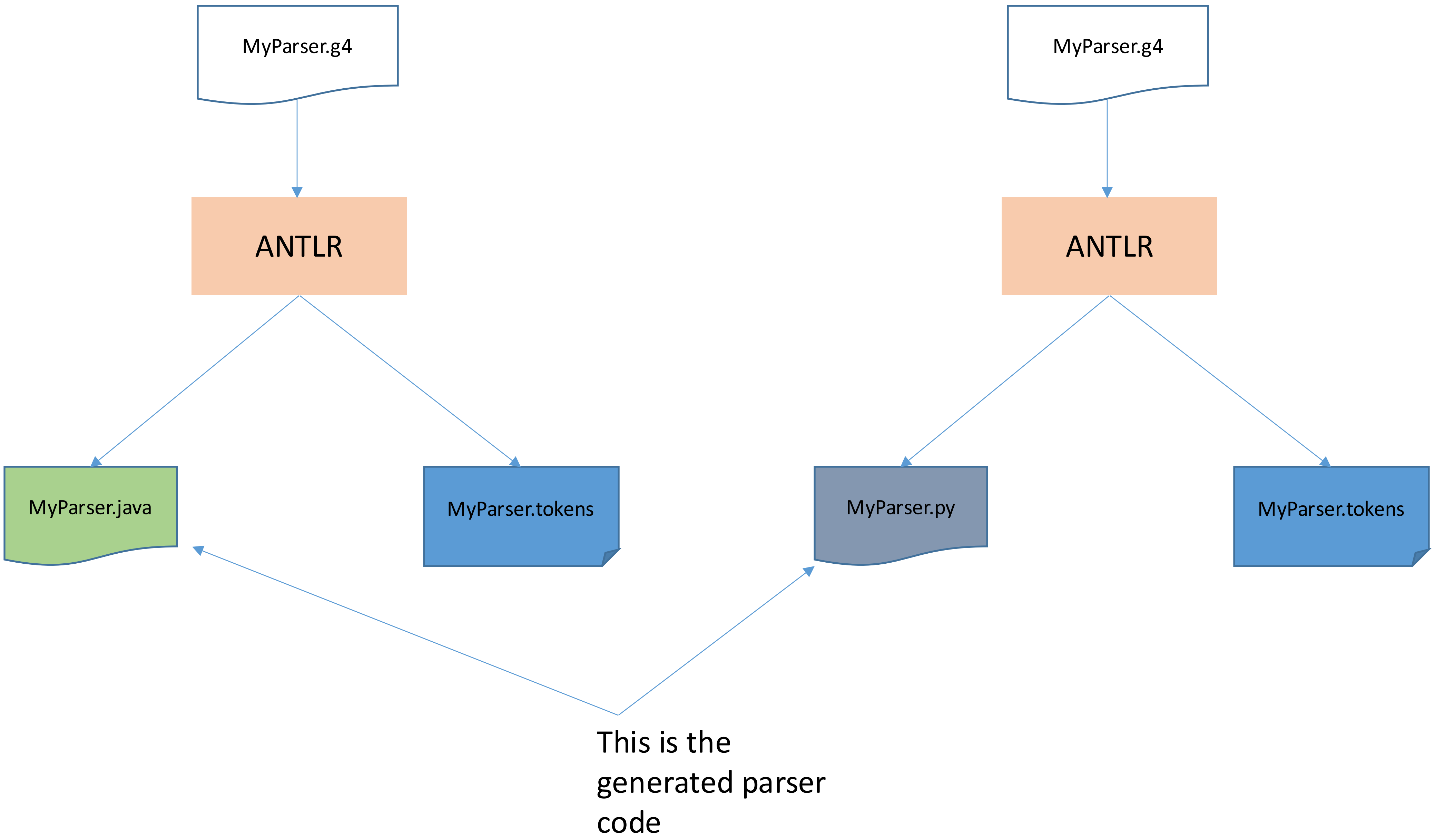
Each token generated by the lexer has a value and a type. ANTLR uses numbers for the type. The first token specified in MyLexer.g4 is assigned the number 1, the second token specified in MyLexer.g4 is assigned the number 2, etc.

GREETING=1
ID=2
WS=3

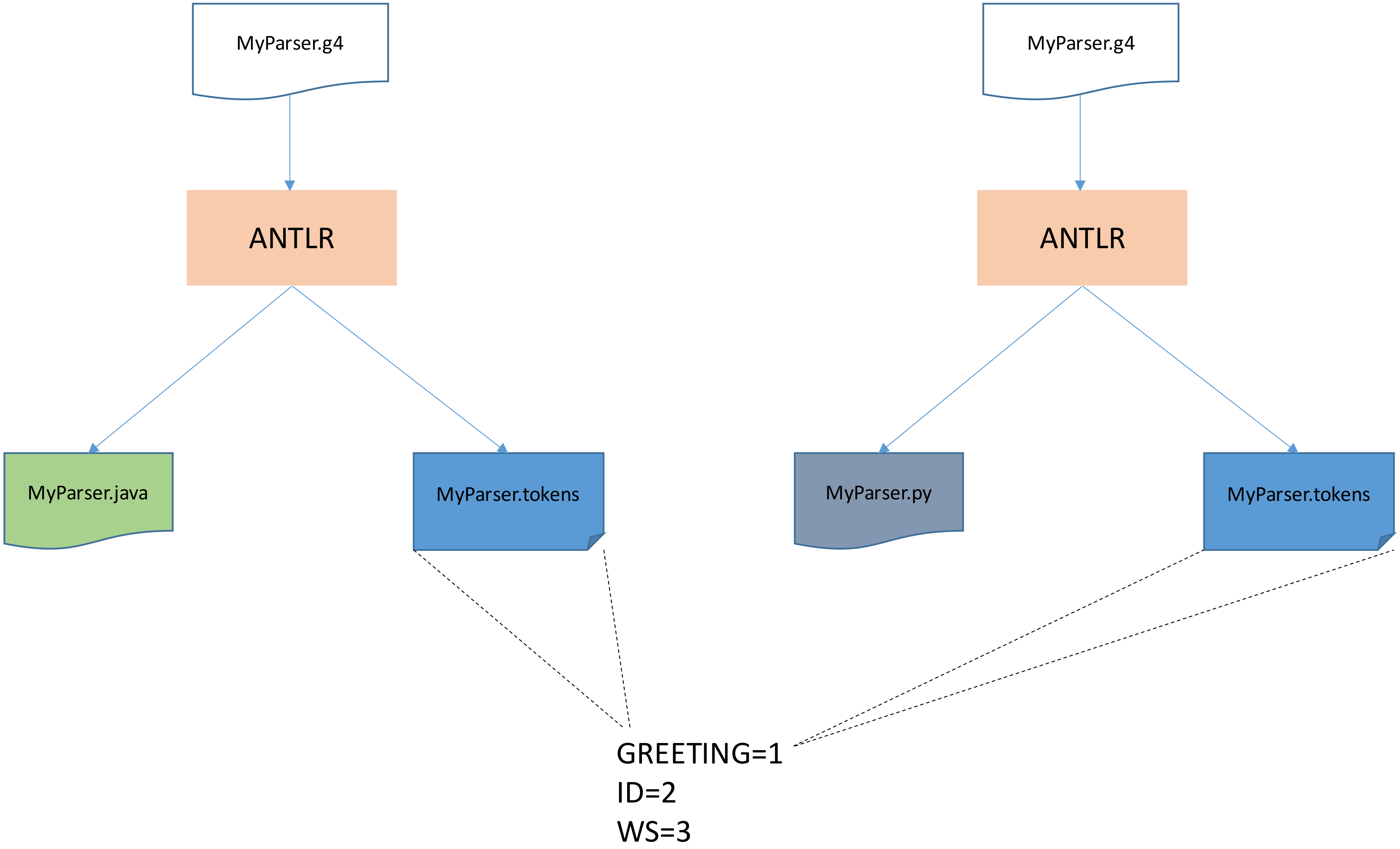

lexer grammar MyLexer ;
GREETING 1 : ('Hello' | 'Greetings') ;
ID 2 : [a-zA-Z]+ ;
WS 3 : [\t\r\n]+ -> skip ;
MyLexer.g4
ANTLR
MyLexer.java
GREETING=1
ID=2
WS=3
lexer grammar MyLexer ;
GREETING 1 : ('Hello' | 'Greetings') ;
ID 2 : [a-zA-Z]+ ;
WS 3 : [\t\r\n]+ -> skip ;
MyLexer.g4
ANTLR
MyLexer.py
GREETING=1
ID=2
WS=3

Then run ANTLR on the parser grammar





MyParser.g4
ANTLR
MyParser.java
MyParser.tokens
MyParser.g4
ANTLR
MyParser.py
MyParser.tokens
This is the generated parser code

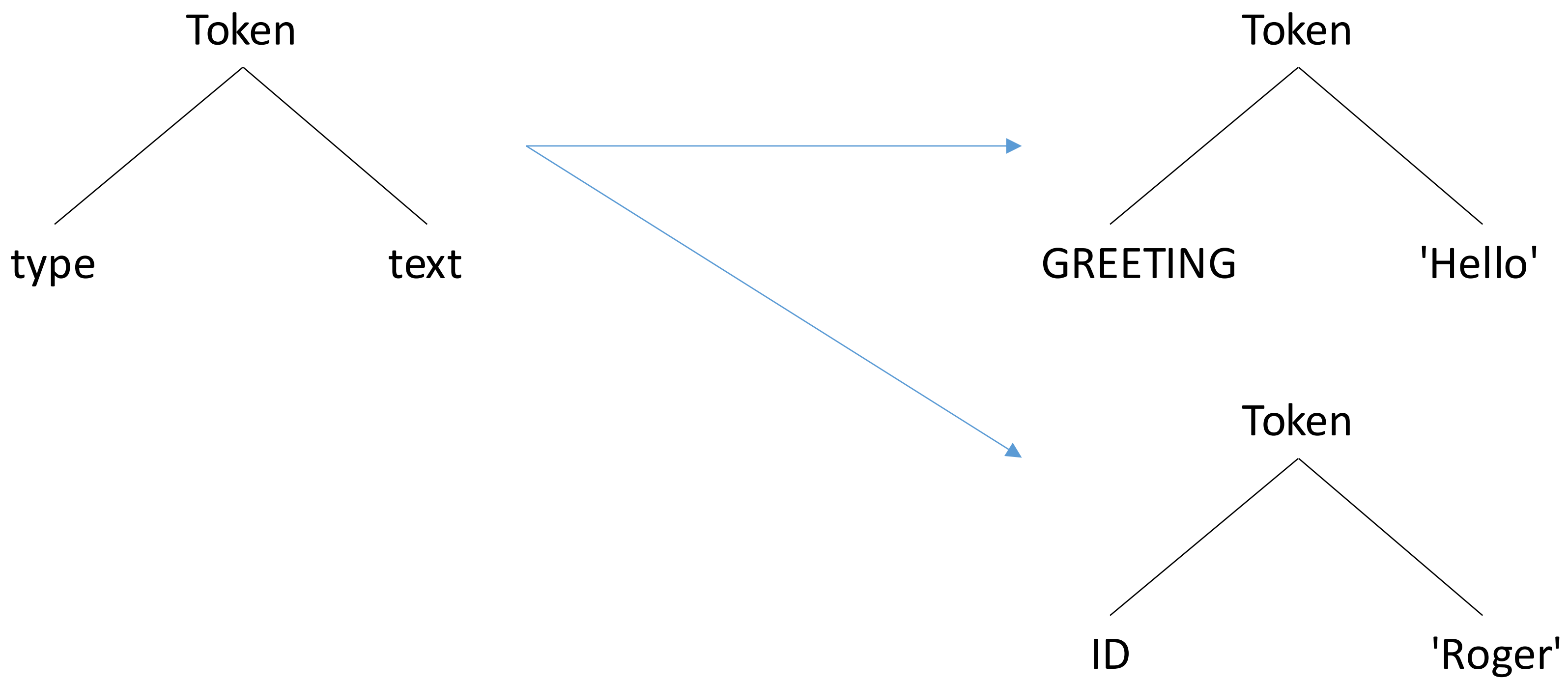

MyParser.g4
ANTLR
MyParser.java
MyParser.tokens
MyParser.g4
ANTLR
MyParser.py
MyParser.tokens
GREETING=1
ID=2
WS=3

Lexical analysis (tokenizing)

Consider how our brains read English text. We don’t read a sentence character by character. Instead, we perceive a sentence as a stream of words. The process of grouping characters into words (tokens) is called *lexical analysis* or simply *tokenizing*. We call a program that tokenizes the input a *lexer*.

Token type

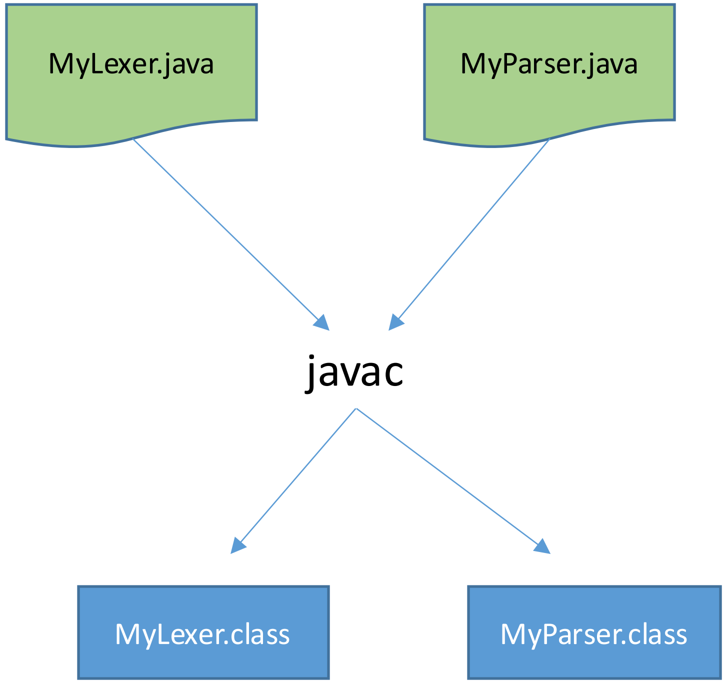
A lexer groups related tokens into token classes, or token types such as GREETING ('Hello' or 'Greetings'), ID (identifier), WS (whitespace), etc.



Parser

A parser receives tokens and determines if they conform to the parser grammar.

Next, compile



Python is an interpreted language. No compiling required.

A "test rig" comes bundled with ANTLR

- Use the test rig to generate a graphic depicting the parse tree.
- I created a batch file, grun.bat, that invokes the test rig.

```
grun My message -gui < input.txt
```

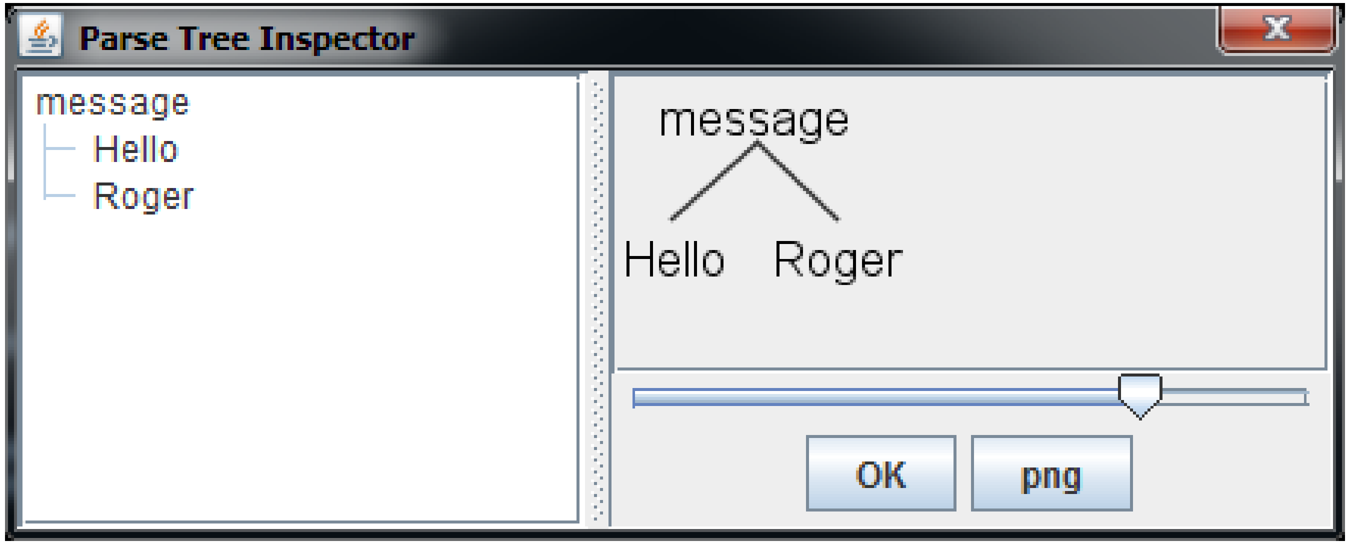
The prefix of the lexer and parser filenames

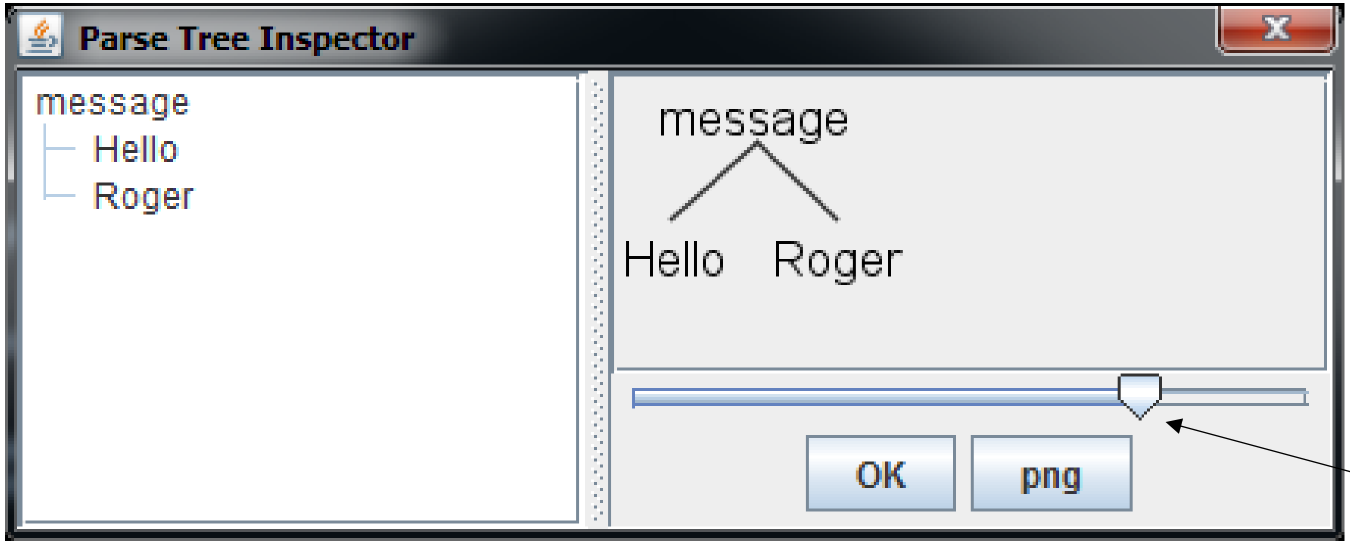
show me a graphic

use the content of this file as input

The starting rule in the parser

grun My message -gui < input.txt





Move this left to make the graphic smaller, right to make it bigger

Comment your grammars!

There are 3 syntaxes for comments:

Javadoc-style comments:

```
/**
 * ... comment ...
 */
```

Multiline comments:

```
/*
   ... comment ...
 */
```

Single-line comments:

```
// ... comment ...
```

MyLexer.g4

```
/**
 * Lexer grammar for a simple greeting
 */
lexer grammar MyLexer;                              // Define a lexer grammar called MyLexer

/* Define three token rules: one for
   the greeting, one for the person's name,
   and one for whitespaces.
 */
GREETING : ('Hello' | 'Greetings') ;                // Match either of these strings
ID  : [a-zA-Z]+ ;                                   // Match lower- and upper-case identifiers
WS  : [ \t\r\n]+ -> skip ;                          // Match any whitespaces and then discard them
```

MyParser.g4

```
/**
 * Parser grammar for a simple greeting
 */
parser grammar MyParser;                       // Define a parser grammar called MyParser

options { tokenVocab=MyLexer; }                // Use the lexer grammar called MyLexer

/* Define one parser rule: a message is
   a greeting followed by a name.
 */
message   : GREETING ID;                       // Match a greeting followed by an identifier
```

run.bat does it all

We have seen that these steps must be taken:

1. Run ANTLR on the lexer grammar
2. Run ANTLR on the parser grammar
3. Where applicable, compile (run javac to compile the Java code)
4. Run the test rig (grun)

I created a batch file – run.bat – which does all those steps. Simply open a command window and type: `run`

See:

java-examples/example01
python-examples/example01

run.bat (for Java target)

```
set CLASSPATH=.;../../antlr-jar/antlr-complete.jar;%CLASSPATH%

echo Running ANTLR on the lexer: MyLexer.g4
java org.antlr.v4.Tool MyLexer.g4 -no-listener -no-visitor

echo Running ANTLR on the parser: MyParser.g4
java org.antlr.v4.Tool MyParser.g4 -no-listener -no-visitor

echo Compiling the Java code that ANTLR generator (the lexer and parser code)
javac *.java

echo Running the test rig on the generated parser, using as input the string in: input.txt
echo And generating a GUI output (i.e., a parse tree graphic)
java org.antlr.v4.gui.TestRig My message -gui < input.txt
```

Python test rig: pygrun

- The creators of ANTLR provide a test rig for Python parsers called pygrun.
- However, pygrun does not provide the ability to generate a GUI.
- So we will just use Java for a while. We will return to generating Python parsers when we discuss *embedding code within the grammar*.

Do Lab1

Other test rig options

grun My message -gui < input.txt

other options: -tokens, -tree, -trace

-tokens

- This option allows you to see what tokens are generated by the lexer.
- So you can create your lexer and test it before creating your parser.

grun MyLexer tokens -tokens < input.txt

This <u>reserved word</u> tells the test rig that only the lexer is to run, not the parser.

Steps to testing the lexer

These are the steps that must be taken:

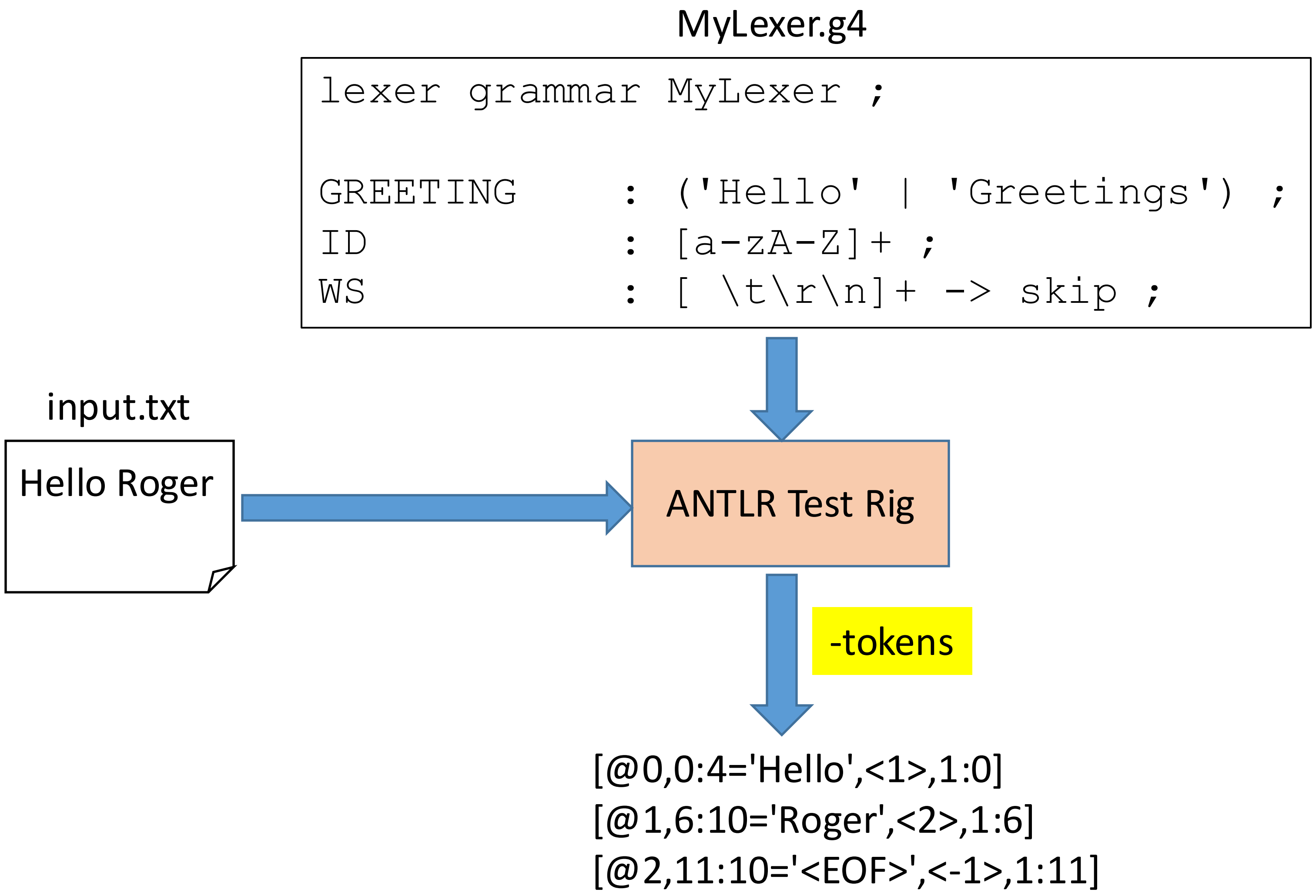
1. Run ANTLR on the lexer grammar
2. Run javac to compile the (lexer) Java code
3. Run the test rig on the lexer (use the `-tokens` option and the `tokens` reserved word)

I created a batch file – run.bat – which does all those steps. Simply open a command window and type: `run`

see java-examples/example02

run.bat

```
set CLASSPATH=.;../../antlr-jar/antlr-complete.jar;%CLASSPATH%

echo Running ANTLR on the lexer: MyLexer.g4
java org.antlr.v4.Tool MyLexer.g4 -no-listener -no-visitor

echo Compiling the Java code that ANTLR generator (the lexer code)
javac *.java

echo Running the test rig on the lexer, using as input the string in: input.txt
echo And generating the token stream
java org.antlr.v4.gui.TestRig MyLexer tokens -tokens < input.txt
```


MyLexer.g4
lexer grammar MyLexer ;
GREETING : ('Hello' | 'Greetings') ;
ID : [a-zA-Z]+ ;
WS : [\t\r\n]+ -> skip ;
input.txt
Hello Roger
ANTLR Test Rig
-tokens
[@0,0:4='Hello',<1>,1:0]
[@1,6:10='Roger',<2>,1:6]
[@2,11:10='<EOF>',<-1>,1:11]

How to read the -token output

Indicates this is the first token recognized (counting starts at 0)

token type

```
[@0,0:4='Hello',<1>,1:0]
```

The token 'Hello' is at position 0 - 4

position of the start of the token

line #

Indicates this is the second token recognized

token type

[@1,6:10='Roger',<2>,1:6]

The token 'Roger'
is at position 6 - 10

position of the start of the token

line #

Indicates this is the third token recognized

token type

```
[@2,11:10='<EOF>',<-1>,1:11]
```

The token 'EOF' is at position 11

position of the start of the token

line #

The test rig -tree option

- This option allows you to see the parse tree in a Lisp-style text form.
- This is sometimes preferable to the `–gui` option since it is text rather than graphical.
- The form of the output is: (*root children*)

```
grun My message -tree < input.txt
```

Steps to using -tree

Here are the steps that must be taken:

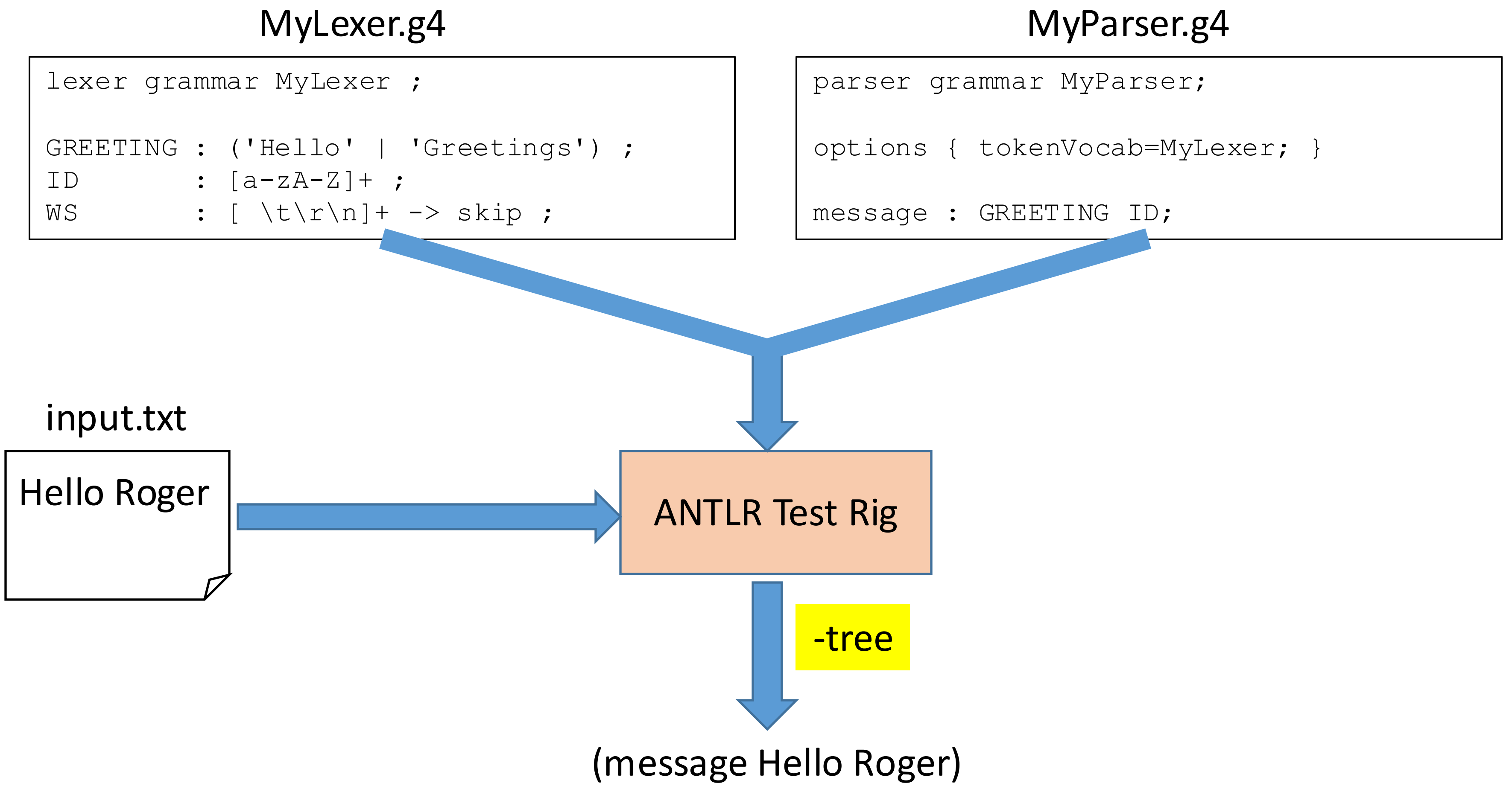
1. Run ANTLR on the lexer grammar
2. Run ANTLR on the parser grammar
3. Run javac to compile the (lexer and parser) Java code
4. Run the test rig on the lexer and parser (use the `-tree` option)

I created a batch file – run.bat – which does all those steps. Simply open a command window and type: `run`

see example03

run.bat

```
set CLASSPATH=.;../../antlr-jar/antlr-complete.jar;%CLASSPATH%

echo Running ANTLR on the lexer: MyLexer.g4
java org.antlr.v4.Tool MyLexer.g4 -no-listener -no-visitor

echo Running ANTLR on the parser: MyParser.g4
java org.antlr.v4.Tool MyParser.g4 -no-listener -no-visitor

echo Compiling the Java code that ANTLR generator (the lexer and parser code)
javac *.java

echo Running the test rig on the generated parser, using as input the string in: input.txt
echo And generating a tree output (i.e., a Lisp-style text form)
java org.antlr.v4.gui.TestRig My message -tree < input.txt
```


MyLexer.g4
lexer grammar MyLexer ;
GREETING : ('Hello' | 'Greetings') ;
ID : [a-zA-Z]+ ;
WS : [\t\r\n]+ -> skip ;
MyParser.g4
parser grammar MyParser;
options { tokenVocab=MyLexer; }
message : GREETING ID;
input.txt
Hello Roger
ANTLR Test Rig
-tree
(message Hello Roger)

The test rig -trace option

This option allows you to see how ANTLR generates the tokens and executes the parser rules.

```
grun My message -trace < input.txt
```

Steps to testing -trace

Here are the steps that must be taken:

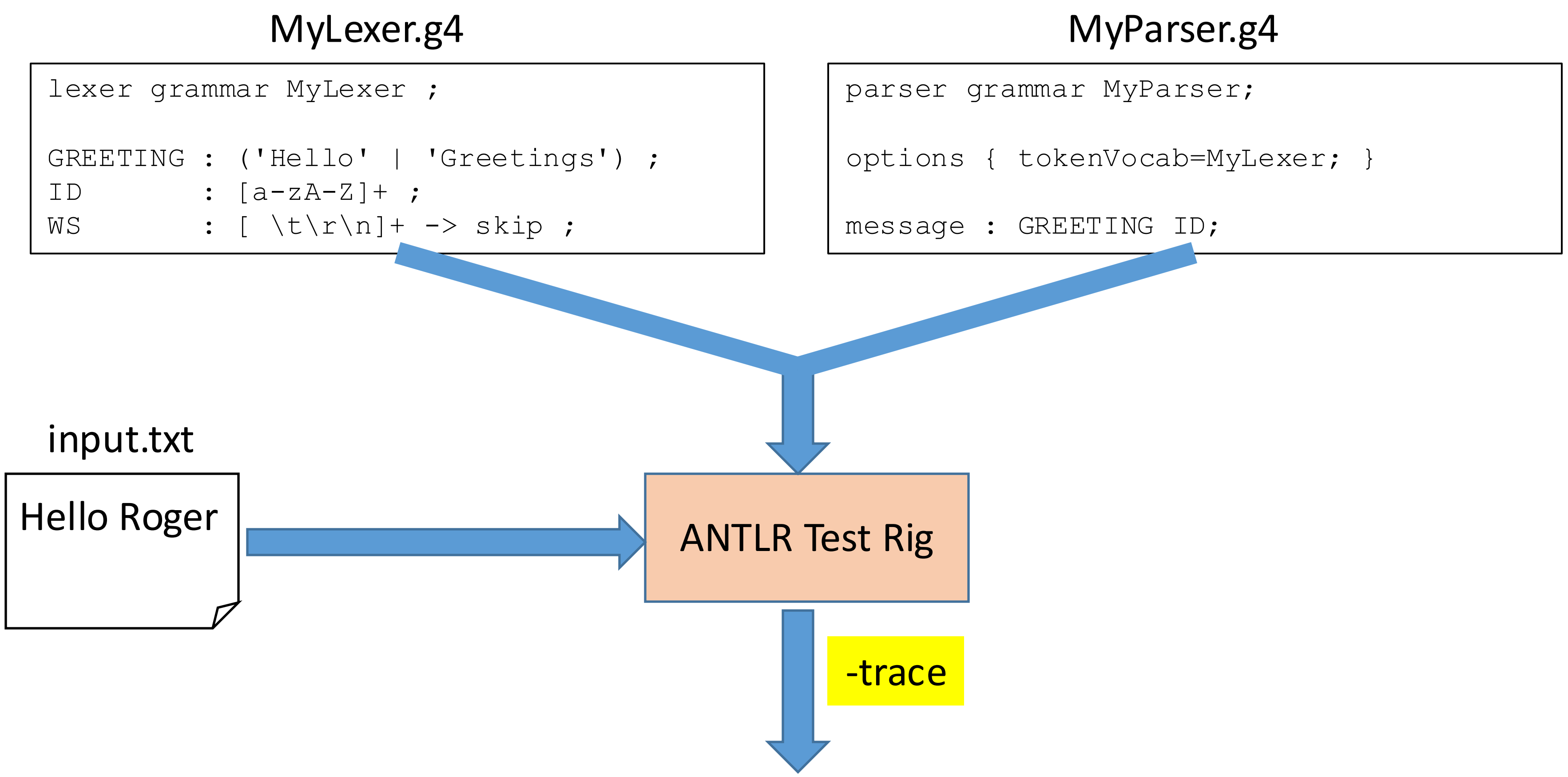
1. Run ANTLR on the lexer grammar
2. Run ANTLR on the parser grammar
3. Run javac to compile the (lexer and parser) Java code
4. Run the test rig on the lexer and parser (use the `–trace` option)

I created a batch file – run.bat – which does all those steps. Simply open a command window and type: `run`

see example04

run.bat

```
set CLASSPATH=.;../../antlr-jar/antlr-complete.jar;%CLASSPATH%

echo Running ANTLR on the lexer: MyLexer.g4
java org.antlr.v4.Tool MyLexer.g4 -no-listener -no-visitor

echo Running ANTLR on the parser: MyParser.g4
java org.antlr.v4.Tool MyParser.g4 -no-listener -no-visitor

echo Compiling the Java code that ANTLR generator (the lexer and parser code)
javac *.java

echo Running the test rig on the generated parser, using as input the string in: input.txt
echo And generating an execution trace
java org.antlr.v4.gui.TestRig My message -trace < input.txt
```




```
enter   message, LT(1)=Hello
consume [@0,0:4='Hello',<1>,1:0] rule message
consume [@1,6:10='Roger',<2>,1:6] rule message
exit    message, LT(1)=<EOF>
```

pygrun supports tree and token flags

python pygrun.py -t My message input.txt

tree

(message Hello Roger)

python pygrun.py -k My message input.txt

token

[@0,0:4='Hello',<1>,1:0]
[@1,6:10='Roger',<2>,1:6]
[@2,11:10='<EOF>',<-1>,1:11]

see python-examples/example02

Do Lab2

The + subrule operator

`+` means "one or more occurrences"

`(INT)+` means "one or more `INT` tokens"

As a shorthand, `INT+` is also okay

Predefined token: EOF

- The EOF token is built into ANTLR, i.e., you don't have to specify the EOF token in your lexer grammar.
- At the end of every input is an EOF token.
- Here is a parser rule that explicitly specifies the EOF token in a parser rule:

```
message : GREETING ID EOF ;
```

Which of these parser rules is better?

```
message : GREETING ID;
```

or

```
message : GREETING ID EOF ;
```

```
message : GREETING ID;
```

or

```
message : GREETING ID EOF ;
```

This one is better. It says that the input must contain a GREETING followed by an ID followed by end-of-file (EOF).

More about this issue →

Trailing Garbage in the Input

Extra data not detected

MyLexer.g4

```
lexer grammar MyLexer ;

GREETING : ('Hello' | 'Greetings') ;
ID       : [a-zA-Z]+ ;
WS       : [ \t\r\n]+ -> skip ;
```

MyParser.g4

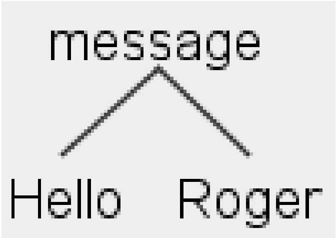
```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

message : GREETING ID;
```

input.txt

Hello Roger Garbage

ANTLR Test Rig

-gui



see example05

Revise the parser grammar

MyParser.g4

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

message : GREETING ID EOF;
```

The (predefined) EOF token must follow the ID token (and nothing else)

Now extra data is detected

MyLexer.g4

```
lexer grammar MyLexer ;

GREETING : ('Hello' | 'Greetings') ;
ID       : [a-zA-Z]+ ;
WS       : [ \t\r\n]+ -> skip ;
```

MyParser.g4

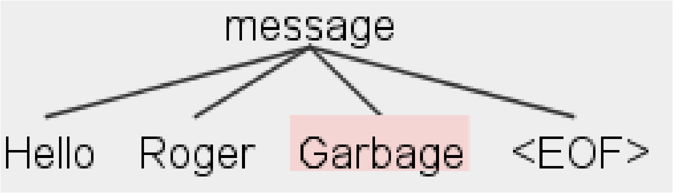
```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

message : GREETING ID EOF;
```

input.txt

Hello Roger Garbage

ANTLR Test Rig

-gui



line 1:12 extraneous input 'Garbage', expecting <EOF>

Do Lab3

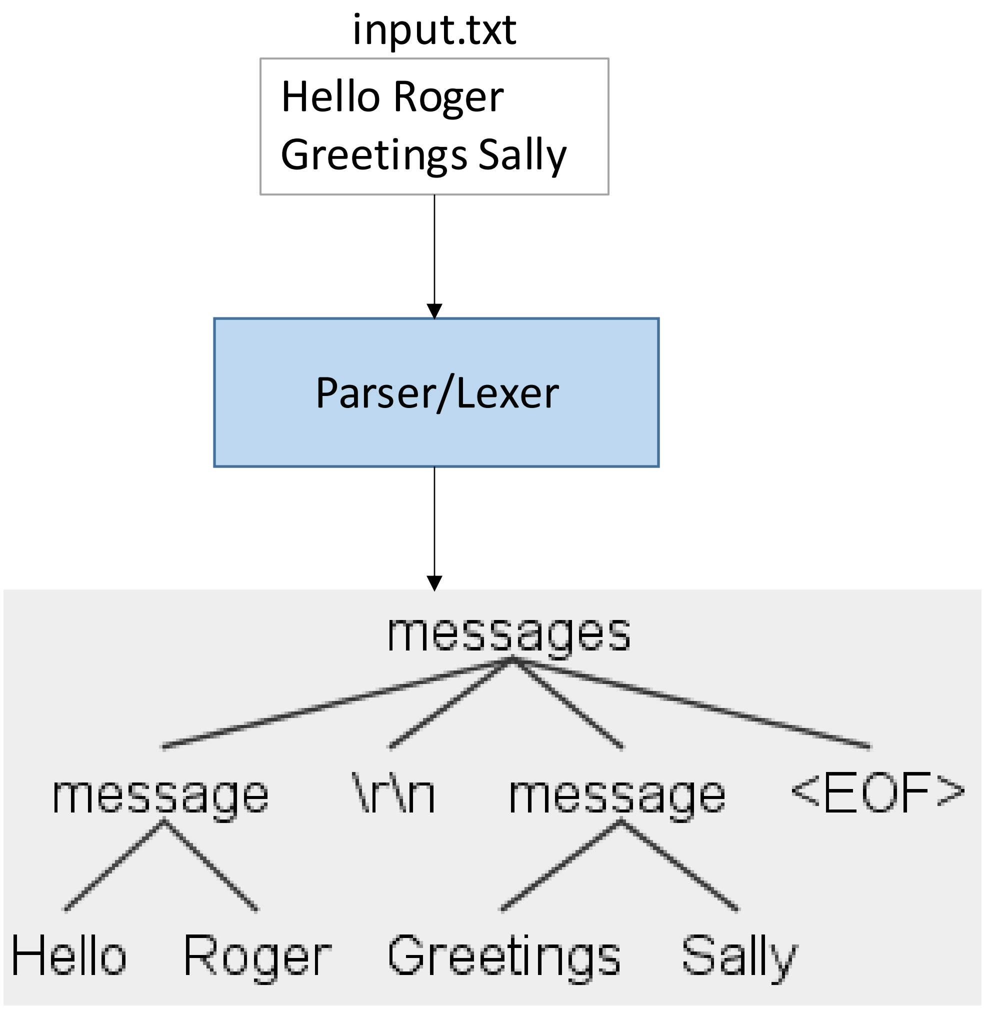
see example06

Our second parser

The input consists of a series (at least one) of greetings. Each greeting is on a new line. After the last greeting there may, or may not, be a new line. Create a parser. Example input:

input.txt

```
Hello Roger
Greetings Sally
```


input.txt
Hello Roger
Greetings Sally
Parser/Lexer
messages
message
\r\n
message
<EOF>
Hello
Roger
Greetings
Sally

The newline token

- The input consists of a series (at least one) of greetings. Each greeting is on a new line...
- In Unix the newline is \n and in Windows the newline is \r\n.
- So the lexer rule for newline (NL) is: an optional \r followed by \n.

```
NL : ('\r')?'\n' ;
```

question mark means "optional"

MyLexer.g4

```
lexer grammar MyLexer;

GREETING : ('Hello' | 'Greetings') ;
ID       : [a-zA-Z]+ ;
NL       : ('\r')?'\n' ;
WS       : [ \t\r\n]+ -> skip ;
```

MyParser.g4

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

messages  : (message NL)* message (NL)? EOF;
message   : GREETING ID;
```

MyParser.g4

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

messages   :  (message NL)* message  (NL)? EOF;
message    : GREETING ID;
```

`messages` is defined as zero or more (`message NL`) pairs followed by a `message`, an optional `NL`, and `EOF`.

Start rule

MyParser.g4

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

messages   :  (message NL)* message  (NL)? EOF;
message    : GREETING ID;
```

We want parsing to proceed using this rule as the starting point

MyLexer.g4

```
lexer grammar MyLexer;

GREETING : ('Hello' | 'Greetings') ;
ID       : [a-zA-Z]+ ;
NL       : ('\r')?'\n' ;
WS       : [ \t\r\n]+ -> skip ;
```

MyParser.g4

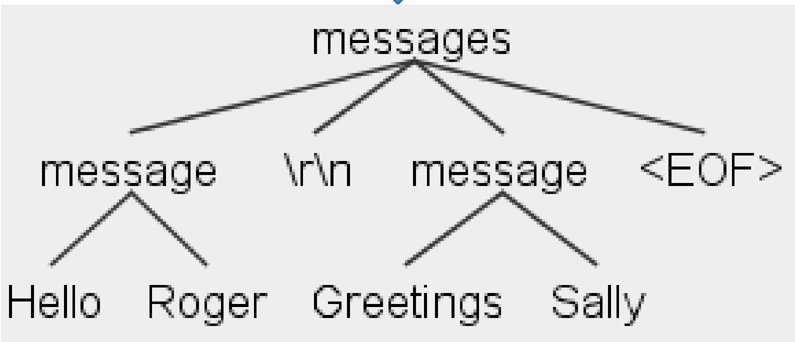
```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

messages   : (message NL)* message (NL)? EOF;
message    : GREETING ID;
```

input.txt

```
Hello Roger
Greetings Sally
```

ANTLR Test Rig

-gui



see example07

Any rule can be the start rule

MyParser.g4

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

messages   : (message NL)* message (NL)? EOF;
message    : GREETING ID;
```

Either of these could be used as the start rule

run.bat

```
set CLASSPATH=.;../../antlr-jar/antlr-complete.jar;%CLASSPATH%

echo Running ANTLR on the lexer: MyLexer.g4
java org.antlr.v4.Tool MyLexer.g4 -no-listener -no-visitor

echo Running ANTLR on the parser: MyParser.g4
java org.antlr.v4.Tool MyParser.g4 -no-listener -no-visitor

echo Compiling the Java code that ANTLR generator (the lexer and parser code)
javac *.java

echo Running the test rig on the generated parser, using as input the string in: input.txt
echo And generating a GUI output (i.e., a parse tree graphic)
java org.antlr.v4.gui.TestRig My messages -gui < input.txt
```

Specify the start rule here

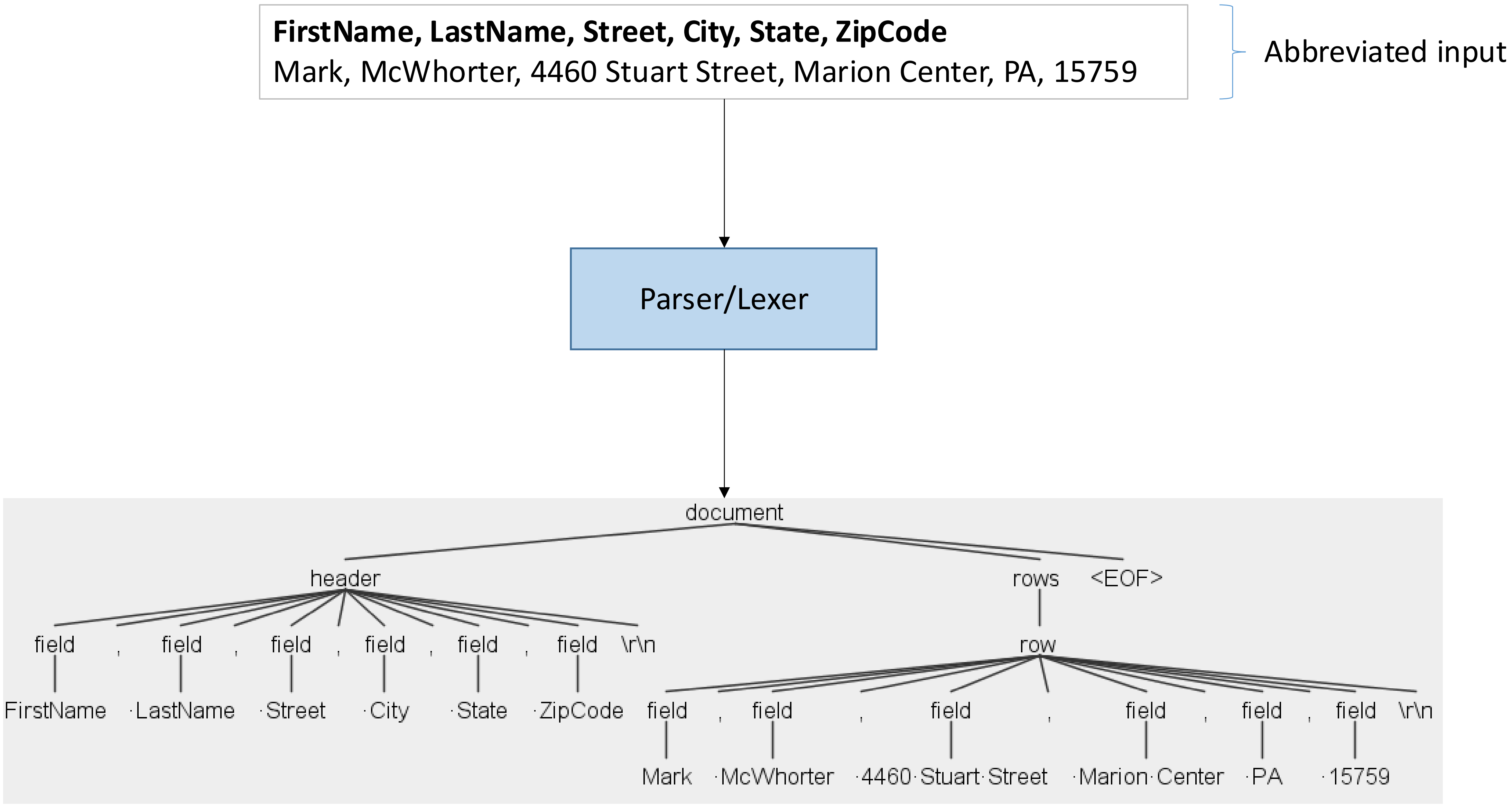
Do Lab4

Our third parser

The input is a CSV file (comma-separated-values file): a series of rows separated by newlines; each row consists of a series of fields separated by commas; a field is a string (any character except comma and newline). The fields in the first row are the column headers. Create a parser. Example input:

input.txt

```
FirstName, LastName, Street, City, State, ZipCode
Mark, McWhorter, 4460 Stuart Street, Marion Center, PA, 15759
Monica, Apodaca, 258 Oliver Street, Fort Worth, TX, 76118
Kenisha, Le, 1579 Olen Thomas Drive, Charlie, TX, 76377
Dustin, Edwards, 2249 Ingram Road, Greensboro, NC, 27401
Ruth, Heath, 988 Davisson Street, Fairmont, IN, 46928
Christin, Dehart, 515 Ash Avenue, Saint Louis, MO, 63108
```


FirstName, LastName, Street, City, State, ZipCode
Mark, McWhorter, 4460 Stuart Street, Marion Center, PA, 15759
Abbreviated input
Parser/Lexer
document
header
rows
<EOF>
field , field , field , field , field , field \r\n
FirstName ·LastName ·Street ·City ·State ·ZipCode
row
field , field , field , field , field , field \r\n
Mark ·McWhorter ·4460·Stuart·Street ·Marion·Center ·PA ·15759

The parser grammar

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

document  : header rows EOF ;
header    : field (COMMA field)* NL ;
rows      : (row)* ;
row       : field (COMMA field)* NL ;
field     : STRING | ;
```

MyParser.g4

See example08

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

document   : header rows EOF ;
header     : field (COMMA field)* NL ;
rows       : (row)* ;
row        : field (COMMA field)* NL ;
field      : STRING | ;
```

input.txt

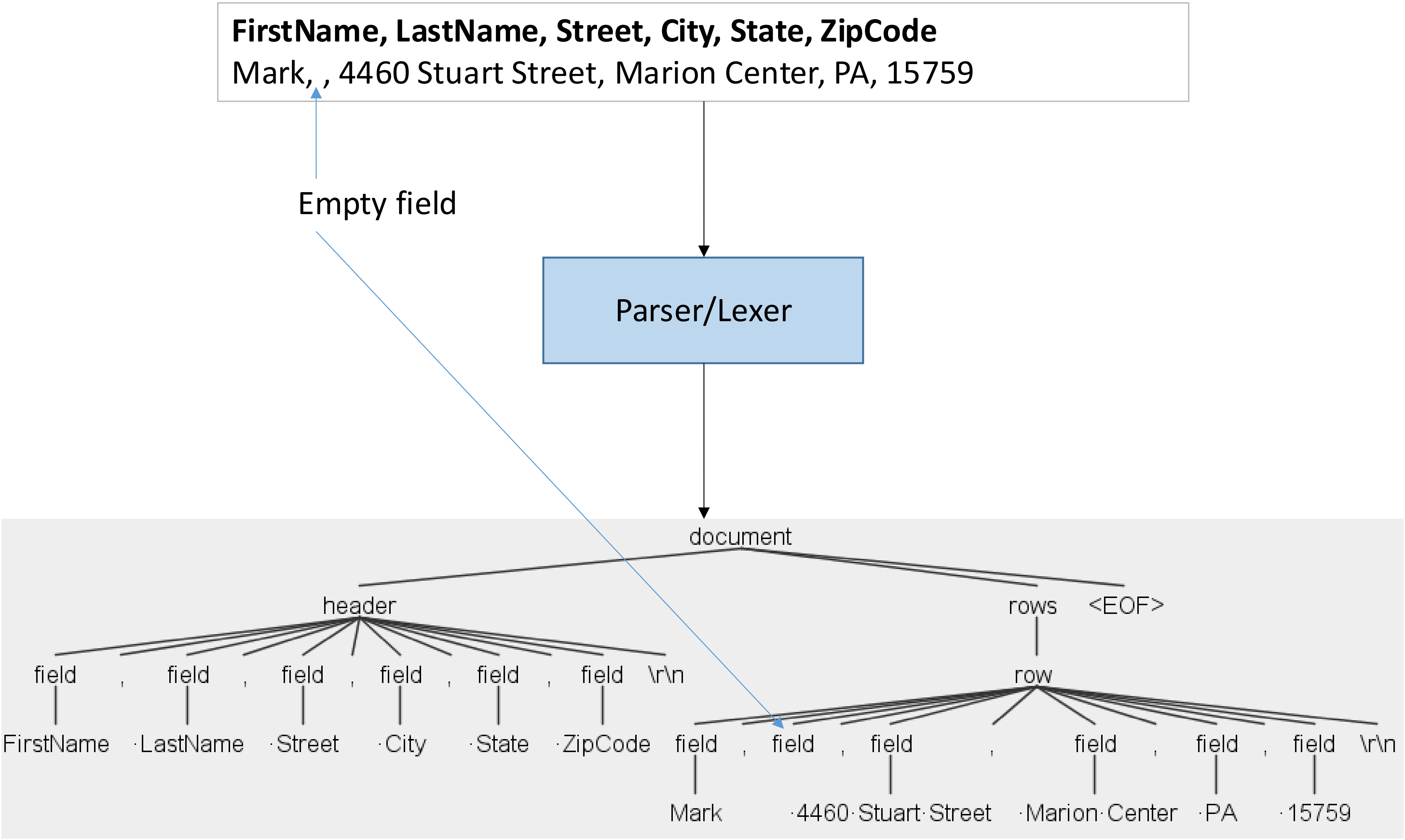
```
FirstName, LastName, Street, City, State, ZipCode
Mark, McWhorter, 4460 Stuart Street, Marion Center, PA, 15759
Monica, Apodaca, 258 Oliver Street, Fort Worth, TX, 76118
Kenisha, Le, 1579 Olen Thomas Drive, Charlie, TX, 76377
Dustin, Edwards, 2249 Ingram Road, Greensboro, NC, 27401
Ruth, Heath, 988 Davisson Street, Fairmont, IN, 46928
Christin, Dehart, 515 Ash Avenue, Saint Louis, MO, 63108
```

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

document   : header rows EOF ;
header     : field (COMMA field)* NL ;
rows       : (row)* ;
row        : field (COMMA field)* NL ;
field      : STRING | ;
```

FirstName, LastName, Street, City, State, ZipCode
Mark, McWhorter, 4460 Stuart Street, Marion Center, PA, 15759
Monica, Apodaca, 258 Oliver Street, Fort Worth, TX, 76118
Kenisha, Le, 1579 Olen Thomas Drive, Charlie, TX, 76377
Dustin, Edwards, 2249 Ingram Road, Greensboro, NC, 27401
Ruth, Heath, 988 Davisson Street, Fairmont, IN, 46928
Christin, Dehart, 515 Ash Avenue, Saint Louis, MO, 63108

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

document   : header rows EOF ;
header     : field (COMMA field)* NL ;
rows       : (row)* ;
row        : field (COMMA field)* NL ;
field      : STRING | ;
```

Empty (i.e., an input field can be empty)



FirstName, LastName, Street, City, State, ZipCode
Mark, , 4460 Stuart Street, Marion Center, PA, 15759
Empty field
Parser/Lexer
document
header
rows
<EOF>
row
field , field , field , field , field , field \r\n
FirstName ·LastName ·Street ·City ·State ·ZipCode
field , field , field , field , field , field \r\n
Mark ·4460·Stuart·Street ·Marion·Center ·PA ·15759

The lexer grammar

```
lexer grammar MyLexer;

COMMA  : ',' ;
NL     : ('\r')?'\n' ;
WS     : [ \t\r\n]+ -> skip ;
STRING : (~[,\r\n])+ ;
```

MyLexer.g4

The "not" operator

```
lexer grammar MyLexer;

COMMA  : ',' ;
NL     : ('\r')?'\n' ;
WS     : [ \t\r\n]+ -> skip ;
STRING : (~[,\r\n])+ ;
```

Tilda (~) means "not". This lexer rule says: a STRING is one or more characters. Each character in STRING is not a comma, carriage return (\r), or newline (\n).

Evaluation order: first-to-last

The first lexer rule is tested on the input. If there is no match, the second lexer rule is tested on the input. If there is no match, the third lexer rule is test on the input. And so forth. If, after trying all the lexer rules on the input there is no match, an error is generated.

```
lexer grammar MyLexer;

COMMA  : ',' ;
NL     : ('\r')?'\n' ;
WS     : [ \t\r\n]+ -> skip ;
STRING : (~[,\r\n])+ ;
```

Order of lexer rules matter!

Suppose that the next symbol in the input is 'apple'. The first lexer rule is tested on the input. There is a match; thus, the lexer sends up to the parser the pair (FRUIT, 'apple'). **Right!**

```
lexer grammar MyLexer;

FRUIT        : 'apple' ;
WS           : [ \t\r\n]+ -> skip ;
VEGETABLE    : (.)+ ;
```

Move the VEGETABLE lexer rule to the first position

```
lexer grammar MyLexer;

VEGETABLE    : (.)+ ;
FRUIT        : 'apple' ;
WS           : [ \t\r\n]+ -> skip ;
```

Now the input matches against VEGETABLE; thus, the lexer sends up to the parser the pair (VEGETABLE, 'apple'). **Wrong!**

Note: the dot (.) in a lexer rule means one ***character***.

Do Lab5

Our fourth parser

The input is an address: number, street, and zipcode. Create a parser.
Example input:

input.txt

```
4460 Stuart Street 15759
```

The parser grammar

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

address   : street zipcode EOF ;
street    : NUMBER ROAD ;
zipcode   : ZIPCODE ;
```

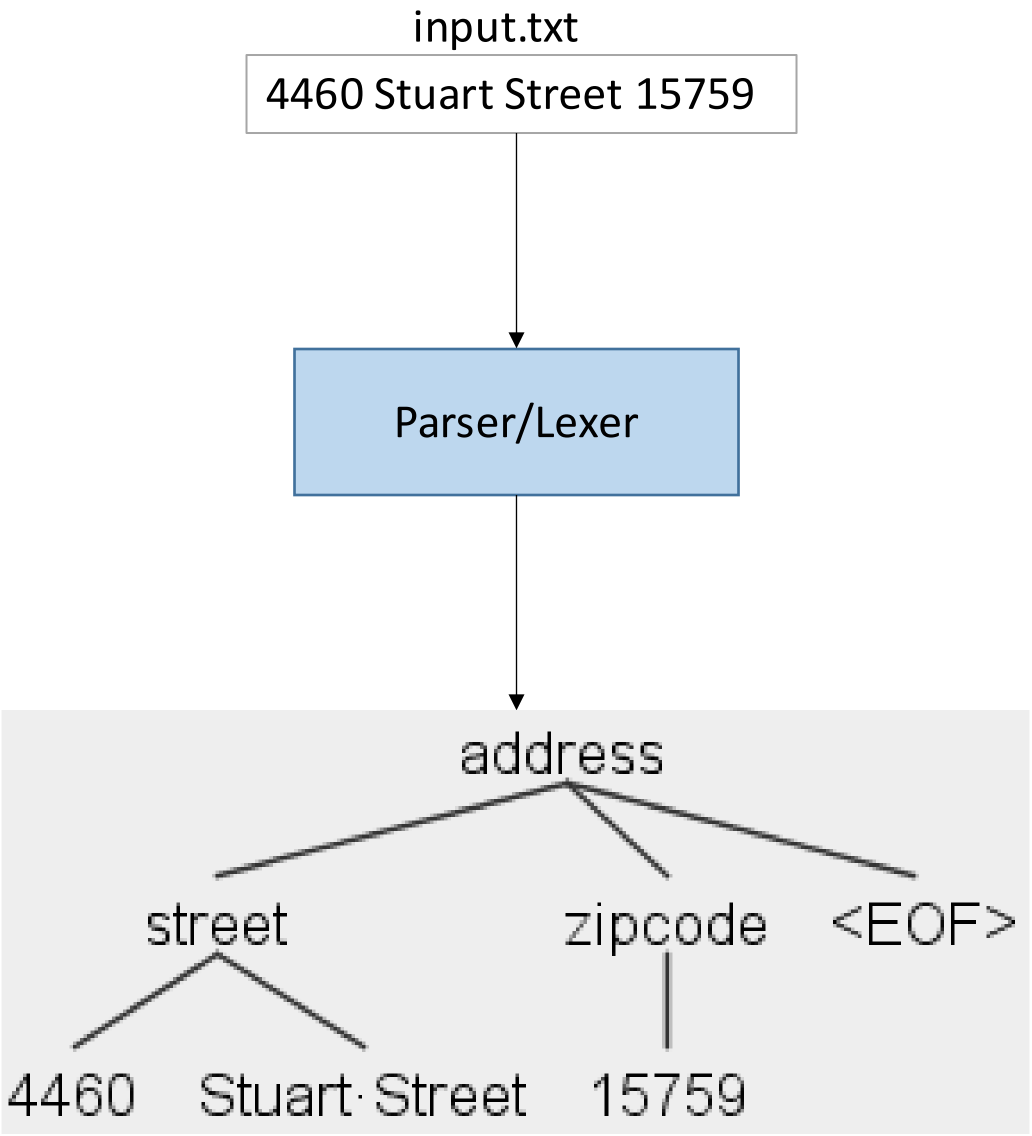
See example09

The lexer grammar

```
lexer grammar MyLexer;

ZIPCODE : [0-9][0-9][0-9][0-9][0-9] ;
NUMBER  : [0-9]+ ;
ROAD    : [a-zA-Z][a-zA-Z ]+[a-zA-Z] ;
WS     : [ \t\r\n]+ -> skip ;
```

```
lexer grammar MyLexer;

ZIPCODE : [0-9][0-9][0-9][0-9][0-9] ;
NUMBER  : [0-9]+ ;
ROAD    : [a-zA-Z][a-zA-Z ]+[a-zA-Z] ;
WS     : [ \t\r\n]+ -> skip ;
```

This one must be listed before NUMBER.


input.txt
4460 Stuart Street 15759
Parser/Lexer
address
street
zipcode
<EOF>
4460
Stuart·Street
15759

Our fifth parser

Same input as with last parser:

input.txt

```
4460 Stuart Street 15759
```

The parser grammar

Vertical bar (|) means 'or' (alternative/choice)

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

address    : NUMBER ROAD ZIPCODE EOF
           | street zipcode EOF
           ;
street     : NUMBER ROAD ;
zipcode    : ZIPCODE ;
```

See example10

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

address    : NUMBER ROAD ZIPCODE EOF
           | street zipcode EOF
           ;
street     : NUMBER ROAD ;
zipcode    : ZIPCODE ;
```

```
lexer grammar MyLexer;

ZIPCODE : [0-9][0-9][0-9][0-9][0-9] ;
NUMBER  : [0-9]+ ;
ROAD    : [a-zA-Z][a-zA-Z ]+[a-zA-Z] ;
WS     : [ \t\r\n]+ -> skip ;
```

Input: 4460 Stuart Street 15759

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

address    : NUMBER ROAD ZIPCODE EOF
           | street zipcode EOF
           ;
street     : NUMBER ROAD ;
zipcode    : ZIPCODE ;
```

(NUMBER, 4460)

First token sent up to the parser is the NUMBER token.

```
lexer grammar MyLexer;

ZIPCODE : [0-9][0-9][0-9][0-9][0-9] ;
NUMBER  : [0-9]+ ;
ROAD    : [a-zA-Z][a-zA-Z ]+[a-zA-Z] ;
WS     : [ \t\r\n]+ -> skip ;
```

4460 Stuart Street 15759

Start rule

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

address     : NUMBER ROAD ZIPCODE EOF
            | street zipcode EOF
            ;
street      : NUMBER ROAD ;
zipcode     : ZIPCODE ;
```

Which rule does the parser use for the NUMBER token?

(NUMBER, 4460)

```
lexer grammar MyLexer;

ZIPCODE : [0-9][0-9][0-9][0-9][0-9] ;
NUMBER  : [0-9]+ ;
ROAD    : [a-zA-Z][a-zA-Z ]+[a-zA-Z] ;
WS     : [ \t\r\n]+ -> skip ;
```

4460 Stuart Street 15759

I switched the order of the alternatives

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

address    : street zipcode EOF
           | NUMBER ROAD ZIPCODE EOF
           ;
street     : NUMBER ROAD ;
zipcode    : ZIPCODE ;
```

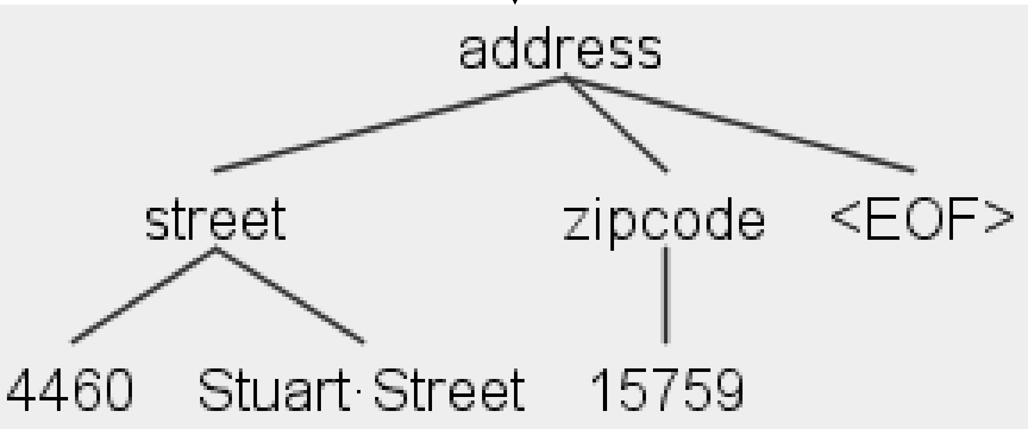
```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

address    : NUMBER ROAD ZIPCODE EOF
           | street zipcode EOF
           ;
street     : NUMBER ROAD ;
zipcode    : ZIPCODE ;
```

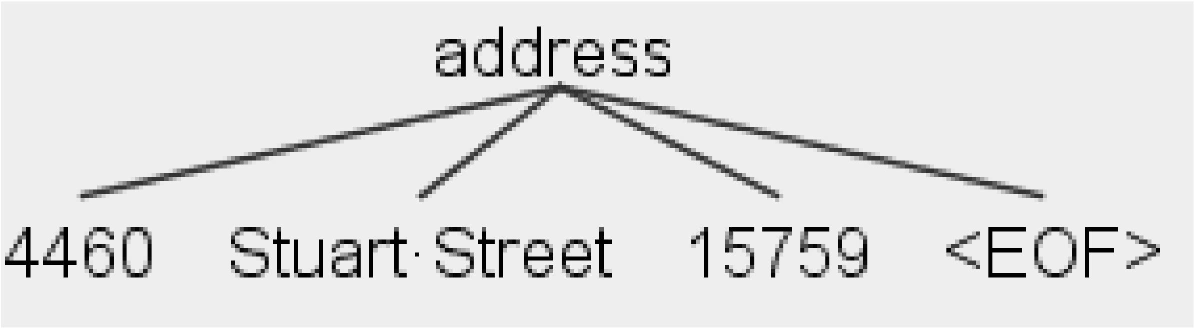
See example10

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

address     : street zipcode EOF
            | NUMBER ROAD ZIPCODE EOF
            ;
street      : NUMBER ROAD ;
zipcode     : ZIPCODE ;
```

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

address     : NUMBER ROAD ZIPCODE EOF
            | street zipcode EOF
            ;
street      : NUMBER ROAD ;
zipcode     : ZIPCODE ;
```

Two different parse trees





Lessons Learned

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

address    : street zipcode EOF
           | NUMBER ROAD ZIPCODE EOF
           ;
street     : NUMBER ROAD ;
zipcode    : ZIPCODE ;
```

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

address    : NUMBER ROAD ZIPCODE EOF
           | street zipcode EOF
           ;
street     : NUMBER ROAD ;
zipcode    : ZIPCODE ;
```

1. In the `address` rule both alternatives are viable.
2. When a rule has two alternatives and both are viable, ANTLR always chooses the <u>first</u> alternative.
3. If an input string could be derived in multiple ways, the grammar is *ambiguous*. *Avoid ambiguous grammars*.

Ambiguous grammars are bad

- The meaning of input is implied by the structure of the parse tree.
- Thus, multiple parse trees indicate that the input has multiple meanings.
- Applications cannot process data with variable meaning.

The meaning of input is implied by the structure of the parse tree.

Do Lab6

How the lexer chooses token rules

Create a parser and lexer for this input

- The input contains a person's name.
- We want to know if the name is: Ken.
- We want the lexer to send up to the parser the token type KEN if the input is 'Ken' and the token type OTHER for any other name.

The lexer grammar

```
lexer grammar MyLexer;

KEN     : 'Ken' ;
OTHER   : [a-zA-Z]+ ;
WS      : [ \t\r\n]+ -> skip ;
```

The more specific rule is listed before the more general rule

```
lexer grammar MyLexer;

KEN     : 'Ken' ;
OTHER   : [a-zA-Z]+ ;
WS      : [ \t\r\n]+ -> skip ;
```

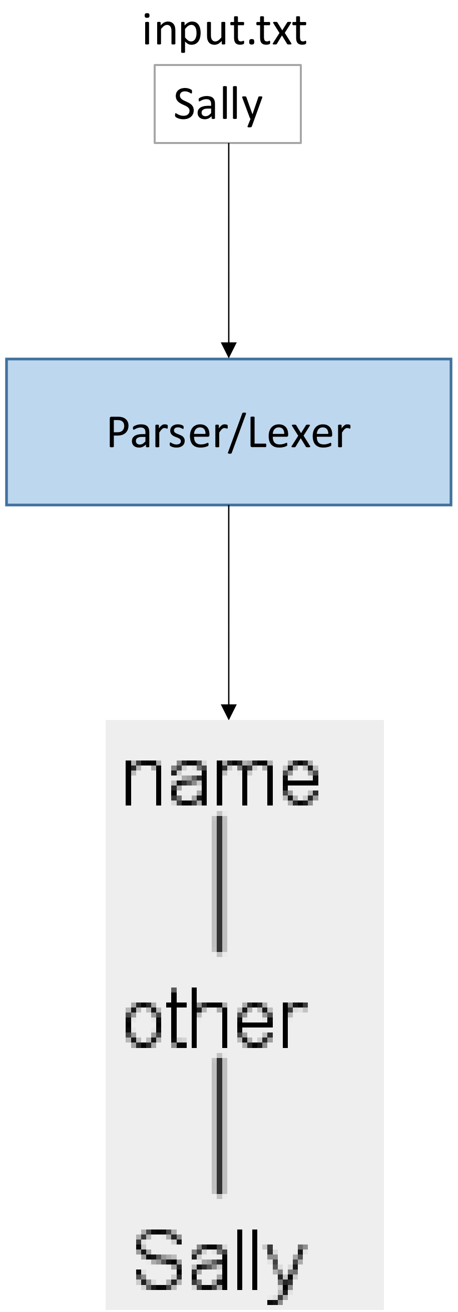
The parser grammar

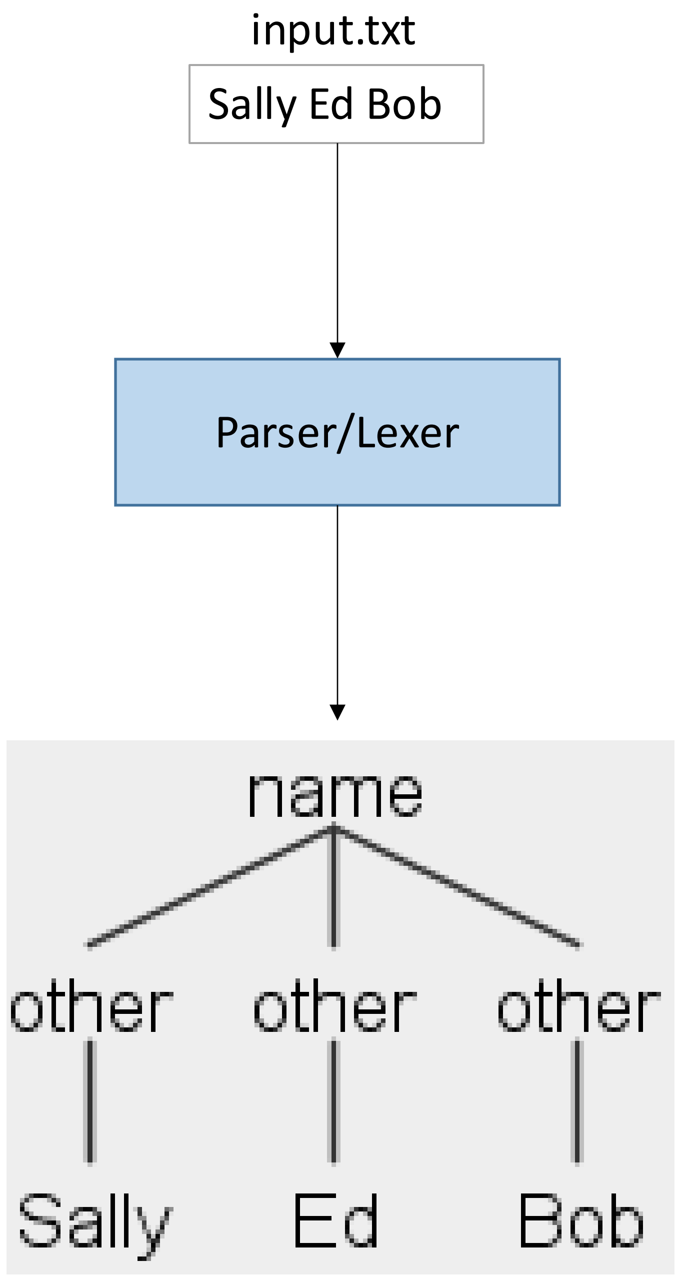
```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

name  : (ken | other)+ ;

ken   : KEN ;

other : OTHER ;
```

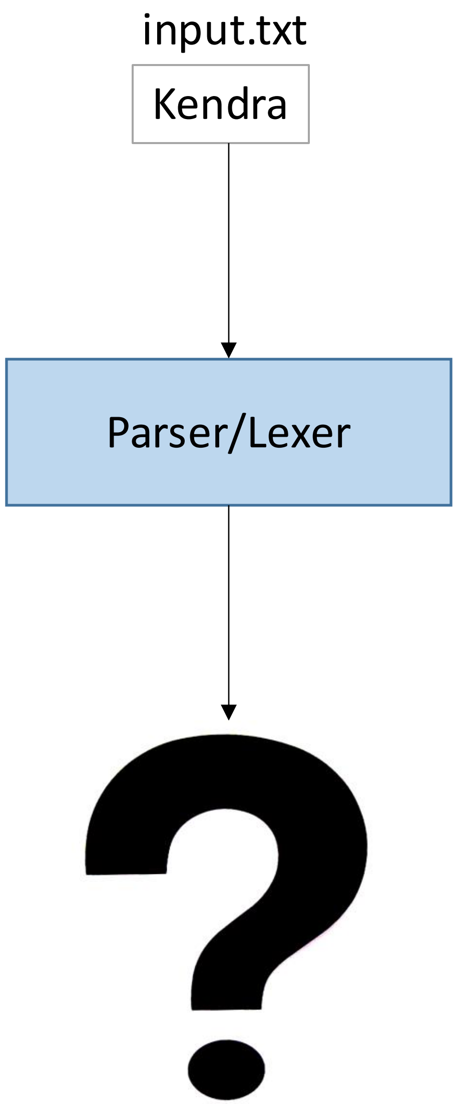
See example11

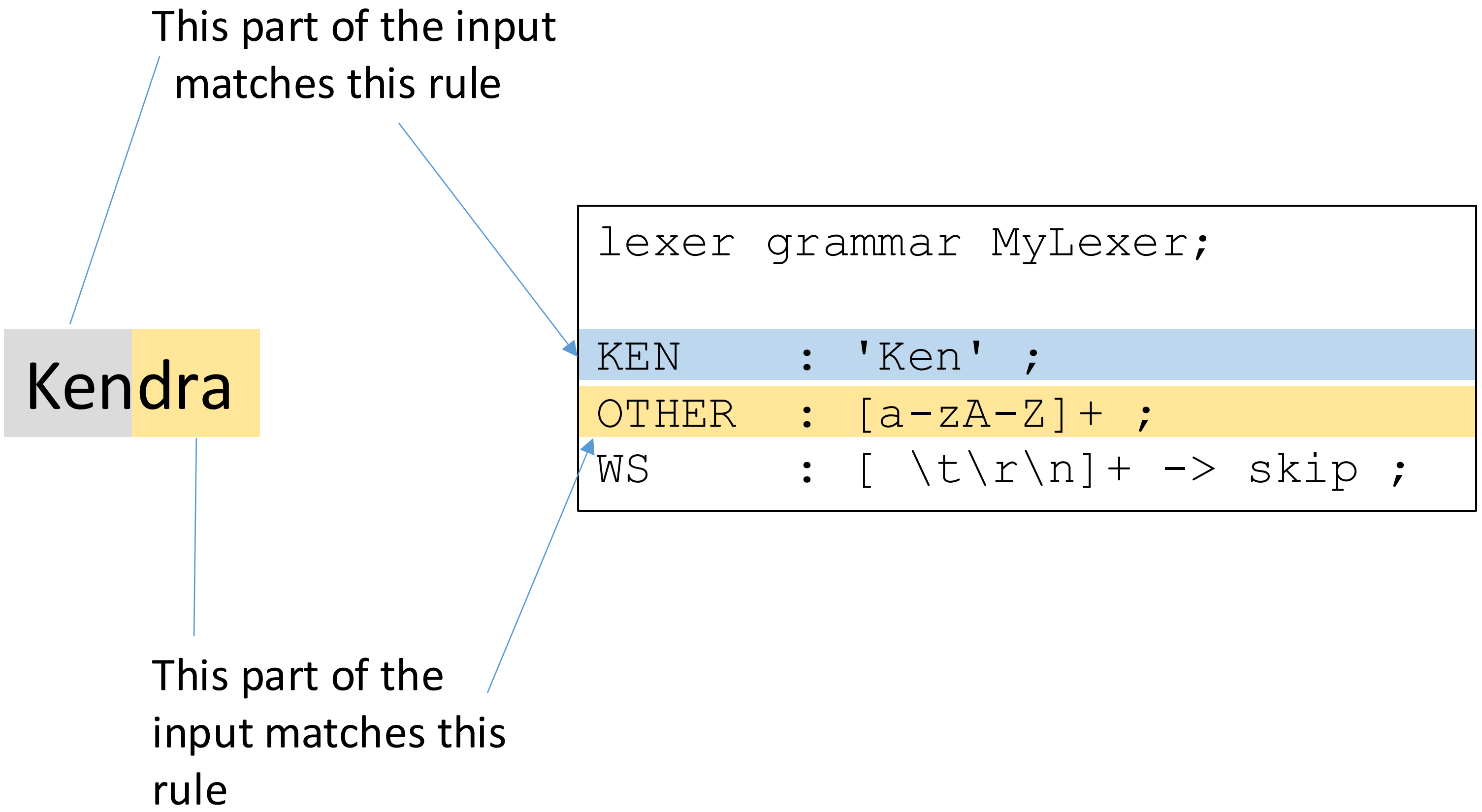
```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

name  : (ken | other)+ ;

ken   : KEN ;

other : OTHER ;
```

Input must contain at least one name.


input.txt
Sally
Parser/Lexer
name
other
Sally

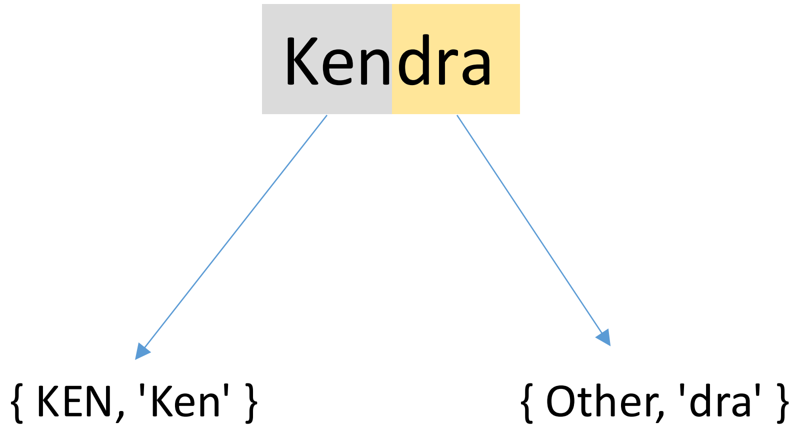

input.txt
Sally Ed Bob
Parser/Lexer
name
other
other
other
Sally
Ed
Bob


input.txt
Kendra
Parser/Lexer

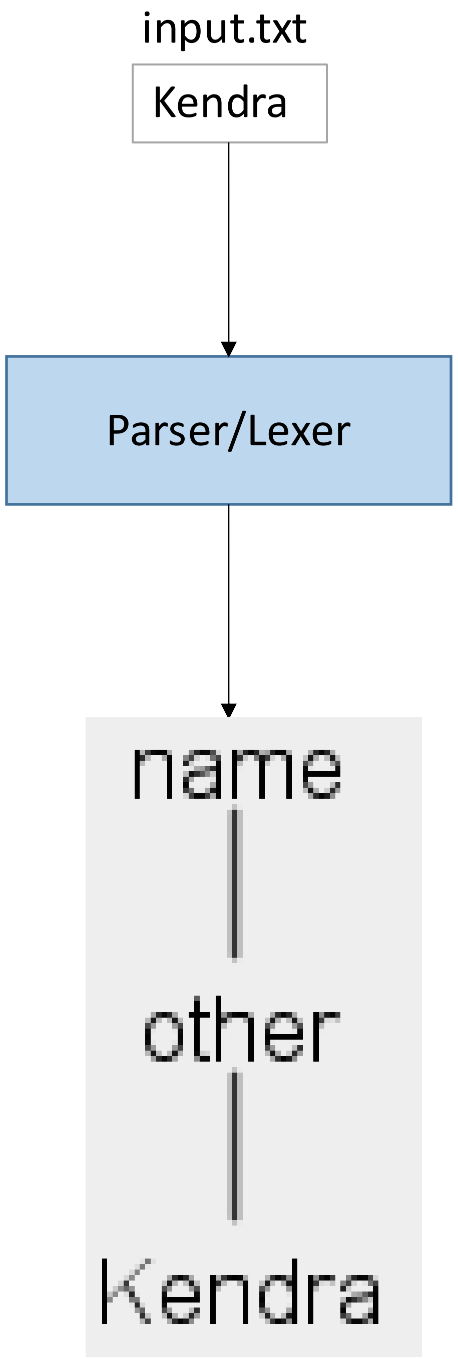


This part of the input matches this rule
Kendra
lexer grammar MyLexer;
KEN : 'Ken' ;
OTHER : [a-zA-Z]+ ;
WS : [\t\r\n]+ -> skip ;
This part of the input matches this rule

Will the lexer split Kendra into two tokens?



This is what the lexer does:



Lesson Learned

Given two (or more) token rules that match the input, the lexer uses the token rule that matches the <u>longest</u> part of the input.

Modified lexer grammar

```
lexer grammar MyLexer;

KEN    : 'Ken' ;
OTHER  : [a-zA-Z][a-zA-Z][a-zA-Z] ;
WS     : [ \t\r\n]+ -> skip ;
```

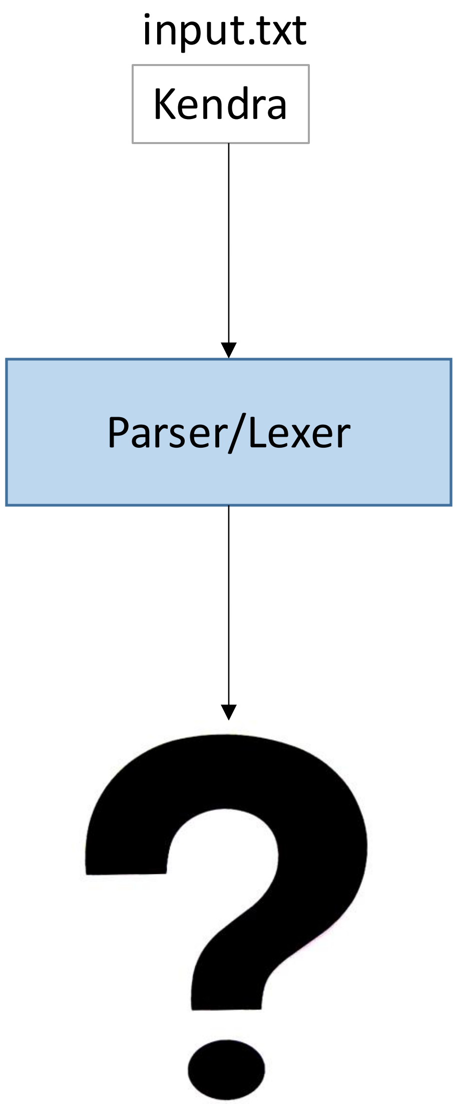
Match strings of length 3

```
lexer grammar MyLexer;

KEN     : 'Ken' ;
OTHER   : [a-zA-Z][a-zA-Z][a-zA-Z] ;
WS      : [ \t\r\n]+ -> skip ;
```

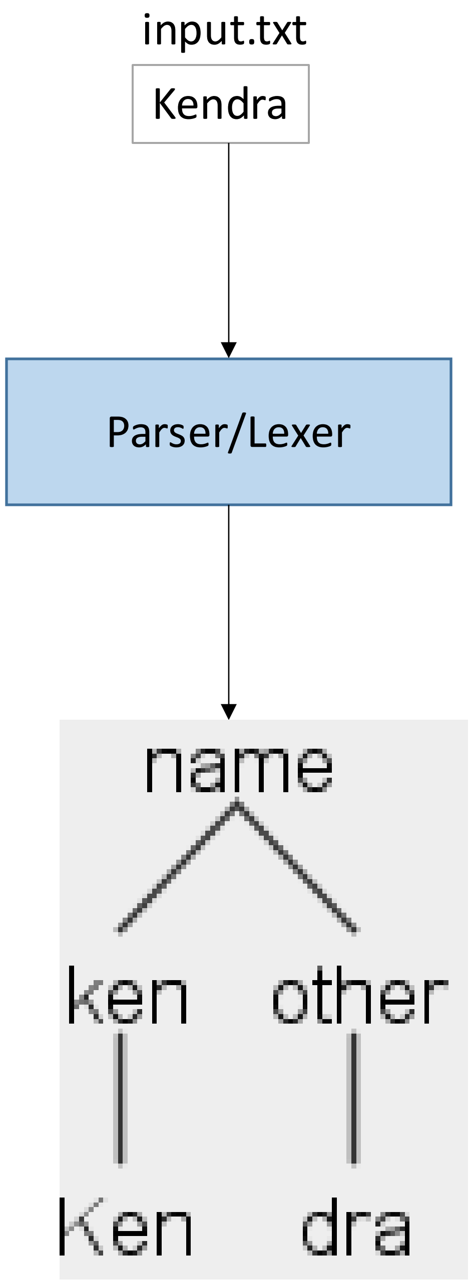
Same parser grammar

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

name  : (ken | other)+ ;

ken   : KEN ;

other : OTHER ;
```

See example12


input.txt
Kendra
Parser/Lexer
?

This is what the lexer does:



Lesson Learned

Given two token rules that match the input *and each match the same length string*, the lexer uses the token rule that is listed *first*.

"not" operator

Tilda (~) means "not"

```
lexer grammar MyLexer;

COMMA  : ',' ;
NL     : ('\r')?'\n' ;
WS     : [ \t\r\n]+ -> skip ;
STRING : (~[,\r\n])+ ;
```

Tilda (~) means "not". This lexer rule says: a STRING is one or more characters. Each character in STRING is not a comma, carriage return (\r), or newline (\n).

Equivalent

STRING : (~'*')+ ;

STRING is one or more characters. Each character in STRING is not an asterisk.

STRING : (~[*])+ ;

Can't negate multi-character literals

Suppose we want to define this rule: a STRING is one or more characters but it can't contain this pair of symbols: */

```
STRING : (~'*/')+ ;
```

ANTLR throws this error:

multi-character literals are not allowed in lexer sets: '*/'

Will this work?

```
STRING : (~[*/])+ ;
```

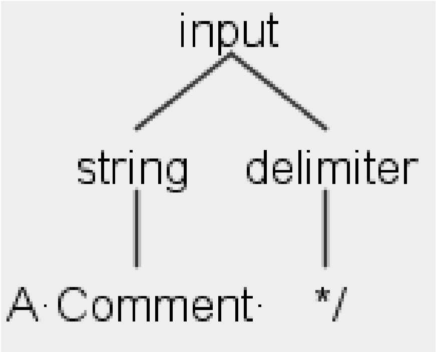
```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

input : (string | delimiter)* ;

string : STRING ;

delimiter : DELIMITER ;
```

```
lexer grammar MyLexer;

STRING     : (~[*/])+ ;

DELIMETER  : '/*' ;
```

input.txt

A comment */

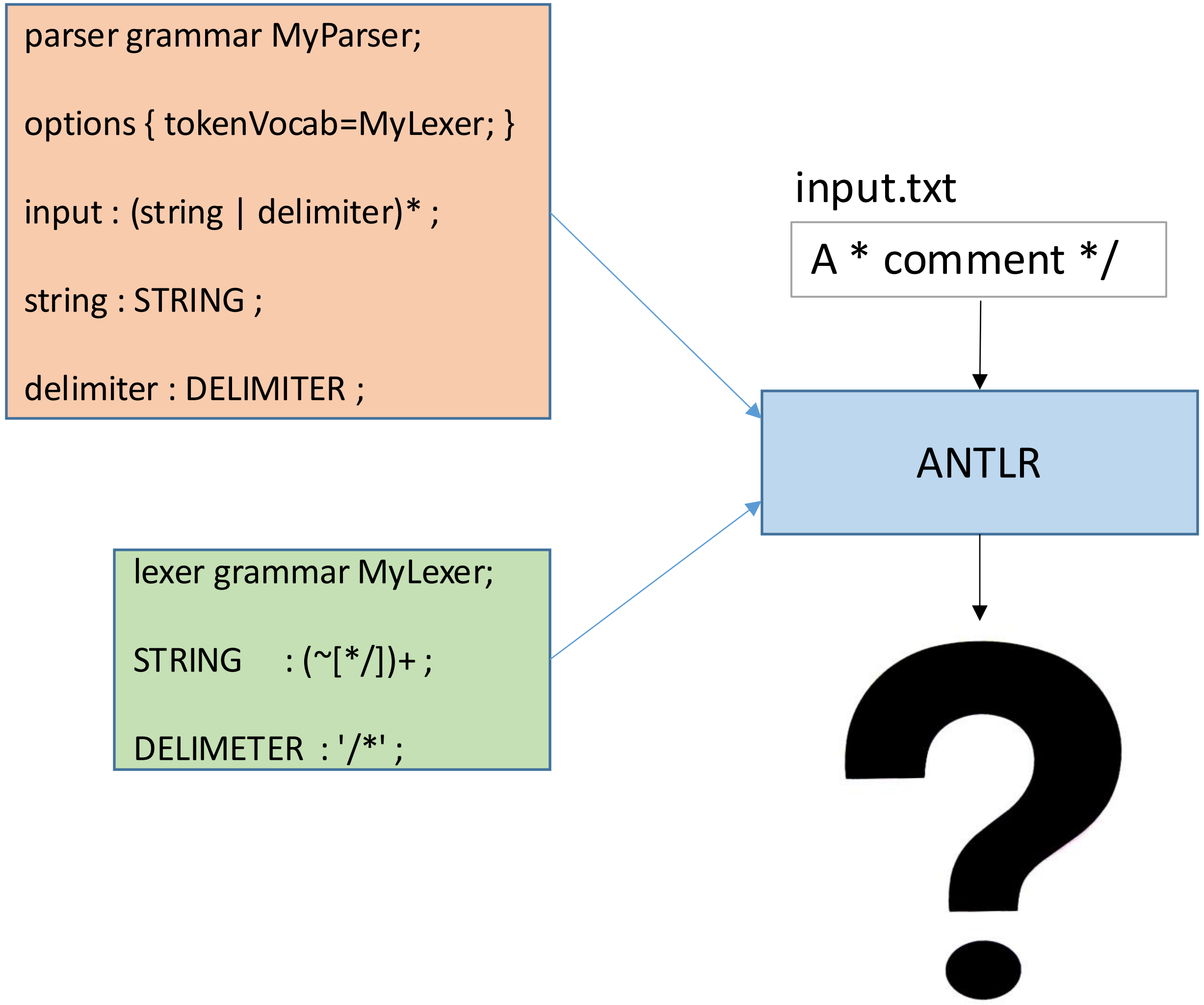
ANTLR

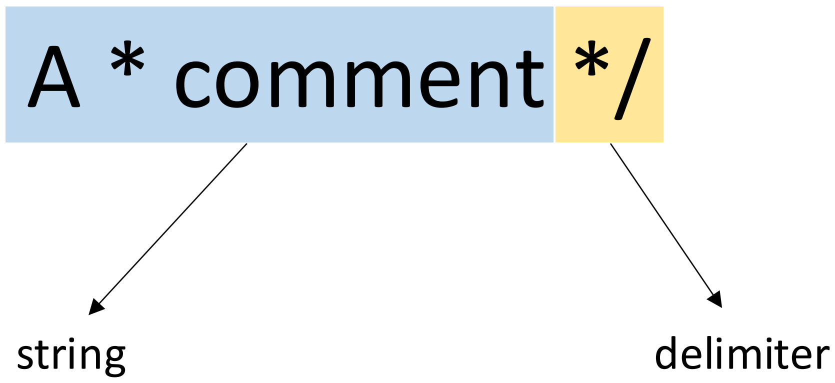


See example12.1

It works!
More accurately, it produces the correct parse tree for this input.
But will it allow STRING to contain an individual character * or /?

Do Lab7


parser grammar MyParser;
options { tokenVocab=MyLexer; }
input : (string | delimiter)* ;
string : STRING ;
delimiter : DELIMITER ;
lexer grammar MyLexer;
STRING : (~[*/])+ ;
DELIMETER : '/*' ;
input.txt
A * comment */
ANTLR
?


A * comment
*/
string
delimiter

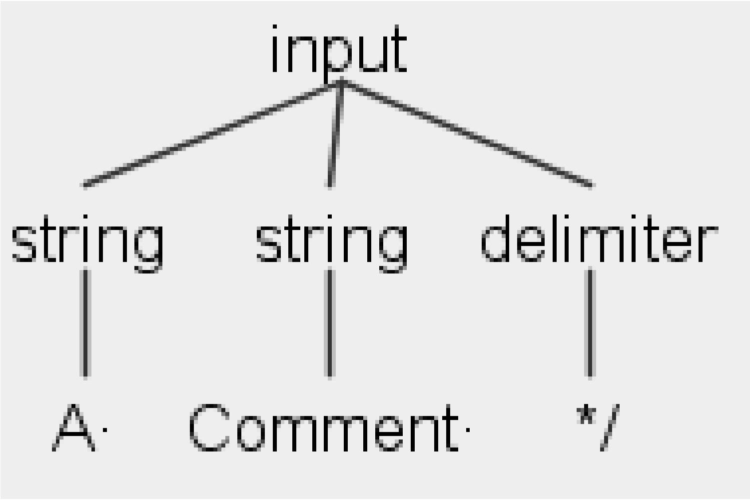
```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

input : (string | delimiter)* ;

string : STRING ;

delimiter : DELIMITER ;
```

```
lexer grammar MyLexer;

STRING    : (~[*/])+ ;

DELIMETER  : '/*' ;
```

input.txt

```
A * comment */
```

ANTLR



WRONG

See example12.2

token recognition error at: '* '

Here's the problem

```
lexer grammar MyLexer;

STRING      : (~[*/])+ ;

DELIMITER   : '*/' ;
```

This rule is not saying this: There must not be an asterisk followed by a forward slash. Rather, the rule is saying this: There must not be an asterisk. There must not be a forward slash.

Consequently, our parser incorrectly parses perfectly fine input.

No solution

The ANTLR grammar is unable to express this rule: A STRING contains one or more characters, except for the pair */

fragment

Fragment

- The lexer uses your lexer rules to tokenize the input.
- Sometimes you want to create a lexer rule, but you don't want the lexer to use it to tokenize the input. Rather, it is simply to be used to help write another lexer rule. To indicate that it is simply a helper, precede it with **fragment**.

```
NAME   :  ALPHA (ALPHA|DIGIT)* ;

fragment ALPHA   :  [a-zA-Z] ;

fragment DIGIT    :  [0-9] ;
```

This rule is used to tokenize the input; it uses two helper rules.

helper rules

Create a lexer rule for a name

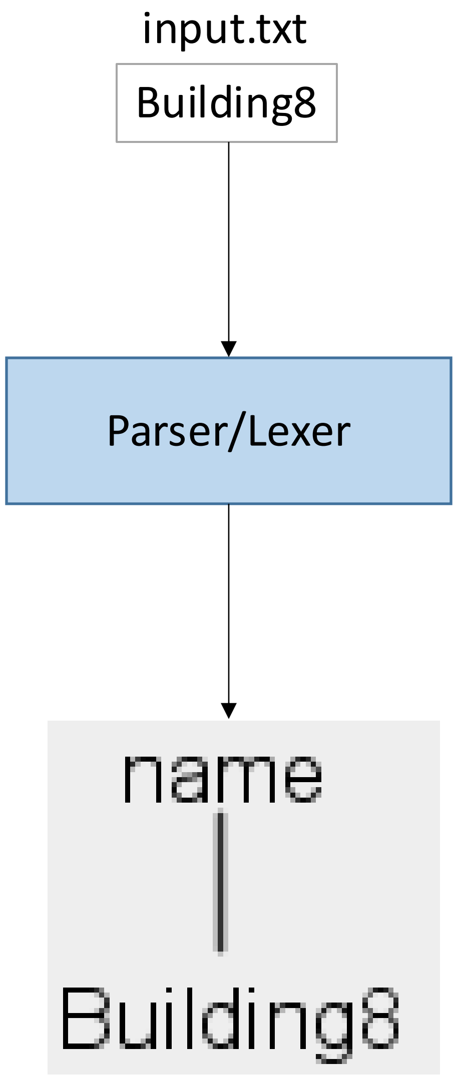
- The input contains a name.
- A name must begin with a letter. After that there may be zero or more letters and/or digits.
- Here is a legal name: `Building8`

The lexer grammar

```
lexer grammar MyLexer;

NAME  : ALPHA (ALPHA | DIGIT)* ;
WS    : [ \t\r\n]+ -> skip ;

fragment ALPHA : [a-zA-Z] ;
fragment DIGIT : [0-9] ;
```

The parser grammar

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

name  : NAME ;
```



See example13

Expressing Lexer Patterns using ANTLR notation versus Regex notation

Regex vs ANTLR notation

```
lexer grammar MyLexer;

ID    : [a-zA-Z]+ ;
INT   : [0-9]+ ;
NL    : '\r'? '\n' ;
```

Regex notation

ANTLR notation

Whitespace

Here are two equivalent ways to instruct ANTLR to discard whitespace (spaces, tabs, carriage return, and newlines):

```
WS  : ('  ' | '\t' | '\r' | '\n')+  ->  skip ;
```
// ANTLR notation

```
WS  : [ \t\r\n]+  ->  skip ;
```
// Regex notation

ID tokens

Here are two equivalent ways to instruct ANTLR to create ID tokens:

```
ID  : ('a' .. 'z' | 'A' .. 'Z')+   ;      // ANTLR notation

ID  : [a-zA-Z]+  ;                        // Regex notation
```

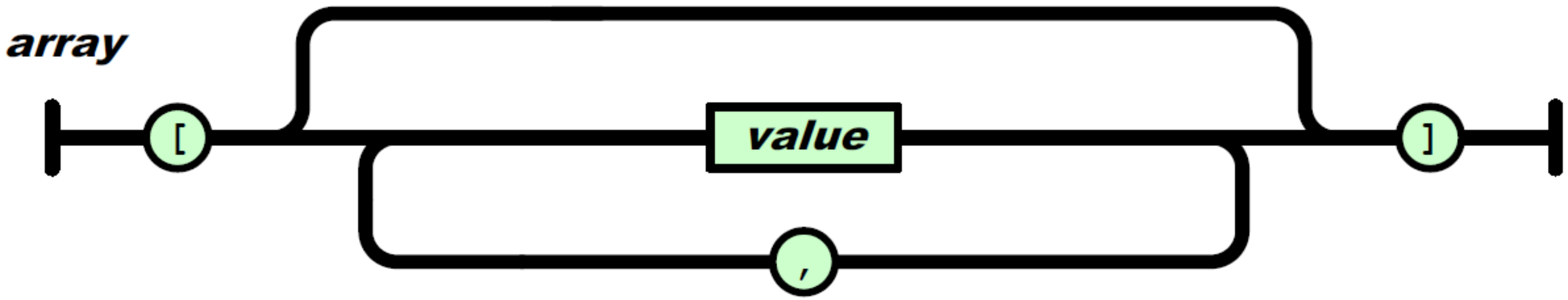
Range operator

```
ID  : ('a' .. 'z' | 'A' .. 'Z')+   ;
```

The .. is called the range operator.

Recursive <u>Lexer</u> Rules

Define a parser for JSON arrays



Let's assume that a "value" can be either an integer or an array.
Note the recursion: an array can contain an array, which can contain an array, and so forth.

Examples of JSON arrays

```
[ ]
[1]
[1, 2]
[1, 2, [3, 4]]
[1, 2, [[3, 4], 5]]
```

An input is an ARRAY if:

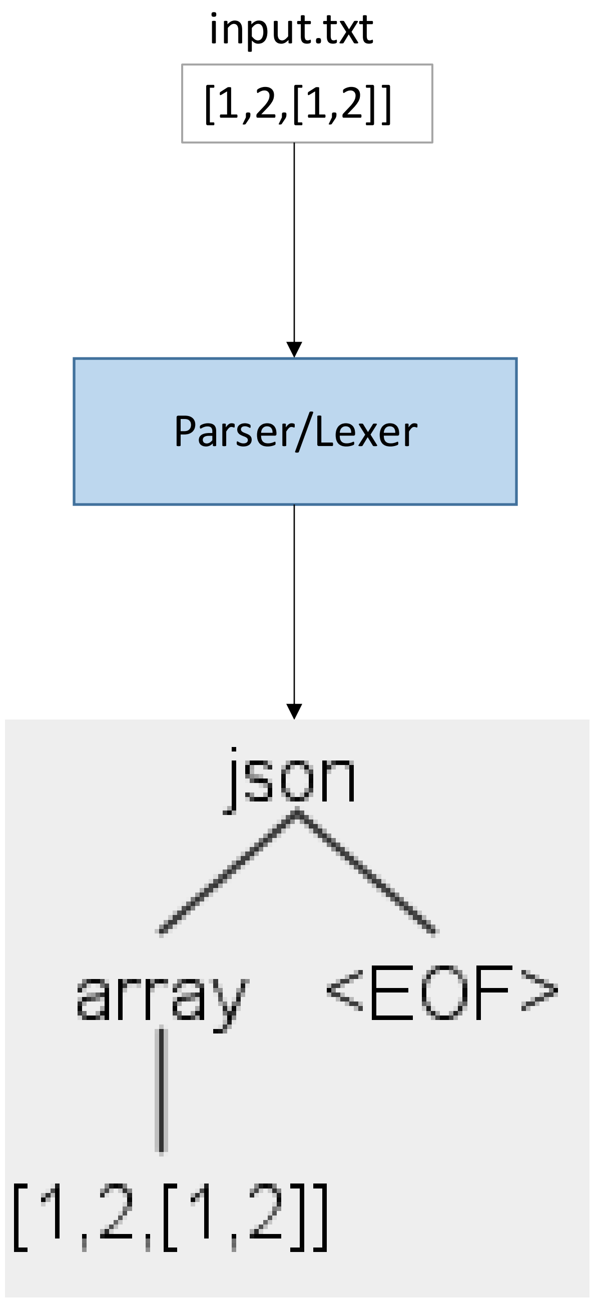
- It contains '[' followed by ']' (an empty array)
- It contains '[' followed by an ARRAY or an INT, followed by zero or more ',' then ARRAY or INT, followed by ']'

The lexer grammar

```
lexer grammar MyLexer;

ARRAY : '[' ']'
        |
        '[' (ARRAY | INT) (',' (ARRAY | INT))* ']'
        ;

WS    : [ \t\r\n]+ -> skip ;

fragment INT : [0-9]+ ;
```

The parser grammar

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

json    : array EOF ;

array   : ARRAY ;
```



See example14

Do Lab8

Recursive Parser Rules

The parser grammar

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

json    : array EOF ;

array   : LB RB
          |
          LB (array | INT) (COMMA (array | INT))* RB
          ;
```

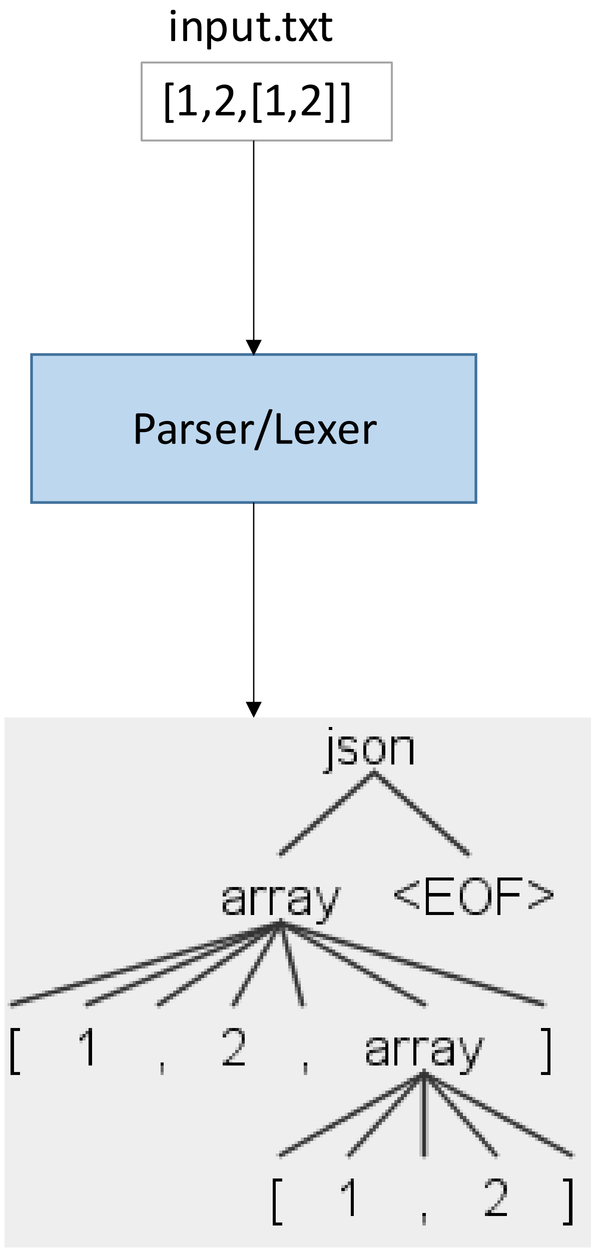
LB = Left Bracket
RB = Right Bracket

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

json    : array EOF ;

array   : LB RB
        |
        LB (array | INT) (COMMA (array | INT))* RB
        ;
```

recurse

The lexer grammar

```
lexer grammar MyLexer;

LB    : '[' ;

RB    : ']' ;

INT   : [0-9]+ ;

COMMA : ',' ;

WS    : [ \t\r\n]+ -> skip ;
```



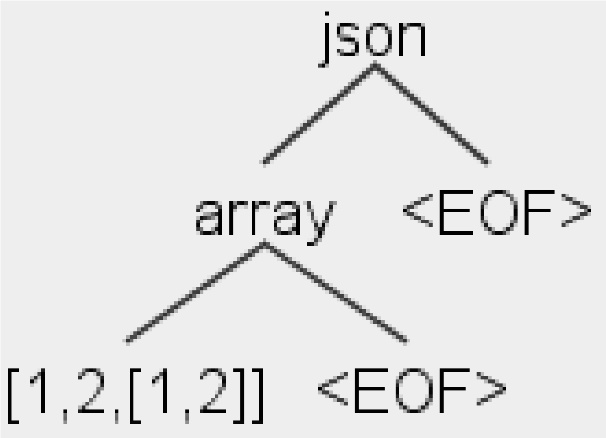
See example15

Drawing the line between the lexer and the parser

Define ARRAY in lexer or parser?

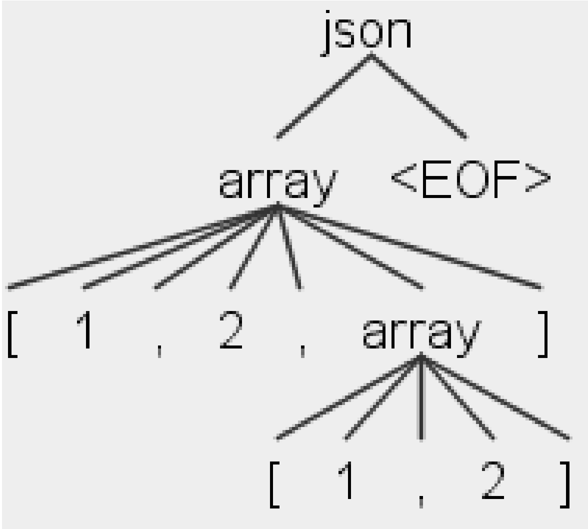
- Two sections back we saw a <u>lexer</u> rule for defining a JSON ARRAY.
- The last section showed a <u>parser</u> rule for defining a JSON ARRAY.
- Should a JSON ARRAY be defined in the lexer or the parser?

Lexer sends ARRAY to the parser



Lexer defines ARRAY

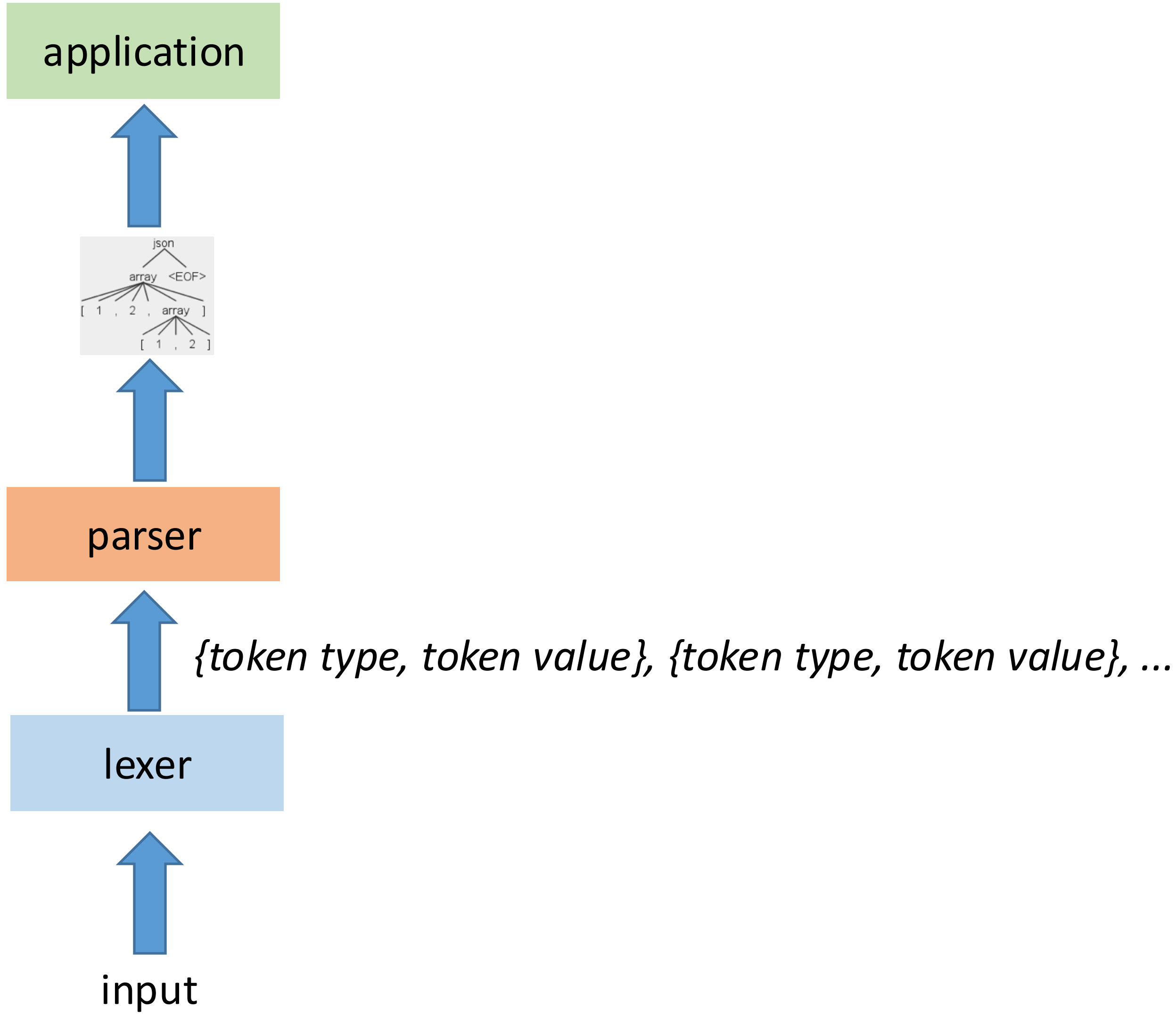
Lexer sends to the parser each part of the array



Parser defines array

Should the lexer define ARRAY or should the parser?

Does the application need to access each part of the array or does the application need just the entire array?



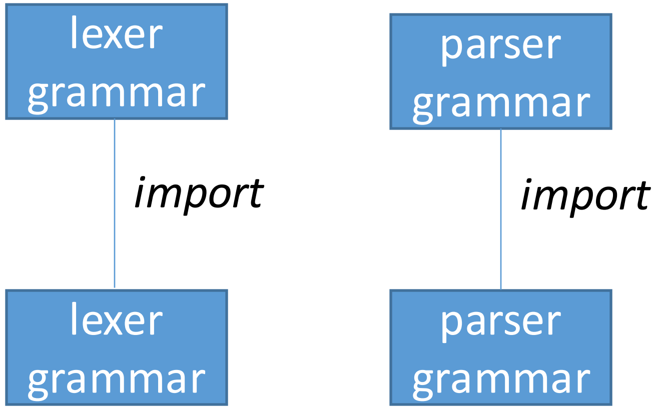
Define in lexer or parser?

Answer: define it in the lexer if the application wants to treat the JSON array as a single, monolithic thing. Define it in the parser if the application wants to process each individual part.

Managing large grammars

Import

- It's a good idea to break up large grammars into logical chunks, just like we do with software.
- A lexer grammar can import other lexer grammars. A parser grammar can import other parser grammars.



Laws of grammar imports

- Suppose grammar A imports grammar B. If there are conflicting rules, then the rules in A override the rules in B. That is, rules from B are not included if the rules are already defined in A.
- The result of processing the imports is a single combined grammar: the ANTLR code generator sees a complete grammar and has no idea there were imported grammars.
- Rules from imported grammars go to the end.

Example

```
lexer grammar MyLexer_imported;

KEYWORDS : 'if' | 'then' | 'begin' ;
```

import

```
lexer grammar MyLexer;

import MyLexer_imported ;

ID   : [a-zA-Z]+ ;
WS   : [ \t\r\n]+ -> skip ;
```

```
lexer grammar MyLexer;

ID   : [a-zA-Z]+ ;
WS   : [ \t\r\n]+ -> skip ;
KEYWORDS : 'if' | 'then' | 'begin' ;
```

Imported rule is unreachable

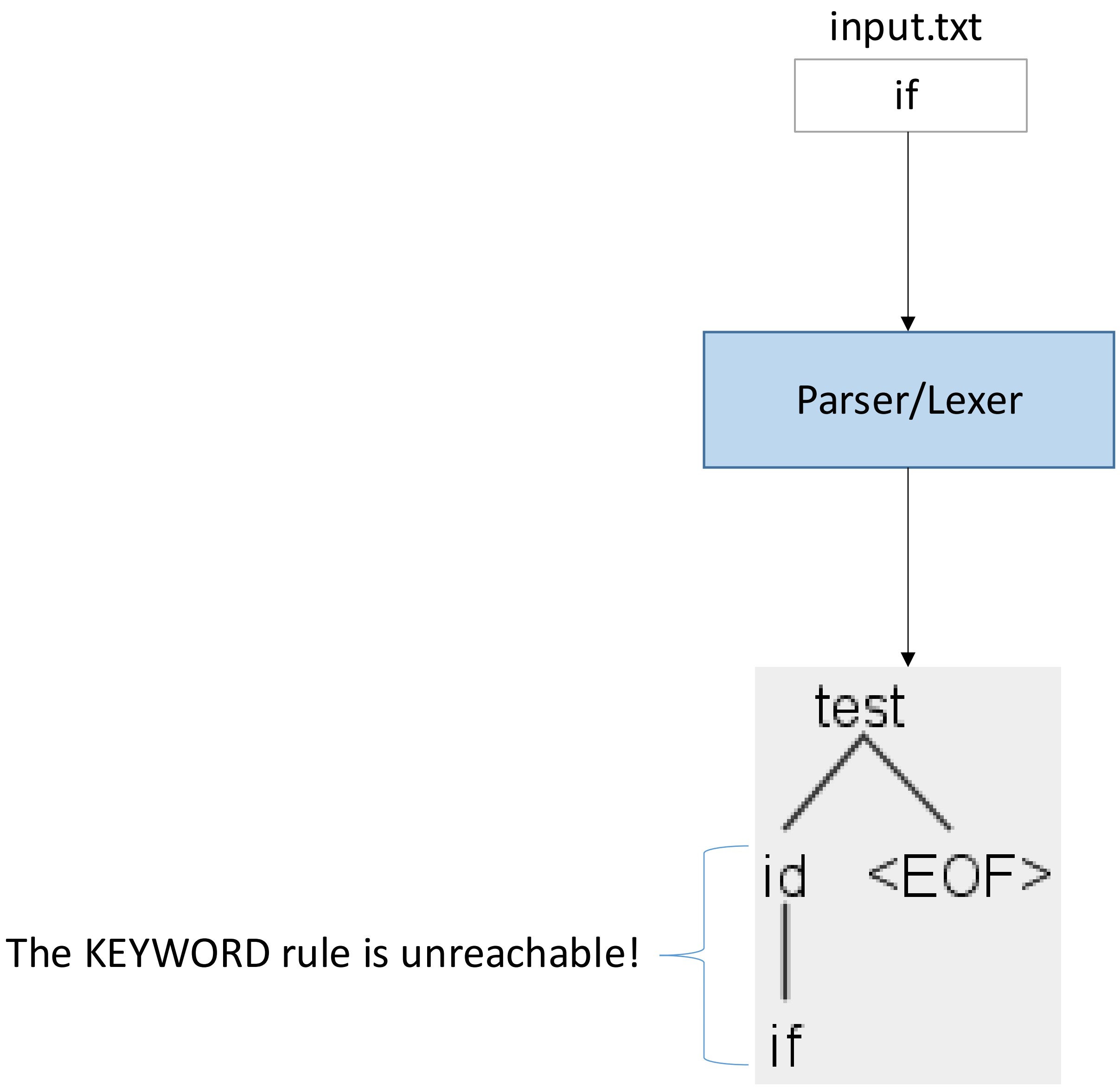
No input will ever match this token rule!

```
lexer grammar MyLexer;

ID   : [a-zA-Z]+ ;
WS   : [ \t\r\n]+ -> skip ;
KEYWORDS : 'if' | 'then' | 'begin' ;
```

First rule captures the input

'if' and 'then' and 'begin' will be matched by this token rule since it is positioned first

```
lexer grammar MyLexer;

ID   : [a-zA-Z]+ ;
WS   : [ \t\r\n]+ -> skip ;
KEYWORDS : 'if' | 'then' | 'begin' ;
```



See example16

Lexers and parser

```
lexer grammar MyLexer_imported;

KEYWORDS : 'if' | 'then' | 'begin' ;
```

import

```
lexer grammar MyLexer;

import MyLexer_imported ;

ID  : [a-zA-Z]+ ;
WS  : [ \t\r\n]+ -> skip ;
```

use

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

test : (id | keyword) EOF ;

id    : ID ;

keyword : KEYWORDS ;
```

run.bat

Run ANTLR on **MyLexer_imported.g4** before running ANTLR on **MyLexer.g4**

```
set CLASSPATH=.;../../antlr-jar/antlr-complete.jar;%CLASSPATH%

echo Running ANTLR on the lexer: MyLexer_imported.g4
java org.antlr.v4.Tool MyLexer_imported.g4 -no-listener -no-visitor

echo Running ANTLR on the lexer: MyLexer.g4
java org.antlr.v4.Tool MyLexer.g4 -no-listener -no-visitor

echo Running ANTLR on the parser: MyParser.g4
java org.antlr.v4.Tool MyParser.g4 -no-listener -no-visitor

echo Compiling the Java code that ANTLR generator (the lexer and parser code)
javac *.java

echo Running the test rig on the generated parser, using as input the string in: input.txt
echo And generating a GUI output (i.e., a parse tree graphic)
java org.antlr.v4.gui.TestRig My test -gui < input.txt
```

Do Lab9

Modes

Input contains two different formats

The input data consists of comma-separated-values (CSV) as we saw earlier. After the CSV is a separator (4 dashes). After that is a series of key-value pairs. Below is sample input.

```
FirstName, LastName, Street, City, State, ZipCode
Mark, McWhorter, 4460 Stuart Street, Marion Center, PA, 15759
Monica, Apodaca, 258 Oliver Street, Fort Worth, TX, 76118
Kenisha, Le, 1579 Olen Thomas Drive, Charlie, TX, 76377
Dustin, Edwards, 2249 Ingram Road, Greensboro, NC, 27401
Ruth, Heath, 988 Davisson Street, Fairmont, IN, 46928
Christin, Dehart, 515 Ash Avenue, Saint Louis, MO, 63108
----
FirstName=John
LastName=Smith
FirstName=Sally
LastName=Johnson
```

Need different lexer rules

```
COMMA  : ',' ;
NL     : ('\r')?'\n' ;
WS     : [ \t\r\n]+ -> skip ;
STRING : (~[,\r\n])+ ;
```

tokenize this part of the input using these lexer rules

```
KEY    : ('FirstName' | 'LastName') ;
EQ     : '=' ;
NL     : ('\r')?'\n' ;
WS     : [ \t\r\n]+ -> skip ;
VALUE  : (~[=\r\n])+ ;
```

tokenize this part of the input using these lexer rules

FirstName, LastName, Street, City, State, ZipCode
Mark, McWhorter, 4460 Stuart Street, Marion Center, PA, 15759
Monica, Apodaca, 258 Oliver Street, Fort Worth, TX, 76118
Kenisha, Le, 1579 Olen Thomas Drive, Charlie, TX, 76377
Dustin, Edwards, 2249 Ingram Road, Greensboro, NC, 27401
Ruth, Heath, 988 Davisson Street, Fairmont, IN, 46928
Christin, Dehart, 515 Ash Avenue, Saint Louis, MO, 63108
\----
FirstName=John
LastName=Smith
FirstName=Sally
LastName=Johnson

Switch to other lexer rules at separator

```
COMMA  : ',' ;
NL     : ('\r')?'\n' ;
WS     : [ \t\r\n]+ -> skip ;
STRING : (~[,\r\n])+ ;
```

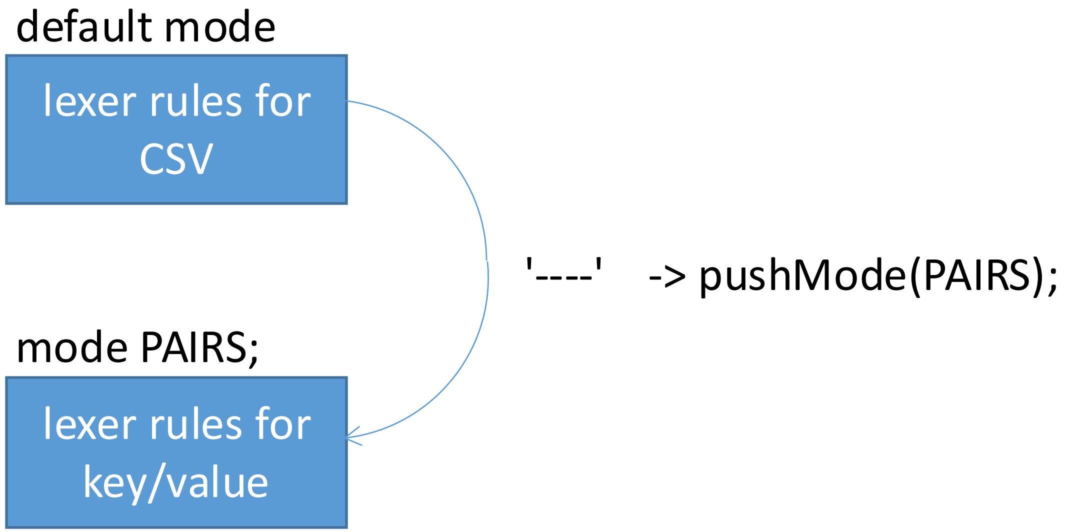
```
KEY      : ('FirstName' | 'LastName') ;
EQ       : '=' ;
NL       : ('\r')?'\n' ;
WS       : [ \t\r\n]+ -> skip ;
VALUE    : (~[=\r\n])+ ;
```

When we get to this separator, switch from using the top lexer rules to using the bottom lexer rules

```
FirstName, LastName, Street, City, State, ZipCode
Mark, McWhorter, 4460 Stuart Street, Marion Center, PA, 15759
Monica, Apodaca, 258 Oliver Street, Fort Worth, TX, 76118
Kenisha, Le, 1579 Olen Thomas Drive, Charlie, TX, 76377
Dustin, Edwards, 2249 Ingram Road, Greensboro, NC, 27401
Ruth, Heath, 988 Davisson Street, Fairmont, IN, 46928
Christin, Dehart, 515 Ash Avenue, Saint Louis, MO, 63108
----
FirstName=John
LastName=Smith
FirstName=Sally
LastName=Johnson
```

Lexical modes

ANTLR provides "mode" for switching between one set of lexer rules to another set of lexer rules:



Multiple sublexers

With modes its like having multiple sublexers, all in one file.

The lexer grammar

```
lexer grammar MyLexer;

COMMA   : ',' ;
NL      : ('\r')?'\n' ;
WS      : [ \t\r\n]+ -> skip ;
SEPARATOR : SEP -> skip, pushMode(PAIRS) ;
STRING : (~[,\r\n])+ ;

fragment SEP : '----' NL ;

mode PAIRS ;
KEY        : ('FirstName' | 'LastName') ;
EQ         : '=' ;
NL2        : ('\r')?'\n' ;
WS2        : [ \t\r\n]+ -> skip ;
VALUE      : (~[=\r\n])+ ;
```

This says: Any token rules after this are part of the PAIRS mode.

```
lexer grammar MyLexer;

COMMA   : ',' ;
NL      : ('\r')?'\n' ;
WS      : [ \t\r\n]+ -> skip ;
SEPARATOR : SEP -> skip, pushMode(PAIRS) ;
STRING : (~[,\r\n])+ ;

fragment SEP : '----' NL ;

mode PAIRS ;
KEY         : ('FirstName' | 'LastName') ;
EQ          : '=' ;
NL2         : ('\r')?'\n' ;
WS2         : [ \t\r\n]+ -> skip ;
VALUE       : (~[=\r\n])+ ;
```

```
lexer grammar MyLexer;

COMMA   : ',' ;
NL      : ('\r')?'\n' ;
WS      : [ \t\r\n]+ -> skip ;
SEPARATOR : SEP -> skip, pushMode(PAIRS) ;
STRING : (~[,\r\n])+ ;

fragment SEP : '----' NL ;

mode PAIRS ;
KEY         : ('FirstName' | 'LastName') ;
EQ          : '=' ;
NL2         : ('\r')?'\n' ;
WS2         : [ \t\r\n]+ -> skip ;
VALUE       : (~[=\r\n])+ ;
```

lexer rules for the default mode.

lexer rules for the PAIRS mode.

Upon encountering the separator in the input, stop using the rules in the default mode and start using the rules in the PAIRS mode.

```
lexer grammar MyLexer;

COMMA   : ',' ;
NL      : ('\r')?'\n' ;
WS      : [ \t\r\n]+ -> skip ;
SEPARATOR : SEP -> skip, pushMode(PAIRS) ;
STRING : (~[,\r\n])+ ;

fragment SEP : '----' NL ;

mode PAIRS ;
KEY         : ('FirstName' | 'LastName') ;
EQ          : '=' ;
NL2         : ('\r')?'\n' ;
WS2         : [ \t\r\n]+ -> skip ;
VALUE       : (~[=\r\n])+ ;
```

Discard the separator *and* push the current (default) mode on the stack and begin using the rules in the PAIRS mode.

```
lexer grammar MyLexer;

COMMA   : ',' ;
NL      : ('\r')?'\n' ;
WS      : [ \t\r\n]+ -> skip ;
SEPARATOR : SEP -> skip, pushMode(PAIRS) ;
STRING : (~[,\r\n])+ ;

fragment SEP : '----' NL ;

mode PAIRS ;
KEY         : ('FirstName' | 'LastName') ;
EQ          : '=' ;
NL2         : ('\r')?'\n' ;
WS2         : [ \t\r\n]+ -> skip ;
VALUE       : (~[=\r\n])+ ;
```

lexer commands

```
lexer grammar MyLexer;

COMMA   : ',' ;
NL      : ('\r')?'\n' ;
WS      : [ \t\r\n]+ -> skip ;
SEPARATOR : SEP -> skip, pushMode(PAIRS) ;
STRING : (~[,\r\n])+ ;

fragment SEP : '----' NL ;

mode PAIRS ;
KEY         : ('FirstName' | 'LastName') ;
EQ          : '=' ;
NL2         : ('\r')?'\n' ;
WS2         : [ \t\r\n]+ -> skip ;
VALUE       : (~[=\r\n])+ ;
```

Both modes need a rule for whitespace. *You cannot have two rules with the same name*. So, for the default mode I named it WS and for the PAIRS mode I named it WS2.

```
lexer grammar MyLexer;

COMMA     : ',' ;
NL        : ('\r')?'\n' ;
WS        : [ \t\r\n]+ -> skip ;
SEPARATOR : SEP -> skip, pushMode(PAIRS) ;
STRING : (~[,\r\n])+ ;

fragment SEP : '----' NL ;

mode PAIRS ;
KEY         : ('FirstName' | 'LastName') ;
EQ          : '=' ;
NL2         : ('\r')?'\n' ;
WS2         : [ \t\r\n]+ -> skip ;
VALUE       : (~[=\r\n])+ ;
```

Same situation with newlines.

```
lexer grammar MyLexer;

COMMA   : ',' ;
NL      : ('\r')?'\n' ;
WS      : [ \t\r\n]+ -> skip ;
SEPARATOR : SEP -> skip, pushMode(PAIRS) ;
STRING : (~[,\r\n])+ ;

fragment SEP : '----' NL ;

mode PAIRS ;
KEY         : ('FirstName' | 'LastName') ;
EQ          : '=' ;
NL2         : ('\r')?'\n' ;
WS2         : [ \t\r\n]+ -> skip ;
VALUE       : (~[=\r\n])+ ;
```

The parser grammar

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

document  : header rows pairs EOF ;
header    : field (COMMA field)* NL ;
rows      : (row)* ;
row       : field (COMMA field)* NL ;
field     : STRING | ;

pairs     : (pair)* ;
pair      : key EQ value NL2;
key       : KEY ;
value     : VALUE ;
```

parser rules for structuring tokens from the CSV portion

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

document  : header rows pairs EOF ;
header    : field (COMMA field)* NL ;
rows      : (row)* ;
row       : field (COMMA field)* NL ;
field     : STRING | ;

pairs     : (pair)* ;
pair      : key EQ value NL2;
key       : KEY ;
value     : VALUE ;
```

parser rules for structuring tokens from the key/value portion

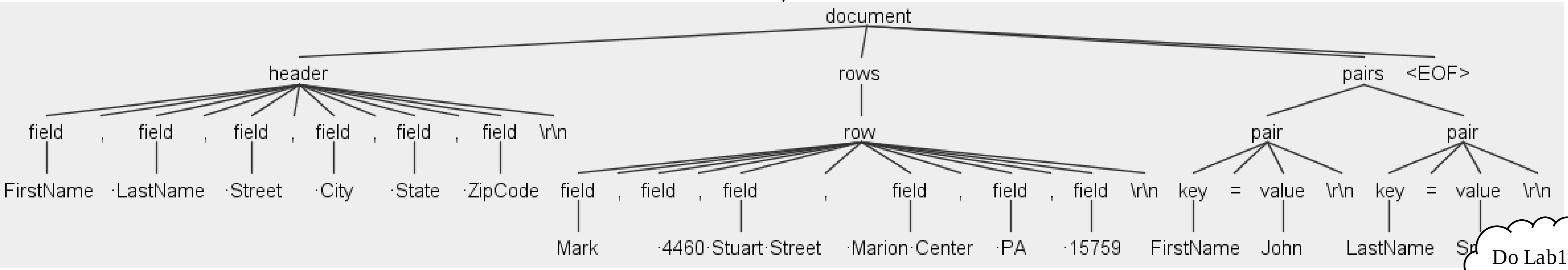
```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

document   : header rows pairs EOF ;
header     : field (COMMA field)* NL ;
rows       : (row)* ;
row        : field (COMMA field)* NL ;
field      : STRING | ;

pairs      : (pair)* ;
pair       : key EQ value NL2;
key        : KEY ;
value      : VALUE ;
```

input.txt

FirstName, LastName, Street, City, State, ZipCode
Mark,, 4460 Stuart Street, Marion Center, PA, 15759

FirstName=John
LastName=Smith

Parser/Lexer



Do Lab10

See example17

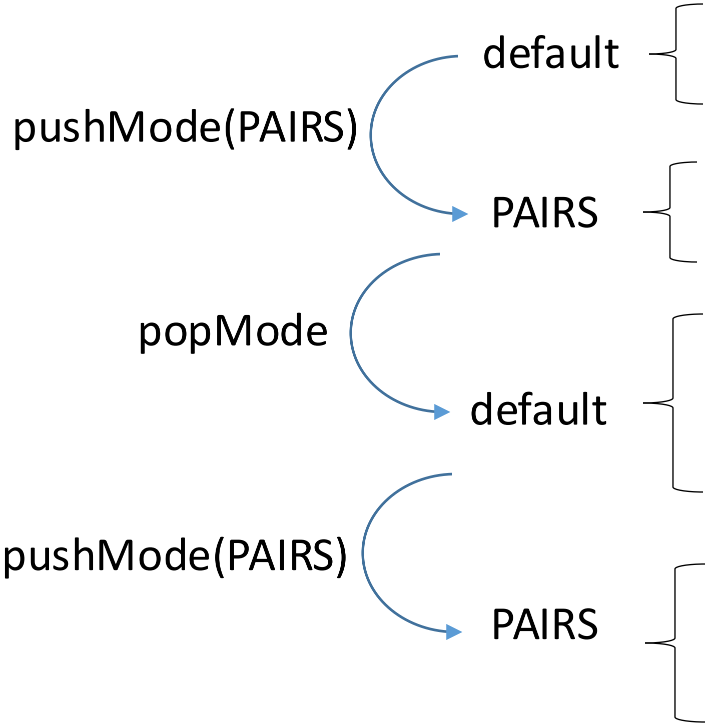
Alternating CSV and Key/Value sections

- The input in the last example had one CSV section followed by one Key/Value pair section (separated by a separator, of course).
- Let's extend it so there can be CSV then Key/Value then CSV then Key/Value then ... (each section separated by a separator, of course).

Sample input

```
FirstName, LastName, Street, City, State, ZipCode
Mark,, 4460 Stuart Street, Marion Center, PA, 15759
----
FirstName=John
LastName=Smith
----
FirstName, LastName, Street, City, State, ZipCode
Monica, Apodaca, 258 Oliver Street, Fort Worth, TX, 76118
Kenisha, Le, 1579 Olen Thomas Drive, Charlie, TX, 76377
Dustin, Edwards, 2249 Ingram Road, Greensboro, NC, 27401
----
FirstName=Harry
FirstName=Bob
FirstName=Lisa
```

Switch between modes



FirstName, LastName, Street, City, State, ZipCode
Mark,, 4460 Stuart Street, Marion Center, PA, 15759

FirstName=John
LastName=Smith

FirstName, LastName, Street, City, State, ZipCode
Monica, Apodaca, 258 Oliver Street, Fort Worth, TX, 76118
Kenisha, Le, 1579 Olen Thomas Drive, Charlie, TX, 76377
Dustin, Edwards, 2249 Ingram Road, Greensboro, NC, 27401

FirstName=Harry
FirstName=Bob
FirstName=Lisa

The lexer grammar

```
lexer grammar MyLexer;

COMMA  : ',' ;
NL     : ('\r')?'\n' ;
WS     : [ \t\r\n]+ -> skip ;
SEPARATOR : SEP -> skip, pushMode(PAIRS) ;
STRING : (~[,\r\n])+ ;

fragment SEP : '----' NL ;

mode PAIRS ;
KEY        : ('FirstName' | 'LastName') ;
EQ         : '=' ;
NL2        : ('\r')?'\n' ;
WS2        : [ \t\r\n]+ -> skip ;
SEPARATOR2 : SEP2 -> skip, popMode ;
VALUE      : (~[=\r\n])+ ;

fragment SEP2 : '----' NL2 ;
```

Start tokenizing the input using the token rules in mode PAIRS

```
lexer grammar MyLexer;

COMMA   : ',' ;
NL      : ('\r')?'\n' ;
WS      : [ \t\r\n]+ -> skip ;
SEPARATOR : SEP -> skip, pushMode(PAIRS) ;
STRING : (~[,\r\n])+ ;

fragment SEP : '----' NL ;

mode PAIRS ;
KEY        : ('FirstName' | 'LastName') ;
EQ         : '=' ;
NL2        : ('\r')?'\n' ;
WS2        : [ \t\r\n]+ -> skip ;
SEPARATOR2 : SEP2 -> skip, popMode ;
VALUE      : (~[=\r\n])+ ;

fragment SEP2 : '----' NL2 ;
```

```
lexer grammar MyLexer;

COMMA   : ',' ;
NL      : ('\r')?'\n' ;
WS      : [ \t\r\n]+ -> skip ;
SEPARATOR : SEP -> skip, pushMode(PAIRS) ;
STRING : (~[,\r\n])+ ;

fragment SEP : '----' NL ;

mode PAIRS ;
KEY         : ('FirstName' | 'LastName') ;
EQ          : '=' ;
NL2         : ('\r')?'\n' ;
WS2         : [ \t\r\n]+ -> skip ;
SEPARATOR2 : SEP2 -> skip, popMode ;
VALUE       : (~[=\r\n])+ ;

fragment SEP2 : '----' NL2 ;
```

Resume tokenizing the input using the token rules in the default mode

Discard separators

```
lexer grammar MyLexer;

COMMA   : ',' ;
NL      : ('\r')?'\n' ;
WS      : [ \t\r\n]+ -> skip ;
SEPARATOR : SEP -> skip, pushMode(PAIRS) ;
STRING : (~[,\r\n])+ ;

fragment SEP : '----' NL ;

mode PAIRS ;
KEY        : ('FirstName' | 'LastName') ;
EQ         : '=' ;
NL2        : ('\r')?'\n' ;
WS2        : [ \t\r\n]+ -> skip ;
SEPARATOR2 : SEP2 -> skip, popMode ;
VALUE      : (~[=\r\n])+ ;

fragment SEP2 : '----' NL2 ;
```

Lexer commands: pushMode, popMode, skip

pushMode(*name of a mode*)

popMode ← no arguments, no parentheses

skip ← no arguments, no parentheses

The parser grammar

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

document  : (header rows pairs)+ EOF ;
header    : field (COMMA field)* NL ;
rows      : (row)* ;
row       : field (COMMA field)* NL ;
field     : STRING | ;

pairs     : (pair)* ;
pair      : key EQ value NL2;
key       : KEY ;
value     : VALUE ;
```

CSV, Key/Value, CSV, Key/Value, ...

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

document    : (header rows pairs)+ EOF ;
header      : field (COMMA field)* NL ;
rows        : (row)* ;
row         : field (COMMA field)* NL ;
field       : STRING | ;

pairs       : (pair)* ;
pair        : key EQ value NL2;
key         : KEY ;
value       : VALUE ;
```

input.txt

FirstName, LastName, Street, City, State, ZipCode
Mark,, 4460 Stuart Street, Marion Center, PA, 15759

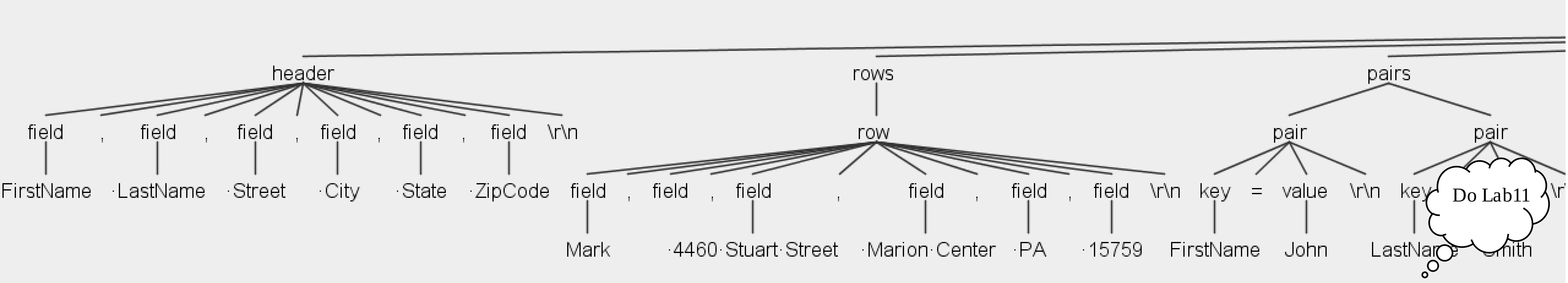
FirstName=John
LastName=Smith

FirstName, LastName, Street, City, State, ZipCode
Monica, Apodaca, 258 Oliver Street, Fort Worth, TX, 76118
Kenisha, Le, 1579 Olen Thomas Drive, Charlie, TX, 76377
Dustin, Edwards, 2249 Ingram Road, Greensboro, NC, 27401

FirstName=Harry
FirstName=Bob
FirstName=Lisa

Parser/Lexer

See example18



Context-sensitive problem #1

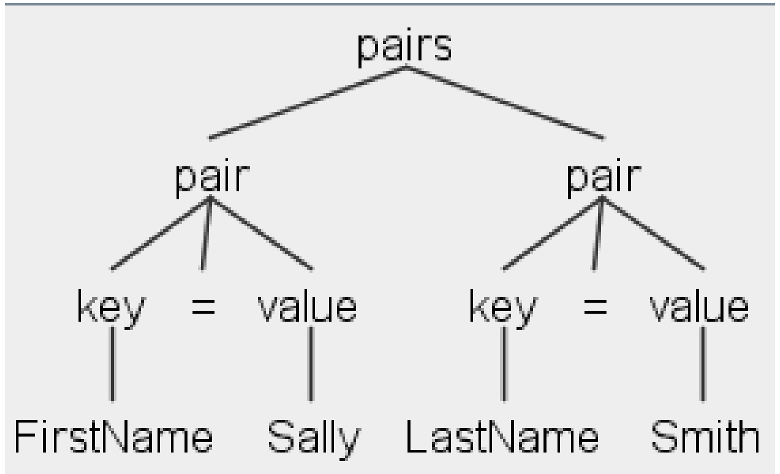
Lexer & parser for key/value pairs

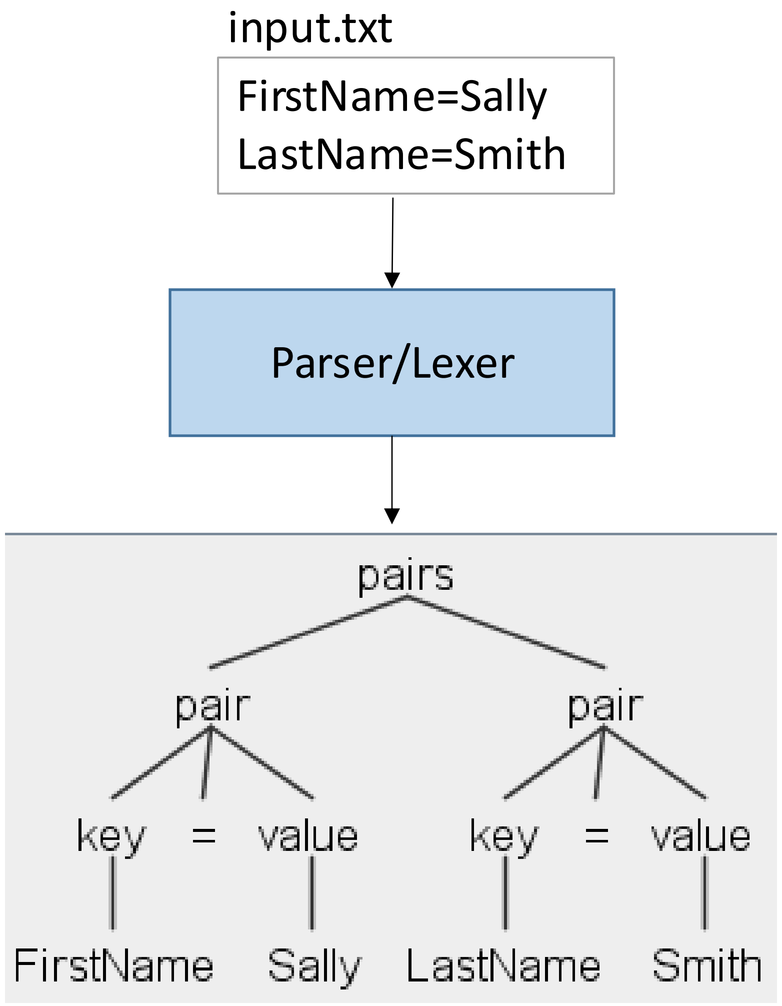
```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

pairs     : (pair)* ;
pair      : key EQ value ;
key       : KEY ;
value     : VALUE ;
```

```
lexer grammar MyLexer;

KEY       : ('FirstName' | 'LastName') ;
VALUE     : (~[=\r\n])+ ;
EQ        : '=' ;
WS        : [ \t\r\n]+ -> skip ;
```

input.txt

```
FirstName=Sally
LastName=Smith
```

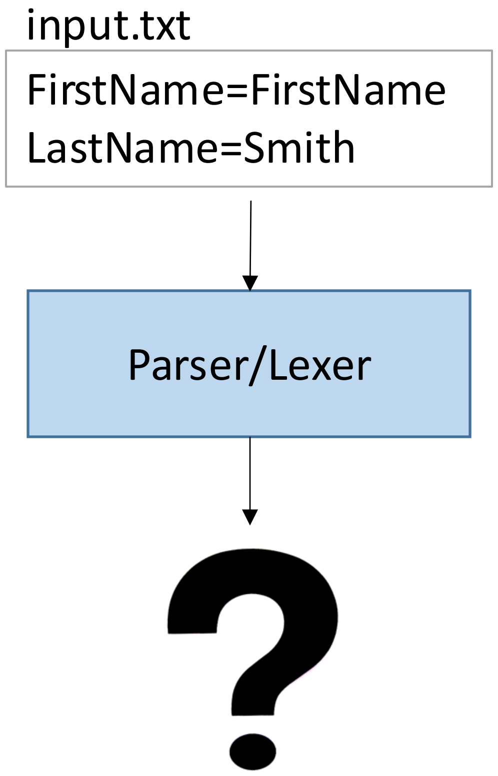




See example19

Person with an unusual first name

Suppose someone's first name is: FirstName. What will ANTLR do with that input?



1st token 2nd token 3rd token

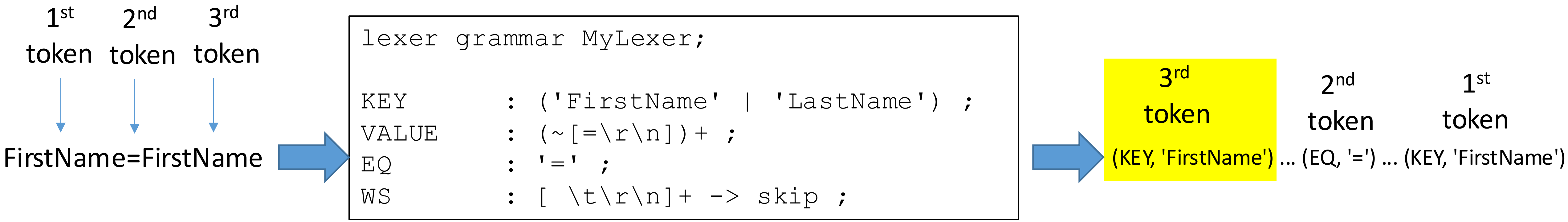
FirstName=FirstName

```
lexer grammar MyLexer;

KEY         : ('FirstName' | 'LastName') ;
VALUE       : (~[=\r\n])+ ;
EQ          : '=' ;
WS          : [ \t\r\n]+ -> skip ;
```

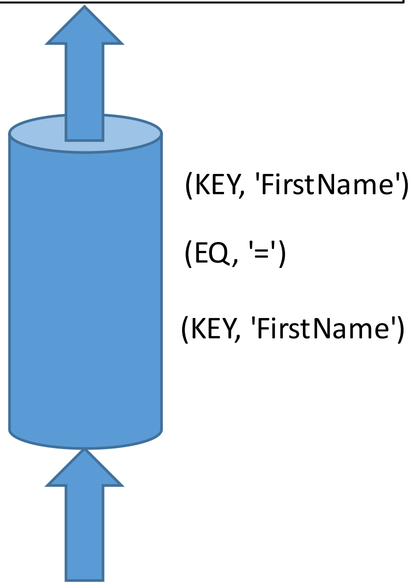
2nd token 1st token

... (EQ, '=') ... (KEY, 'FirstName')

What will be the 3rd token?


1st token
2nd token
3rd token
FirstName=FirstName
lexer grammar MyLexer;
KEY : ('FirstName' | 'LastName') ;
VALUE : (~[=\r\n])+ ;
EQ : '=' ;
WS : [\t\r\n]+ -> skip ;
3rd token
2nd token
1st token
(KEY, 'FirstName') ... (EQ, '=') ... (KEY, 'FirstName')

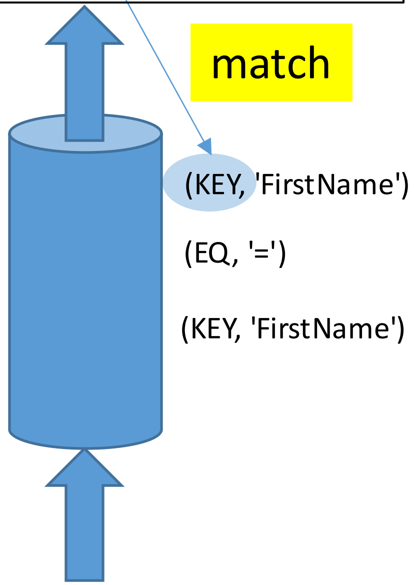
```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

pairs     : (pair)* ;
pair      : key EQ value ;
key       : KEY ;
value     : VALUE ;
```




```
lexer grammar MyLexer;

KEY       : ('FirstName' | 'LastName') ;
VALUE     : (~[=\r\n])+ ;
EQ        : '=' ;
WS        : [ \t\r\n]+ -> skip ;
```

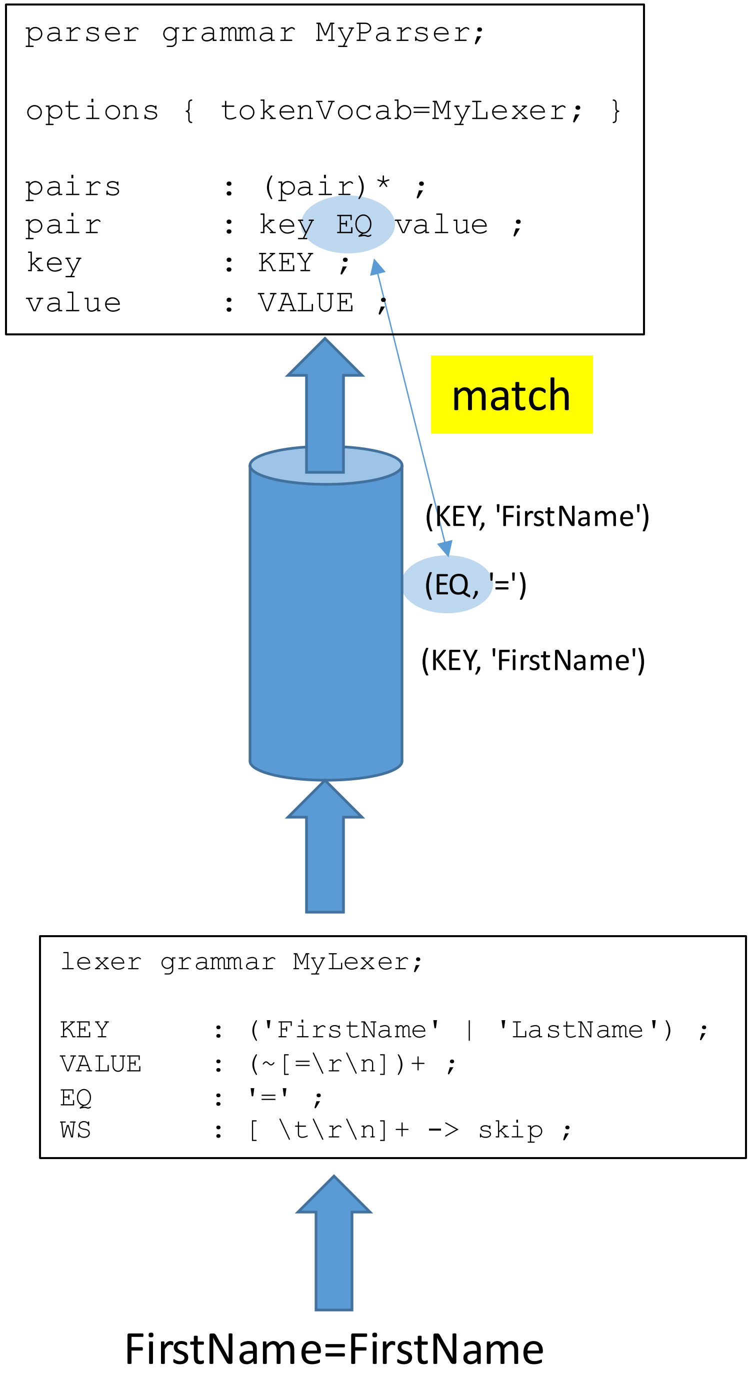
FirstName=FirstName

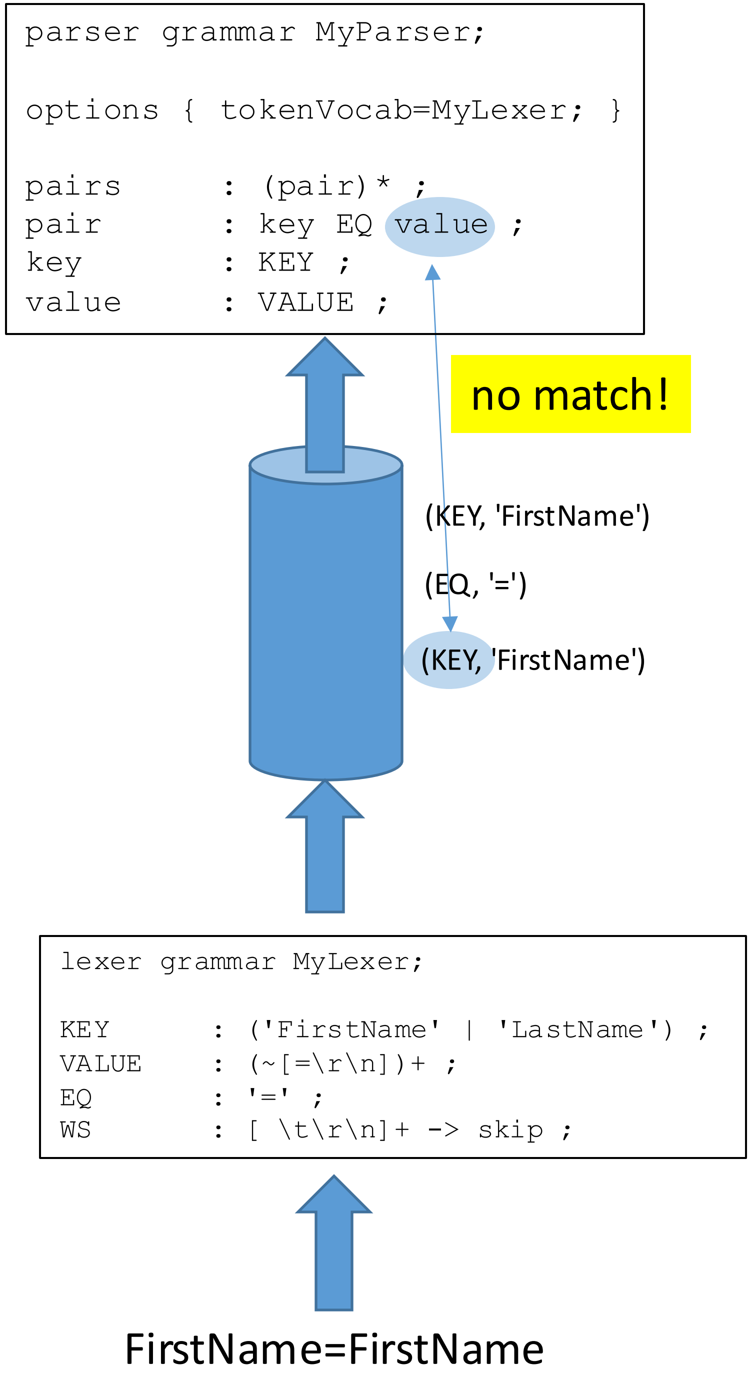
```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

pairs      : (pair)* ;
pair       : key EQ value ;
key        : KEY ;
value      : VALUE ;
```

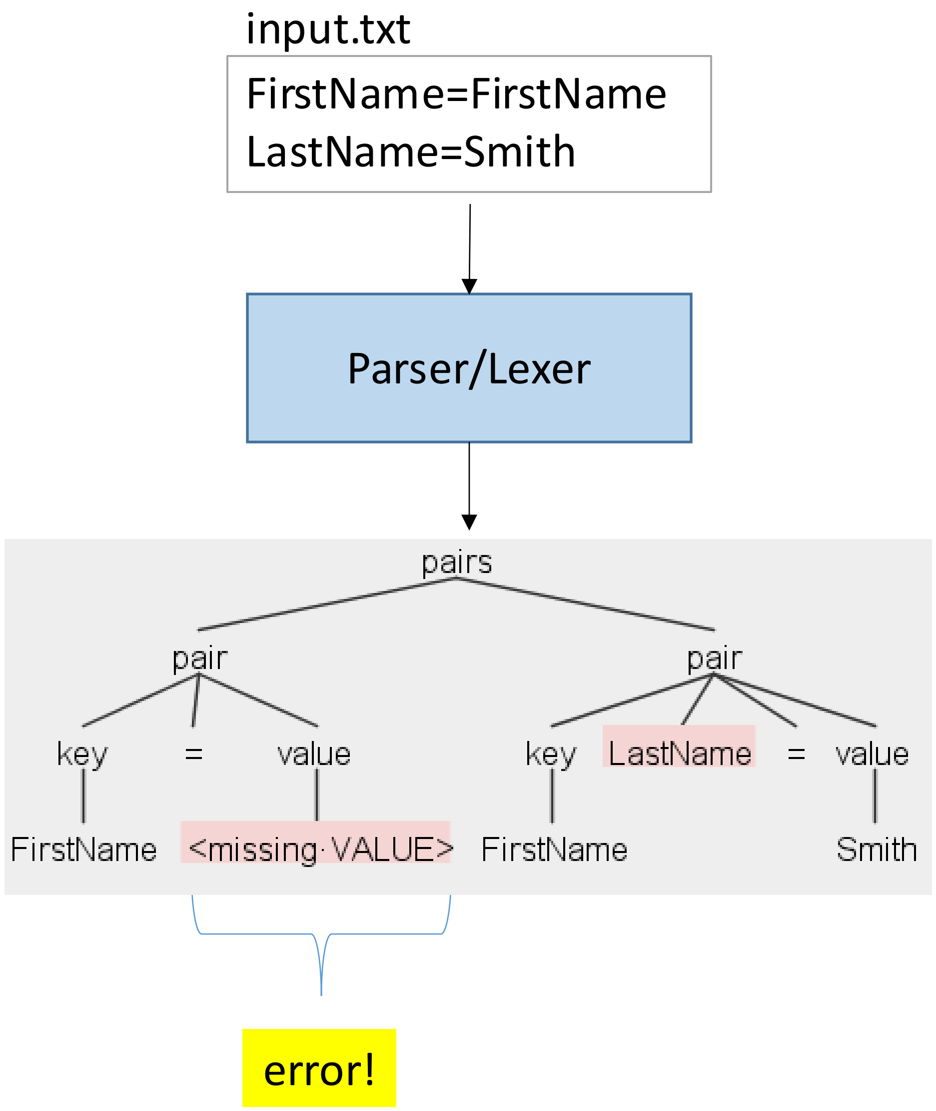



```
lexer grammar MyLexer;

KEY      : ('FirstName' | 'LastName') ;
VALUE    : (~[=\r\n])+ ;
EQ       : '=' ;
WS       : [ \t\r\n]+ -> skip ;
```

FirstName=FirstName


parser grammar MyParser;
options { tokenVocab=MyLexer; }
pairs : (pair)* ;
pair : key EQ value ;
key : KEY ;
value : VALUE ;
match
(KEY, 'FirstName')
(EQ, '=')
(KEY, 'FirstName')
lexer grammar MyLexer;
KEY : ('FirstName' | 'LastName') ;
VALUE : (~[=\r\n])+ ;
EQ : '=' ;
WS : [\t\r\n]+ -> skip ;
FirstName=FirstName


parser grammar MyParser;
options { tokenVocab=MyLexer; }
pairs : (pair)* ;
pair : key EQ value ;
key : KEY ;
value : VALUE ;
no match!
(KEY, 'FirstName')
(EQ, '=')
(KEY, 'FirstName')
lexer grammar MyLexer;
KEY : ('FirstName' | 'LastName') ;
VALUE : (~[=\r\n])+ ;
EQ : '=' ;
WS : [\t\r\n]+ -> skip ;
FirstName=FirstName


input.txt
FirstName=FirstName
LastName=Smith
Parser/Lexer
pairs
pair
pair
key
=
value
key
LastName
=
value
FirstName
<missing VALUE>
FirstName
Smith
error!

How do we solve the problem?

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

pairs     : (pair)* ;
pair      : key EQ value ;
key       : KEY ;
value     : VALUE ;
```

Hint: need to modify the parser grammar.
Leave the lexer grammar alone.

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

pairs      : (pair)* ;
pair       : key EQ value ;
key        : KEY ;
value      : VALUE ;
```

After the equals (EQ) token we must allow either value or key

The solution

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

pairs     : (pair)* ;
pair      : key EQ (key | value) ;
key       : KEY ;
value     : VALUE ;
```

See example20

Do Lab12

Context-sensitive problem #2

Dash is used for the separator and empty field

Use dash to denote empty field

Use dash as the separator

FirstName, LastName, Street, City, State, ZipCode
Mark,-, 4460 Stuart Street, Marion Center, PA, 15759
-
FirstName=John
LastName=Smith

input.txt

Will this correctly tokenize the dashes?

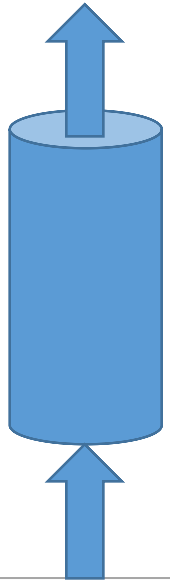
```
lexer grammar MyLexer;

COMMA   : ',' ;
DASH    : '-' ;
NL      : ('\r')?'\n' ;
WS      : [ \t\r\n]+ -> skip ;
SEPARATOR : SEP -> skip, pushMode(PAIRS) ;
STRING : (~[,\r\n])+ ;

fragment SEP : DASH NL ;

mode PAIRS ;
KEY        : ('FirstName' | 'LastName') ;
EQ         : '=' ;
NL2        : ('\r')?'\n' ;
WS2        : [ \t\r\n]+ -> skip ;
SEPARATOR2 : SEP2 -> skip, popMode ;
VALUE      : (~[=\r\n])+ ;

fragment SEP2 : '-' NL2 ;
```

How will the lexer tokenize this dash?

```
lexer grammar MyLexer;

COMMA  : ',' ;
DASH   : '-' ;
NL     : ('\r')?'\n' ;
WS     : [ \t\r\n]+ -> skip ;
SEPARATOR : SEP -> skip, pushMode(PAIRS) ;
STRING : (~[,\r\n])+ ;

fragment SEP : DASH NL ;

mode PAIRS ;
KEY         : ('FirstName' | 'LastName') ;
EQ          : '=' ;
NL2         : ('\r')?'\n' ;
WS2         : [ \t\r\n]+ -> skip ;
SEPARATOR2 : SEP2 -> skip, popMode ;
VALUE       : (~[=\r\n])+ ;

fragment SEP2 : '-' NL2 ;
```

```
FirstName, LastName, Street, City, State, ZipCode
Mark,-, 4460 Stuart Street, Marion Center, PA, 15759
-
FirstName=John
LastName=Smith
```

Parser



(DASH, '-')

FirstName, LastName, Street, City, State, ZipCode
Mark,-, 4460 Stuart Street, Marion Center, PA, 15759
-
FirstName=John
LastName=Smith

How will the lexer tokenize this dash?

```
lexer grammar MyLexer;

COMMA   : ',' ;
DASH    : '-' ;
NL      : ('\r')?'\n' ;
WS      : [ \t\r\n]+ -> skip ;
SEPARATOR : SEP -> skip, pushMode(PAIRS) ;
STRING : (~[,\r\n])+ ;

fragment SEP : DASH NL ;

mode PAIRS ;
KEY        : ('FirstName' | 'LastName') ;
EQ         : '=' ;
NL2        : ('\r')?'\n' ;
WS2        : [ \t\r\n]+ -> skip ;
SEPARATOR2 : SEP2 -> skip, popMode ;
VALUE      : (~[=\r\n])+ ;

fragment SEP2 : '-' NL2 ;
```

FirstName, LastName, Street, City, State, ZipCode
Mark,-, 4460 Stuart Street, Marion Center, PA, 15759
-
FirstName=John
LastName=Smith

(SEPARATOR, '-\r\n')

FirstName, LastName, Street, City, State, ZipCode
Mark,-, 4460 Stuart Street, Marion Center, PA, 15759
-\r\n
FirstName=John
LastName=Smith

Remember: the longest token is matched

```
lexer grammar MyLexer;

COMMA   : ',' ;
DASH    : '-' ;
NL      : ('\r')?'\n' ;
WS      : [ \t\r\n]+ -> skip ;
SEPARATOR : SEP -> skip, pushMode(PAIRS) ;
STRING : (~[,\r\n])+ ;

fragment SEP : DASH NL ;

mode PAIRS ;
KEY         : ('FirstName' | 'LastName') ;
EQ          : '=' ;
NL2         : ('\r')?'\n' ;
WS2         : [ \t\r\n]+ -> skip ;
SEPARATOR2 : SEP2 -> skip, popMode ;
VALUE       : (~[=\r\n])+ ;

fragment SEP2 : '-' NL2 ;
```

FirstName, LastName, Street, City, State, ZipCode
Mark,-, 4460 Stuart Street, Marion Center, PA, 15759
-
FirstName=John
LastName=Smith

The dash matches both DASH and SEP. However, the input contains a dash followed by a newline. So the lexer matches the longest token.

See example21

How will the lexer tokenize this dash?

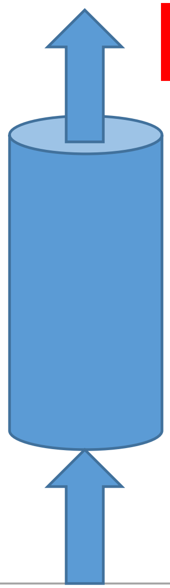
```
lexer grammar MyLexer;

COMMA   : ',' ;
DASH    : '-' ;
NL      : ('\r')?'\n' ;
WS      : [ \t\r\n]+ -> skip ;
SEPARATOR : SEP -> skip, pushMode(PAIRS) ;
STRING : (~[,\r\n])+ ;

fragment SEP : DASH ;

mode PAIRS ;
KEY         : ('FirstName' | 'LastName') ;
EQ          : '=' ;
NL2         : ('\r')?'\n' ;
WS2         : [ \t\r\n]+ -> skip ;
SEPARATOR2 : SEP2 -> skip, popMode ;
VALUE       : (~[=\r\n])+ ;

fragment SEP2 : '-' NL2 ;
```

FirstName, LastName, Street, City, State, ZipCode
Mark,-, 4460 Stuart Street, Marion Center, PA, 15759
-
FirstName=John
LastName=Smith

Notice: NL is removed

Parser

This results in an error



(DASH, '-')

FirstName, LastName, Street, City, State, ZipCode
Mark,-, 4460 Stuart Street, Marion Center, PA, 15759
-
FirstName=John
LastName=Smith

Context-sensitive problem #3

Continuations in CSV

- In previous examples the newline character indicated the end of a row.
- Let's allow a row to continue onto another line by placing a backslash at the end of the line:

row #1	field1, field2, \ field3
row #2	field4, field5, field6
row #3	field7, field8, field9

The lexer grammar

```
lexer grammar MyLexer;

COMMA        : ',' ;
NL           : ('\r')?'\n' ;
CONTINUATION : BACKSLASH NL -> skip ;
WS           : [ \t\r\n]+ -> skip ;
STRING       : (~[\\,\r\n])+ ;

fragment BACKSLASH    : '\\' ;
```

See example22

```
lexer grammar MyLexer;

COMMA          : ',' ;
NL             : ('\r')?'\n' ;
CONTINUATION : BACKSLASH NL -> skip ;
WS             : [ \t\r\n]+ -> skip ;
STRING         : (~[\\,\r\n])+ ;

fragment BACKSLASH    : '\\' ;
```

discard "backslash newline"

field1, field2, \
field3
field4, field5, field6
field7, field8, field9

The parser grammar (no changes)

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

document  : header rows EOF ;
header    : field (COMMA field)* NL ;
rows      : (row)* ;
row       : field (COMMA field)* NL ;
field     : STRING | ;
```

See example22

Continuations plus escaped commas in fields

In addition to using the backslash to continue a row, let's use the backslash within a field to escape a comma.

1st field

```
row #1   fie\,ld1, field2, \
         field3
row #2   field4, field5, field6
row #3   field7, field8, field9
```

The lexer grammar

```
lexer grammar MyLexer;

COMMA        : ',' ;
NL           : ('\r')?'\n' ;
CONTINUATION : BACKSLASH NL -> skip ;
WS           : [ \t\r\n]+ -> skip ;
STRING       : (ESCAPED_COMMA|~[\\,\r\n])+ ;

fragment BACKSLASH     : '\\' ;
fragment ESCAPED_COMMA : '\\,' ;
```

See example23

```
lexer grammar MyLexer;

COMMA        : ',' ;
NL           : ('\r')?'\n' ;
CONTINUATION : BACKSLASH NL -> skip ;
WS           : [ \t\r\n]+ -> skip ;
STRING       : (ESCAPED_COMMA|~[\\,\r\n])+ ;

fragment BACKSLASH     : '\\' ;
fragment ESCAPED_COMMA : '\\,' ;
```

A field (STRING) can contain escaped commas and/or any character except backslash, comma, or newline.

Why we discard whitespace

Here's why we discard whitespace

- We have been discarding whitespace using rules like this:

```
WS : [ \t\r\n]+ -> skip ;
```

- If we didn't do that, the parser rules would need to account for whitespace all over the place:

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }
document  : (header rows pairs)+ EOF ;
header    : field WS? (COMMA WS? field)* WS? NL ;
rows      : (row)* ;
row       : field  WS? (COMMA WS? field)* WS? NL ;
field     : STRING | ;

pairs     : (pair)* ;
pair      : key  WS? EQ WS? value WS? NL2;
key       : KEY ;
value     : VALUE ;
```

Ditto for comments

We also typically discard comments so that parser rules don't have to continually check for them.

```
LINE_COMMENT  : '//' .*? '\r'? '\n' ;
BLOCK_COMMENT : '/*' .*? '*/' ;
```

Common token rules

```
ID: ID_LETTER (ID_LETTER | DIGIT)* ;

STRING : '"' (ESC | .)*? '"' ;

LINE_COMMENT      : '//' .*? '\r'? '\n' ;
BLOCK_COMMENT     : '/*' .*? '*/' ;

WS  : [ \t\r\n]+ -> skip ;

fragment ID_LETTER : 'a' .. 'z' | 'A' .. 'Z' | '_' ;
fragment DIGIT : '0' .. '9' ;
fragment ESC : '\\' [btnr"\\] ;       backspace, tab, newline, etc.
```

Lexer command: more

The "more" lexer command

The "more" command tells the lexer to keep going (collect more input) and return to the parser the input collected using the current token rule (the rule with the "more" command) plus the input collected as a result of "keep going".

Compare a lexer rule without and with the "more" lexer command

```
CHAR : . ;
```

This token rule says: *Get the next character in the input and send it up to the parser*.

```
CHAR : . -> more ;
```

This token rule says: *Get the next character in the input, but don't stop there. Get the following character in the input. Keep gobbling up the input characters until the input matches a rule that doesn't have "more".*

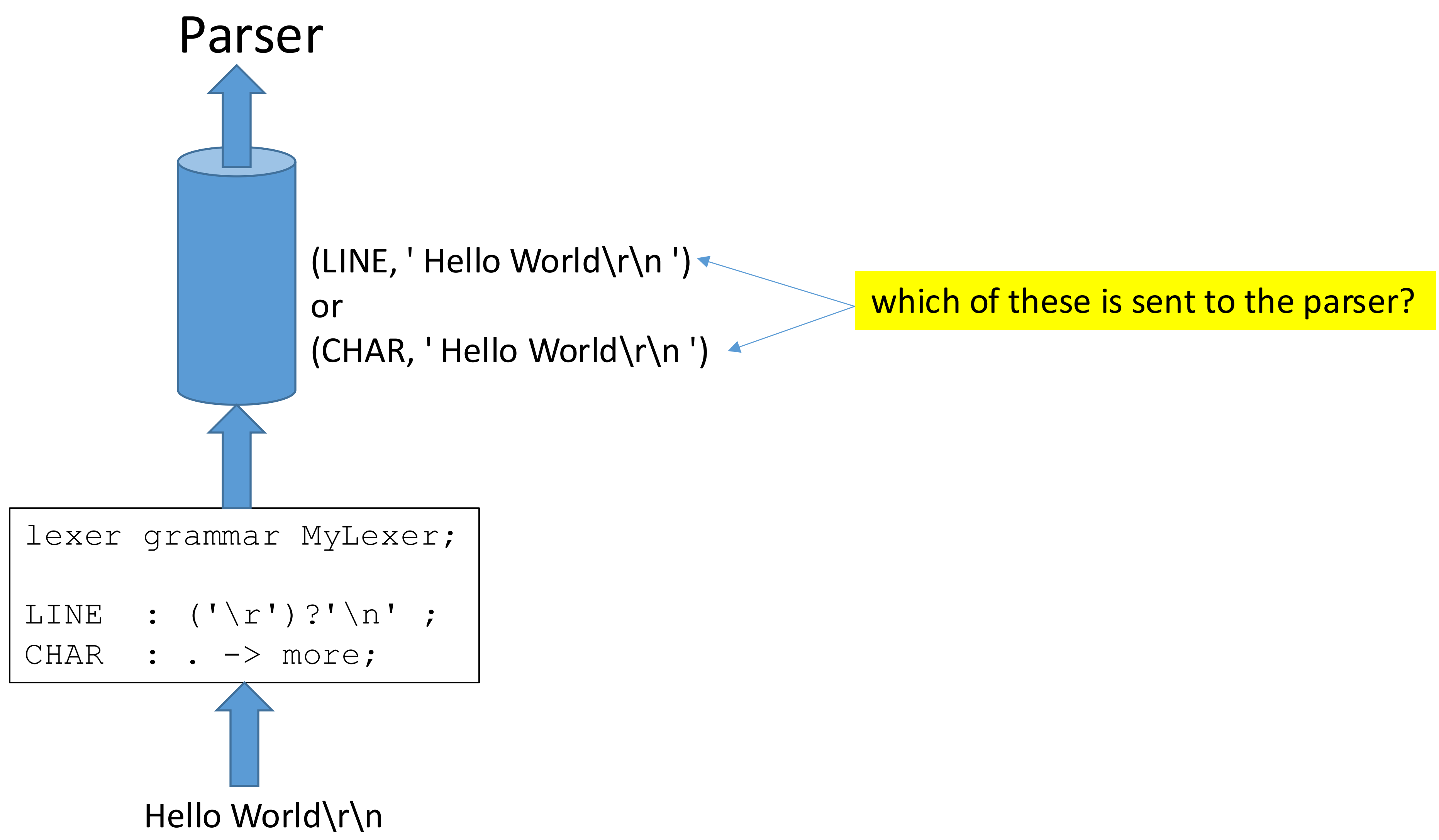
A lexer grammar that uses the "more" lexer command

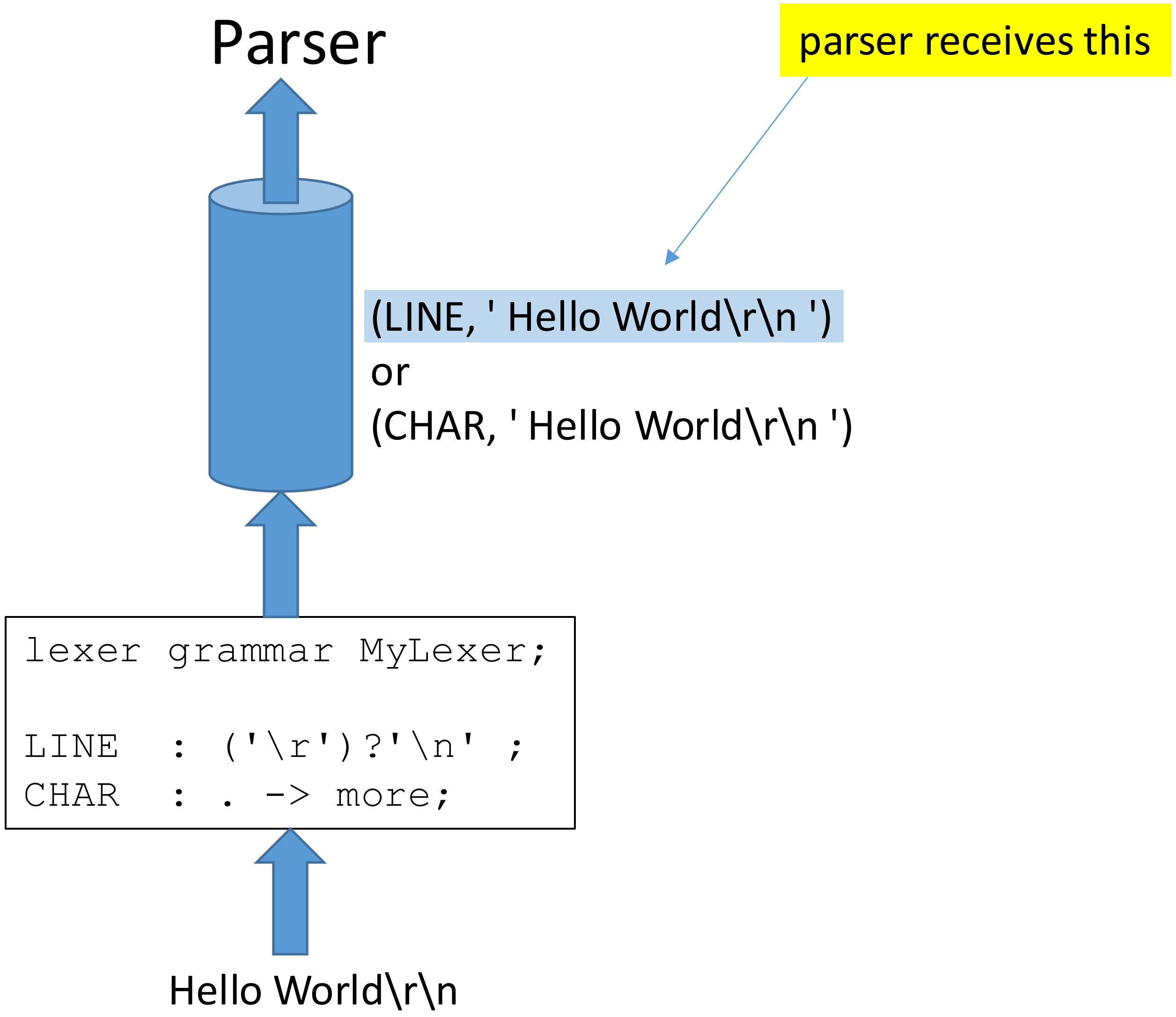
```
lexer grammar MyLexer;

LINE  : ('\r')?'\n' ;
CHAR  : . -> more;
```

See example24

```
lexer grammar MyLexer;

LINE   : ('\r')?'\n' ;
CHAR   : . -> more;
```

Keep gobbling up the input until getting to the newline, at which point the lexer switches to the LINE rule.

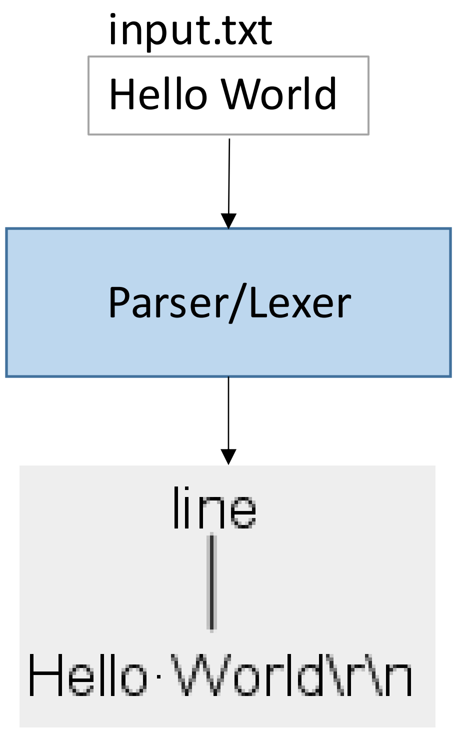

Parser
(LINE, ' Hello World\r\n ')
or
(CHAR, ' Hello World\r\n ')
which of these is sent to the parser?
lexer grammar MyLexer;
LINE : ('\r')?'\n' ;
CHAR : . -> more;
Hello World\r\n


Parser
parser receives this
(LINE, ' Hello World\r\n ')
or
(CHAR, ' Hello World\r\n ')
lexer grammar MyLexer;
LINE : ('\r')?'\n' ;
CHAR : . -> more;
Hello World\r\n

Parser grammar

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

line  : LINE ;
```

parser grammar is expecting to receive a LINE token (not a CHAR token)


input.txt
Hello World
Parser/Lexer
line
Hello·World\r\n

Equivalent

```
lexer grammar MyLexer;

LINE   : ('\r')?'\n' ;
CHAR   : .    -> more;
```

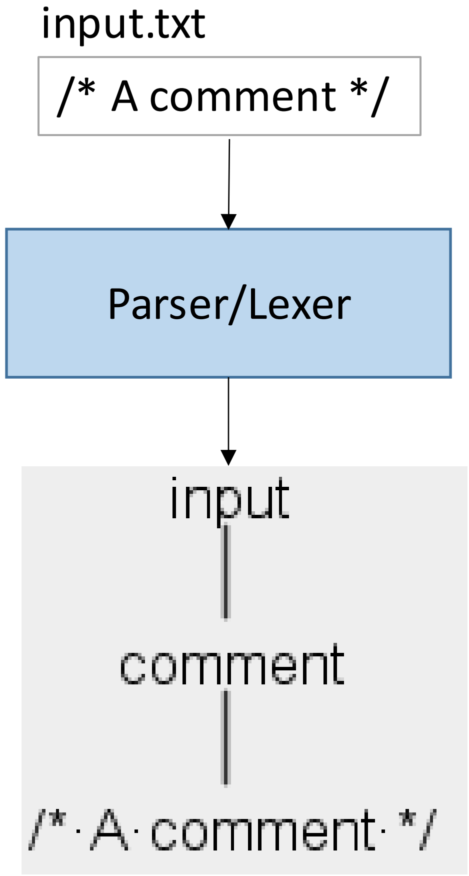
```
lexer grammar MyLexer;

LINE   : (.)+?('\r')?'\n' ;
```

non-greedy operator

Do Lab13

Non-greedy operator

Problem: Create a parser/lexer for a comment



Will this lexer rule work?

```
COMMENT : '/*' (.)* '*/'  ;
```

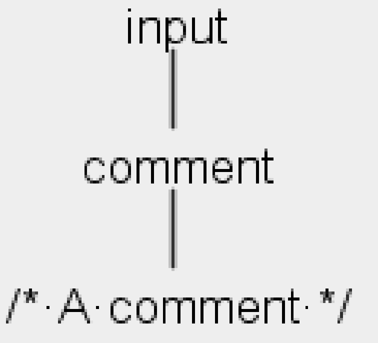
```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

input : comment ;

comment : COMMENT ;
```

```
lexer grammar MyLexer;

COMMENT : '/*' (.)* '*/' ;
```

input.txt

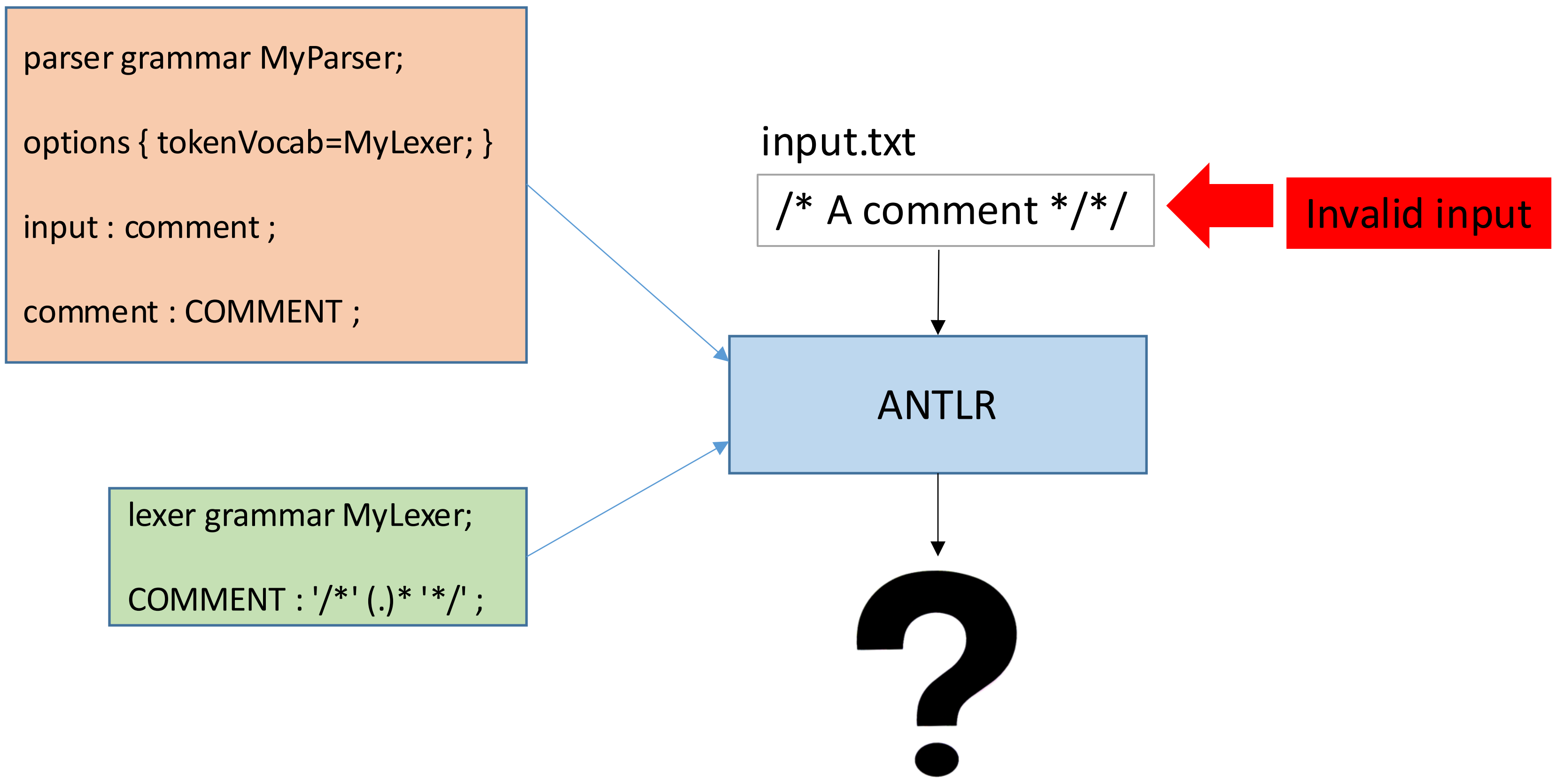
```
/* A comment */
```

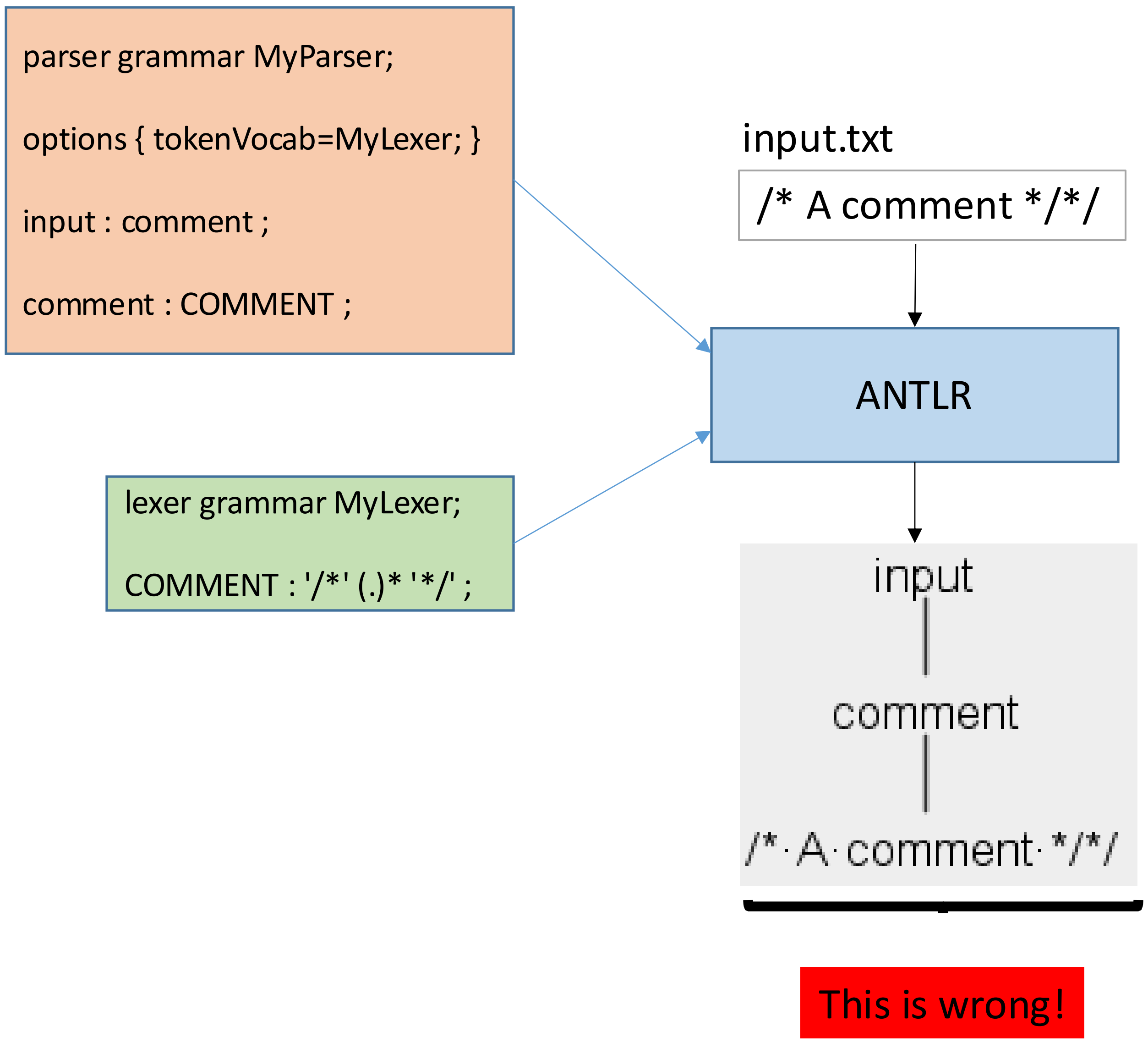
ANTLR

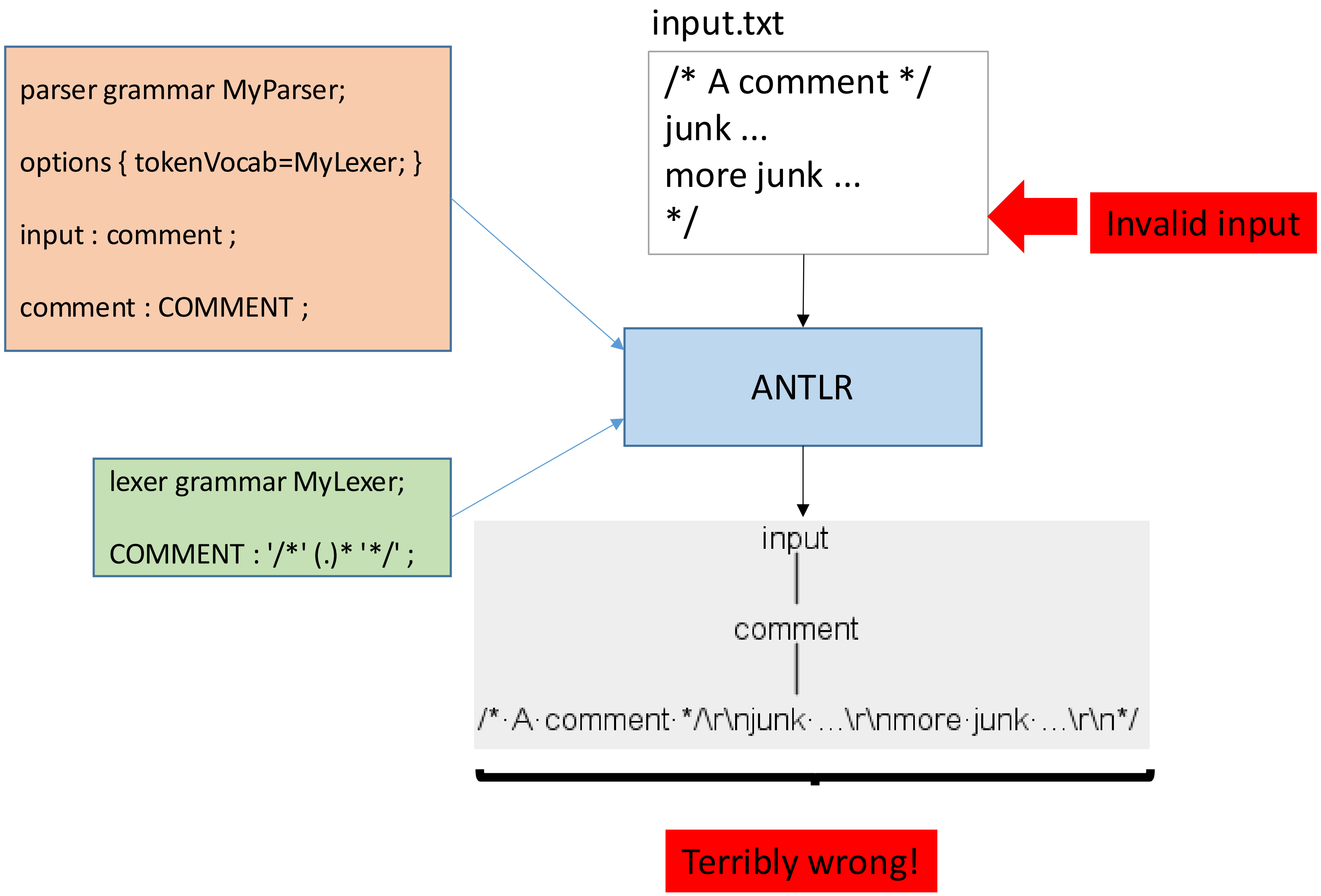


See example25

It works!
More accurately, it parses correct input.
But will it catch incorrect input?


parser grammar MyParser;
options { tokenVocab=MyLexer; }
input : comment ;
comment : COMMENT ;
lexer grammar MyLexer;
COMMENT : '/*' (.)* '*/' ;
input.txt
/* A comment */*/
Invalid input
ANTLR
?


parser grammar MyParser;
options { tokenVocab=MyLexer; }
input : comment ;
comment : COMMENT ;
lexer grammar MyLexer;
COMMENT : '/*' (.)* '*/' ;
input.txt
/* A comment */*/
ANTLR
input
comment
/*·A·comment·*/*/
This is wrong!



parser grammar MyParser;
options { tokenVocab=MyLexer; }
input : comment ;
comment : COMMENT ;
lexer grammar MyLexer;
COMMENT : '/*' (.)* '*/' ;
input.txt
/* A comment */
junk ...
more junk ...
*/
Invalid input
ANTLR
input
comment
/*·A·comment·*/\r\njunk·...\r\nmore·junk·...\r\n*/
Terribly wrong!

Greedy operator

```
COMMENT : '/*' (.)* '*/' ;
```

This is a greedy operator: it gobbles up as much input as possible such that '*/' remains to complete the token definition.

Non-greedy operator

```
COMMENT : '/*' (.)*? '*/' ;
```

This is a non-greedy operator: it gobbles up the minimum input needed such that '*/' remains to complete the token definition.

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

input : comment ;

comment : COMMENT ;
```

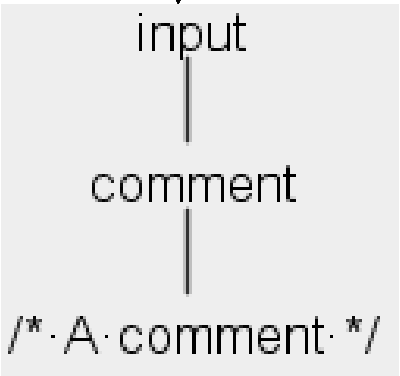
```
lexer grammar MyLexer;

COMMENT : '/*' (.)*? '*/' ;
```

non-greedy operator

input.txt

```
/* A comment */
junk ...
more junk ...
*/
```

ANTLR

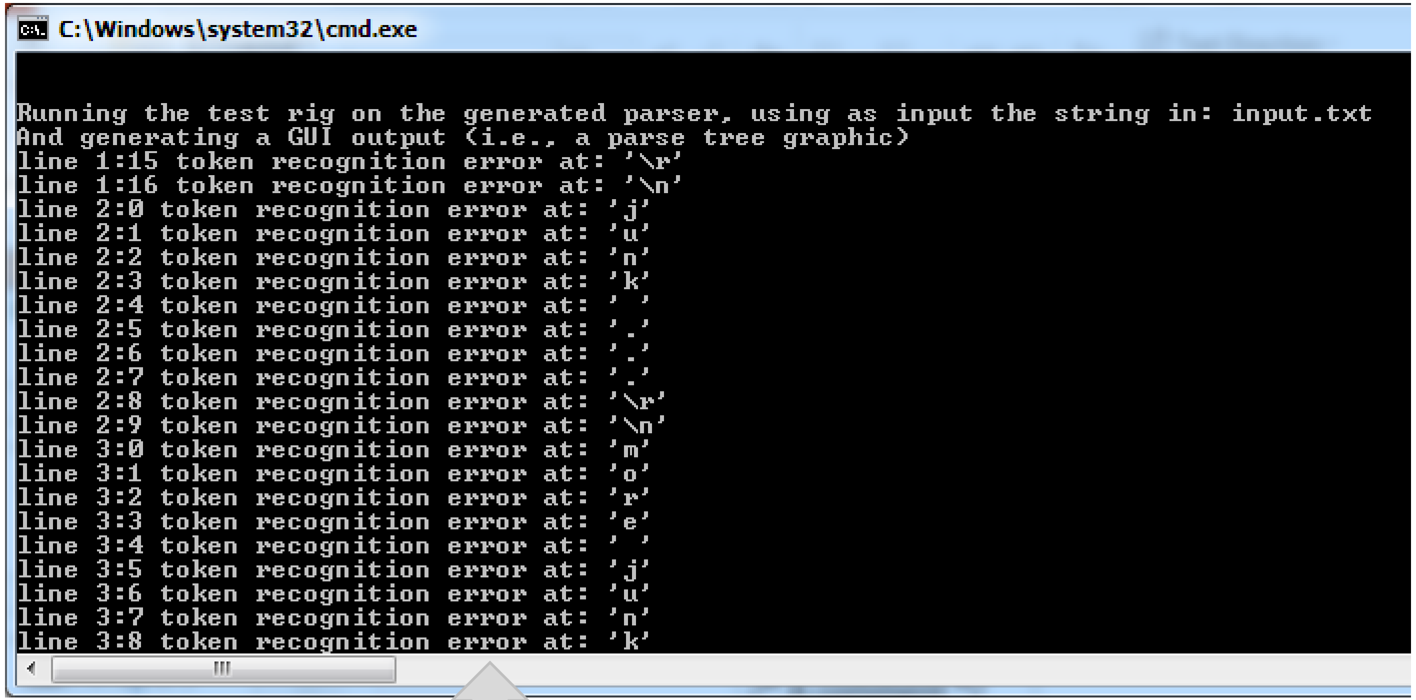


See example26

input.txt

```
/* A comment */
junk ...
more junk ...
*/
```

Lexer doesn't recognize this input (no rules for this input)



This is what we want. We want the lexer to find and report these errors!

Greedy vs. non-greedy operators

Greedy Operators

(...)*

(...)+

(...)?

Non-greedy Operators

(...)*?

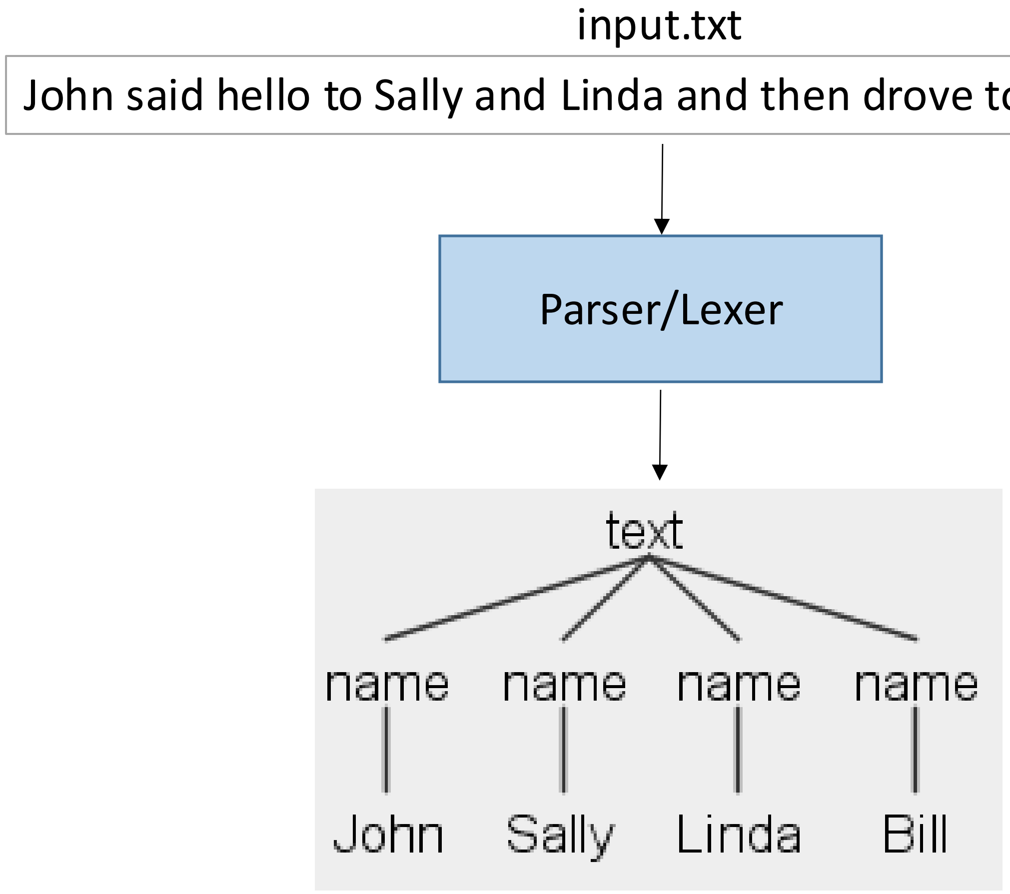
(...)+?

(...)??

Do Lab14

Fuzzy Parsing
(a.k.a. course-grain parsing)

Just want the names



Skip (discard) everything in the input except the names

The lexer grammar

```
lexer grammar MyLexer;

NAME : 'John' | 'Bill' | 'Sally' | 'Linda' ;

STUFF : (.)+? -> skip ;
```

See example27

```
lexer grammar MyLexer;

NAME : 'John' | 'Bill' | 'Sally' | 'Linda' ;

STUFF : (.)+? -> skip ;
```

Discard everything except names

The parser grammar

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

text : name* ;

name : NAME ;
```

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

text : name* ;

name : NAME ;
```

The parser is just expecting name tokens from the lexer

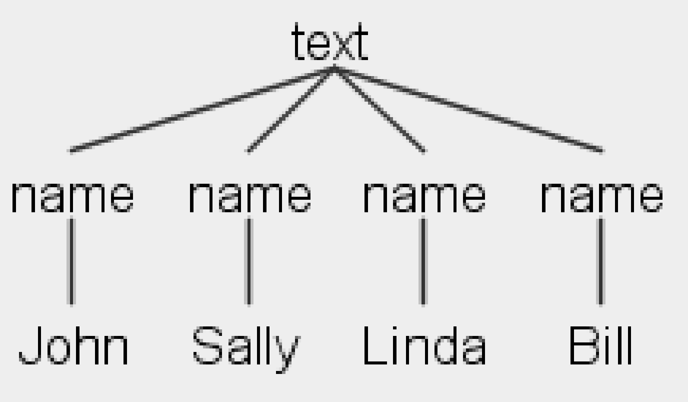
input.txt

John said hello to Sally and Linda and then drove to Bill's house.

```
lexer grammar MyLexer;

NAME : 'John' | 'Bill' | 'Sally' | 'Linda' ;

STUFF : (.)+? -> skip ;
```

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

text : name* ;

name : NAME ;
```

ANTLR



Do Lab15

We hardcoded the names in the lexer

```
lexer grammar MyLexer;

NAME : 'John' | 'Bill' | 'Sally' | 'Linda' ;

STUFF : (.)+? -> skip ;
```

hardcoded names. What if we didn't know the names? How would you write the lexer rules?

We hardcoded the names in the lexer

```
lexer grammar MyLexer;

NAME : 'John' | 'Bill' | 'Sally' | 'Linda' ;

STUFF : (.)+? -> skip ;
```

hardcoded names. What if we didn't know the names? How would you write the lexer rules?

Answer: I think it's impossible.
Lesson Learned: when designing a data format think hard about the items that people will want to extract from the data. If people can't create lexer rules to obtain the data, then the application will have to do it, and you've lost much of the benefit of parsing.

The Tokens Section

tokens { ... }

This is how to define a new token type.

```
lexer grammar MyLexer ;

tokens {
   STRING
}

ID : [a-zA-Z]+ ;

WS : [ \r\n] -> skip ;
```

Purpose and syntax

- The purpose of the **`tokens`** section is to define token types needed by a grammar for which there is no associated lexical rule.
- The syntax is:

tokens { Token1, ..., TokenN }

https://theantlrguy.atlassian.net/wiki/display/ANTLR4/Grammar+Structure#GrammarStructure-TokensSection

```
lexer grammar MyLexer ;

tokens { STRING }

ID : [a-zA-Z]+ ;

WS : [ \r\n] -> skip ;
```

ANTLR

```
STRING=1
ID=2
WS=3
```

MyLexer.tokens

See example28

See next section for a good use case for the tokens section →

Lexer command: type()

One token type for two different inputs

Problem: The input contains some strings delimited by single quotes and other strings delimited by double quotes. The parser doesn't care how the strings are delimited, it just wants to receive STRING tokens. You are to create lexer rules that send STRING tokens up to the parser, regardless of how the strings are delimited.

input.txt

```
"Hello"  'World'
```

The lexer grammar

```
lexer grammar MyLexer ;

tokens { STRING }

DOUBLE : '"' .*? '"'          -> type(STRING) ;

SINGLE : '\'' .*? '\''        -> type(STRING) ;

WS : [ \r\n\t]+               -> skip ;
```

Create a STRING token type

```
lexer grammar MyLexer ;

tokens { STRING }

DOUBLE : '"' .*? '"'          -> type(STRING) ;

SINGLE : '\'' .*? '\''        -> type(STRING) ;

WS : [ \r\n\t]+               -> skip ;
```

```
lexer grammar MyLexer ;

tokens { STRING }

DOUBLE : '"' .*? '"'          -> type(STRING) ;

SINGLE : '\'' .*? '\''        -> type(STRING) ;

WS : [ \r\n\t]+               -> skip ;
```

Use this lexer rule for input delimited by double quotes. Send up to the parser the input and use STRING as its type. For example, if the input is "Hello" then send up to the parser ("Hello", STRING)

```
lexer grammar MyLexer ;

tokens { STRING }

DOUBLE : '"' .*? '"'           -> type(STRING) ;

SINGLE : '\'' .*? '\''         -> type(STRING) ;

WS : [ \r\n\t]+                -> skip ;
```

Use this lexer rule for input delimited by single quotes. Send up to the parser the input and use STRING as its type. For example, if the input is 'World' then send up to the parser ('World', STRING)

The parser grammar

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

text : string* ;

string : STRING ;
```

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

text : string* ;

string : STRING ;
```

The parser expects to receive from the lexer a series of STRING tokens.

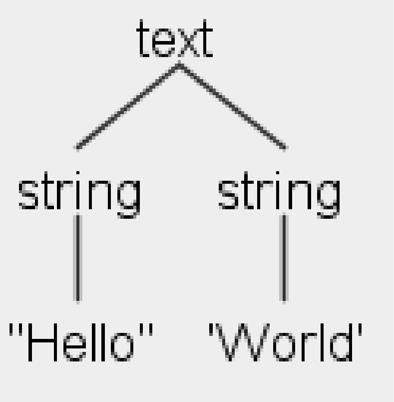
```
lexer grammar MyLexer ;

tokens { STRING }

DOUBLE : '"' .*? '"'        -> type(STRING) ;

SINGLE : '\'' .*? '\''      -> type(STRING) ;

WS : [ \r\n\t]+             -> skip ;
```

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

text : string* ;

string : STRING ;
```

input.txt

"Hello" 'World'

ANTLR



See example29

Do Lab16

Escaping characters using \uXXXX

Escape a character

You can escape characters using \uXXXX, where XXXX is the Unicode (hex) number of a character. For example, here are the hex numbers for each character in the name Roger:

R = \u0052

o = \u006F

g = \u0067

e = \u0065

r = \u0072

The lexer grammar

```
lexer grammar MyLexer;

NAME : '\u0052\u006F\u0067\u0065\u0072' ;

WS : [ \r\n\t]+    -> skip ;
```

The parser grammar

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

text : name EOF ;

name : NAME ;
```

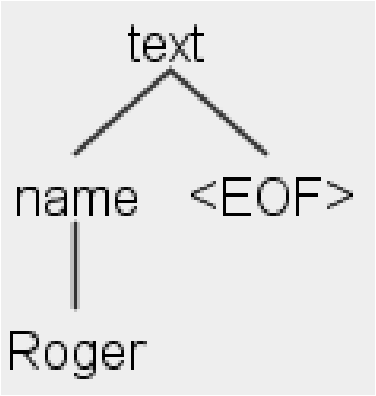
```
lexer grammar MyLexer;

NAME : '\u0052\u006F\u0067\u0065\u0072' ;

WS : [ \r\n\t]+   -> skip ;
```

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

text : name EOF ;

name : NAME ;
```

input.txt

Roger

ANTLR



See example30

Bad news

- The \uXXXX character escape mechanism works properly only with ASCII characters.
- I tried using \uXXXX with non-ASCII characters and ANTLR did not recognize the characters. For example, the code point for the ó character is hex F3. The UTF-8 encoding of code point F3 is two bytes (C3 B3). ANTLR erroneously processes those two bytes as two characters C3 = Ã and B3 = ³ rather than as one UTF-8 character, ó.
- Bummer.

Do Lab17

Empty alternative

Empty alternative

The following grammar says: A list contains one or more integers, followed by either '#' or nothing (the empty string):

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

list: INT+ endMarker ;

endMarker: EndMarker
         |
         ;
```

See example31

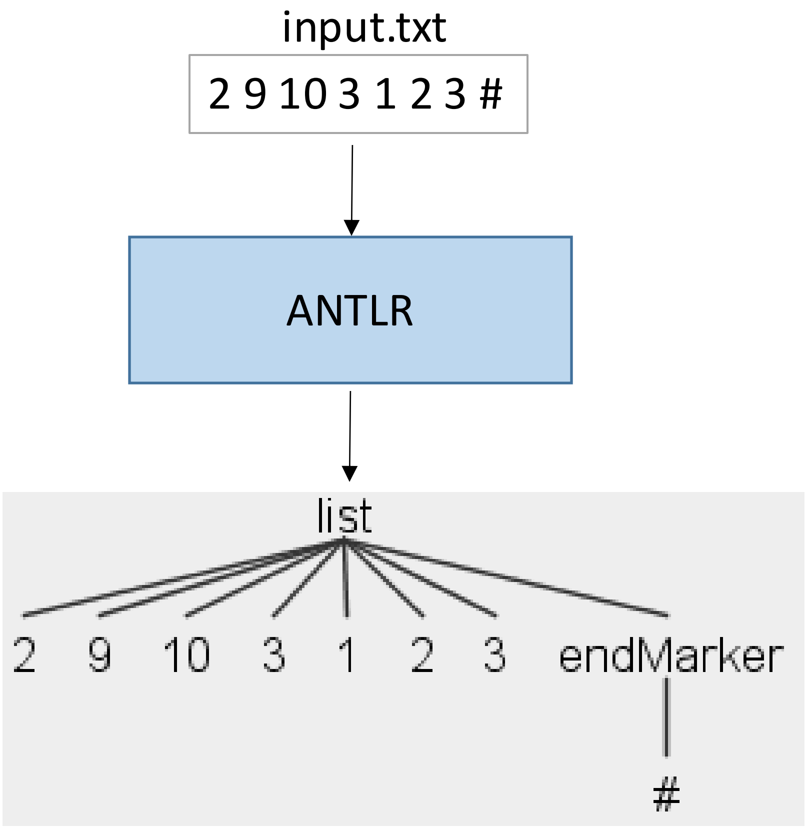
```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

list: INT+ endMarker ;

endMarker: EndMarker
         |
         ;
```

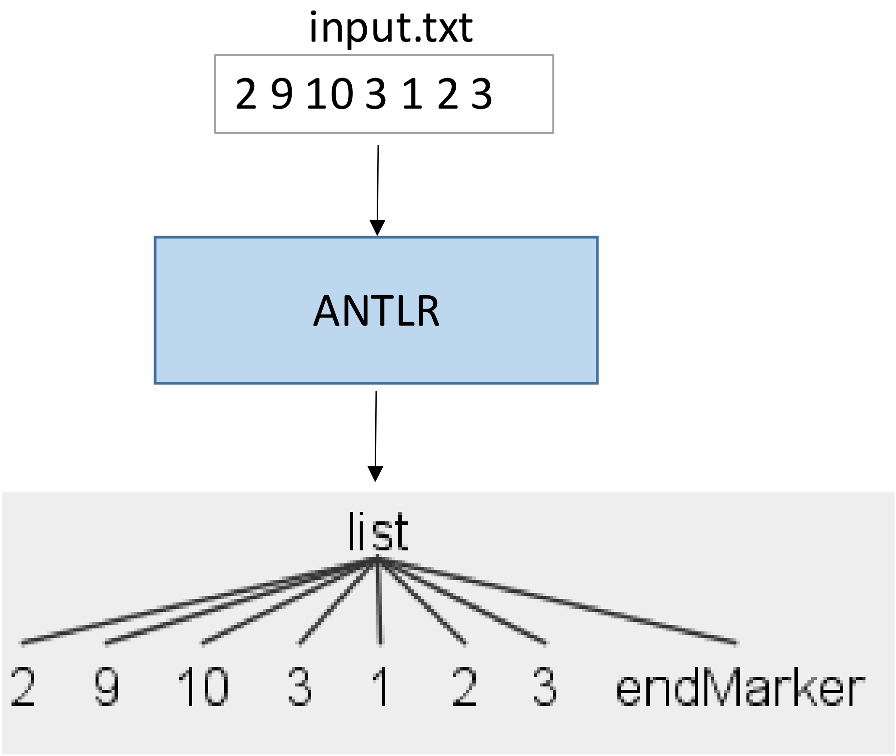
empty alternative

The lexer grammar

```
lexer grammar MyLexer;

INT  : [0-9]+ ;

EndMarker: '#' ;

WS  : [ \t\r\n]+ -> skip ;
```

Input with, and without, endMarker





Dot (.) in parser versus in lexer

Dot (.) is different in lexer and parser

- The dot (.) in a lexer rule means one *character*.
- The dot (.) in a parser rule means one *token*.
- That's a huge difference in meaning! Be careful!

```
lexer grammar MyLexer;

WORD  : 'Apple' . ;

WS  : [ \t\r\n]+ -> skip ;
```

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

words: .+ EOF ;
```

One character

One token

See example32

Do Lab18

Lexer command: channel

Channels

- The secret to preserving but ignoring whitespace is to send the tokens to the parser on a different *channel*.
- A parser tunes to only a single channel. We can pass anything we want on the other channel(s).

The "channel" lexer command

channel(*name*): send the tokens to the parser on channel *name*.

```
WS : [ \t\r\n]+ -> channel(1) ;
```

"*Hey, transmit to the parser the whitespace tokens on channel 1*."

Channels

- Channels are like different radio frequencies. The parser tunes to one channel and ignores tokens on other channels.
- Lexer rules are responsible for putting tokens on different channels.
- Channels allow us to categorize the input tokens.

Default channel

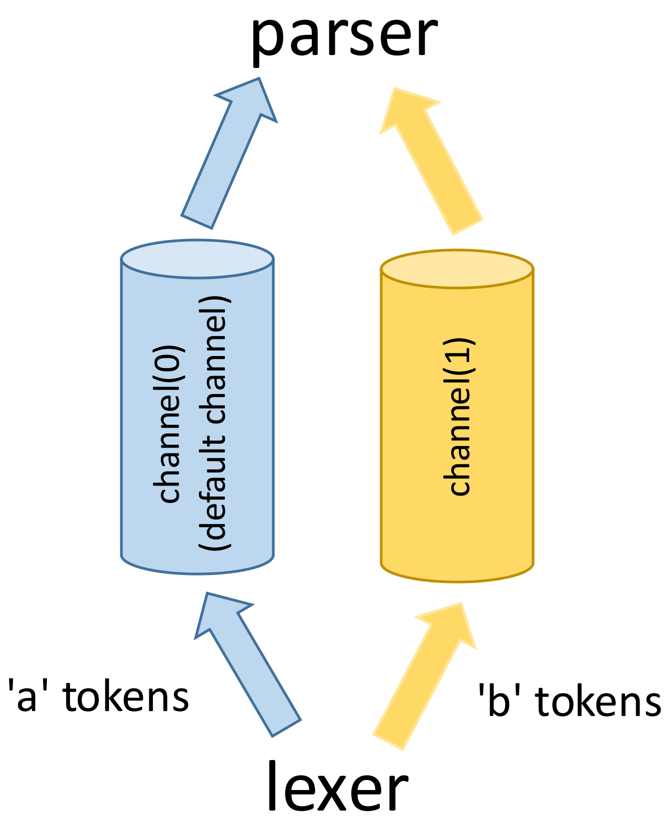
- In all our lexers thus far we have not specified a channel, so what channel have those tokens been going on?
- Answer: if you don't specify a channel, the token is sent up to the parser on channel 0 (the default channel).

Problem

- The input is a list of **`a`**s and **`b`**s, intermingled.
- Disentangle them: put the **`a`**s on one channel, the **`b`**s on another channel
- Here's a sample input: **`abaa`**

Put **a**s and **b**s on different channels

Let's put the 'a' tokens on the default channel and the 'b' tokens on channel 1.



The lexer grammar

```
lexer grammar MyLexer;

A   : 'a' -> channel(0) ;
B   : 'b' -> channel(1) ;
WS      : [ \t\r\n]+ -> skip ;
```

The parser grammar

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

text : (a | b)+ EOF ;
a    : A ;
b    : B ;
```

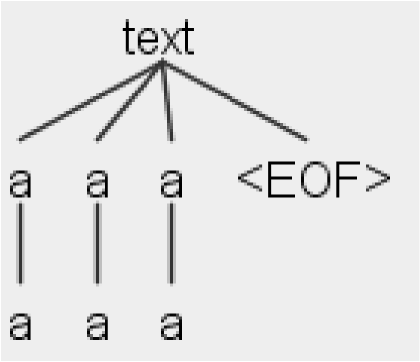
```
lexer grammar MyLexer;

A   : 'a' -> channel(0) ;
B   : 'b' -> channel(1) ;
WS      : [ \t\r\n]+   -> skip ;
```

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

text : (a | b)+ EOF ;
a     : A ;
b     : B ;
```

input.txt

aaba

ANTLR

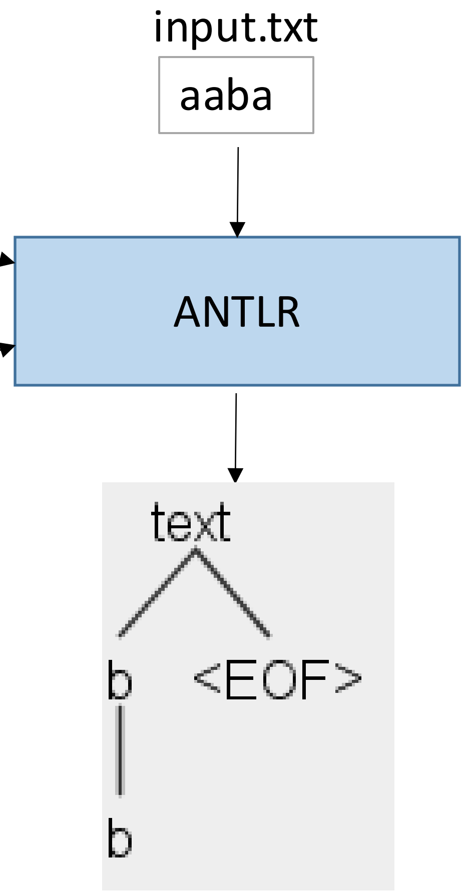


See example33

Swap channels

```
lexer grammar MyLexer;

A  : 'a' -> channel(1) ;
B  : 'b' -> channel(0) ;
WS     : [ \t\r\n]+   -> skip ;
```

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

text : (a | b)+ EOF ;
a    : A ;
b    : B ;
```



Parser rules for different channels?

- We've seen how the lexer can categorize tokens by putting them on different channels.
- Is there a way to create parser rules for each channel; for example, here are the parser rules for channel 1, there are the parser rules for channel 2, and so forth?

Answer

- The default channel is channel(0).
- Whatever tokens are placed on channel(0) will be processed by the parser rules.
- There is no way to create parser rules for other channels.

Lexer commands

Here are the lexer commands

- skip
- more
- popMode
- mode(x)
- pushMode(x)
- type(x)
- channel(x)

Common Token Rules

Common token rules

```
ID              : ID_LETTER (ID_LETTER | DIGIT)* ;

STRING          : '"' (ESC | .)*? '"' ;

LINE_COMMENT    : '//' .*? '\r'? '\n' ;
BLOCK_COMMENT   : '/*' .*? '*/' ;

WS              : [ \t\r\n]+ -> skip ;

fragment ID_LETTER : 'a' .. 'z' | 'A' .. 'Z' | '_' ;
fragment DIGIT     : '0' .. '9' ;
fragment ESC       : '\\' [btnr"\\] ;
```

```
ID               : ID_LETTER (ID_LETTER | DIGIT)* ;

STRING           : '"' (ESC | .)*? '"' ;

LINE_COMMENT     : '//' .*? '\r'? '\n' ;
BLOCK_COMMENT    : '/*' .*? '*/' ;

WS               : [ \t\r\n]+ -> skip ;

fragment ID_LETTER : 'a' .. 'z' | 'A' .. 'Z' | '_' ;
fragment DIGIT     : '0' .. '9' ;
fragment ESC       : '\\' [btnr"\\] ;
```

Helper rules. Can be used by the lexer rules.

```
ID                : ID_LETTER (ID_LETTER | DIGIT)* ;

STRING            : '"' (ESC | .)*? '"' ;

LINE_COMMENT      : '//' .*? '\r'? '\n' ;
BLOCK_COMMENT     : '/*' .*? '*/' ;

WS                : [ \t\r\n]+ -> skip ;

fragment ID_LETTER : 'a' .. 'z' | 'A' .. 'Z' | '_' ;
fragment DIGIT     : '0' .. '9' ;
fragment ESC       : '\\' [btnr"\\] ;
```

An ID_LETTER is a lowercase or uppercase letter or an underscore.

```
ID                : ID_LETTER (ID_LETTER | DIGIT)* ;

STRING            : '"' (ESC | .)*? '"' ;

LINE_COMMENT      : '//' .*? '\r'? '\n' ;
BLOCK_COMMENT     : '/*' .*? '*/' ;

WS                : [ \t\r\n]+ -> skip ;

fragment ID_LETTER : 'a' .. 'z' | 'A' .. 'Z' | '_' ;
fragment DIGIT     : '0' .. '9' ;
fragment ESC       : '\\' [btnr"\\] ;
```

A DIGIT is one of: '0' through '9'.

```
ID               : ID_LETTER (ID_LETTER | DIGIT)* ;

STRING           : '"' (ESC | .)*? '"' ;

LINE_COMMENT     : '//' .*? '\r'? '\n' ;
BLOCK_COMMENT    : '/*' .*? '*/' ;

WS               : [ \t\r\n]+ -> skip ;

fragment ID_LETTER : 'a' .. 'z' | 'A' .. 'Z' | '_' ;
fragment DIGIT     : '0' .. '9' ;
fragment ESC       : '\\' [btnr"\\] ;
```

A ESC character is one of: \b, \t, \n, \r, \", or \\.

An ID token is one that starts with a letter or underscore, followed by zero or more letters, underscores, and digits.

```
ID                : ID_LETTER (ID_LETTER | DIGIT)* ;

STRING            : '"' (ESC | .)*? '"' ;

LINE_COMMENT      : '//' .*? '\r'? '\n' ;
BLOCK_COMMENT     : '/*' .*? '*/' ;

WS                : [ \t\r\n]+ -> skip ;

fragment ID_LETTER : 'a' .. 'z' | 'A' .. 'Z' | '_' ;
fragment DIGIT     : '0' .. '9' ;
fragment ESC       : '\\' [btnr"\\] ;
```

A STRING token is a series of characters embedded within quotes.

```
ID               : ID_LETTER (ID_LETTER | DIGIT)* ;

STRING           : '"' (ESC | .)*? '"' ;

LINE_COMMENT     : '//' .*? '\r'? '\n' ;
BLOCK_COMMENT    : '/*' .*? '*/' ;

WS               : [ \t\r\n]+ -> skip ;

fragment ID_LETTER : 'a' .. 'z' | 'A' .. 'Z' | '_' ;
fragment DIGIT     : '0' .. '9' ;
fragment ESC       : '\\' [btnr"\\] ;
```

Non-greedy operator: stop gobbling up the input upon reaching the first matching quote.

```
ID                : ID_LETTER (ID_LETTER | DIGIT)* ;

STRING            : '"' (ESC | .)*? '"' ;

LINE_COMMENT      : '//' .*? '\r'? '\n' ;
BLOCK_COMMENT     : '/*' .*? '*/' ;

WS                : [ \t\r\n]+ -> skip ;

fragment ID_LETTER : 'a' .. 'z' | 'A' .. 'Z' | '_' ;
fragment DIGIT     : '0' .. '9' ;
fragment ESC       : '\\' [btnr"\\] ;
```

A LINE_COMMENT token starts with two forward slashes and goes to the end of the line.

```
ID                : ID_LETTER (ID_LETTER | DIGIT)* ;

STRING            : '"' (ESC | .)*? '"' ;

LINE_COMMENT      : '//' .*? '\r'? '\n' ;
BLOCK_COMMENT     : '/*' .*? '*/' ;

WS                : [ \t\r\n]+ -> skip ;

fragment ID_LETTER : 'a' .. 'z' | 'A' .. 'Z' | '_' ;
fragment DIGIT     : '0' .. '9' ;
fragment ESC       : '\\' [btnr"\\] ;
```

A BLOCK_COMMENT token starts with /* and ends with */.

```
ID                  : ID_LETTER (ID_LETTER | DIGIT)* ;

STRING              : '"' (ESC | .)*? '"' ;

LINE_COMMENT        : '//' .*? '\r'? '\n' ;
BLOCK_COMMENT       : '/*' .*? '*/' ;

WS                  : [ \t\r\n]+ -> skip ;

fragment ID_LETTER : 'a' .. 'z' | 'A' .. 'Z' | '_' ;
fragment DIGIT     : '0' .. '9' ;
fragment ESC       : '\\' [btnr"\\] ;
```

A WS (whitespace) token is a series of spaces, tabs, carriage returns, and/or newlines. Discard these tokens.

```
ID                : ID_LETTER (ID_LETTER | DIGIT)* ;

STRING            : '"' (ESC | .)*? '"' ;

LINE_COMMENT      : '//' .*? '\r'? '\n' ;
BLOCK_COMMENT     : '/*' .*? '*/' ;

WS                : [ \t\r\n]+ -> skip ;

fragment ID_LETTER : 'a' .. 'z' | 'A' .. 'Z' | '_' ;
fragment DIGIT     : '0' .. '9' ;
fragment ESC       : '\\' [btnr"\\] ;
```

Library of ANTLR lexer/parser grammars

- https://github.com/antlr/grammars-v4
- At that web site you will find ANTLR grammars for parsing JSON, XML, CSV, C, C++, icalendar, mysql, and many others.

Changing operator association in a parse tree

This section might not be terribly useful

- All of the previous slides have focused on parsing data formats. I believe that will be the common case.
- The next few slides discuss an issue that is only relevant when parsing a programming language (e.g., when parsing a C program or a Java program). I believe this will be less common.

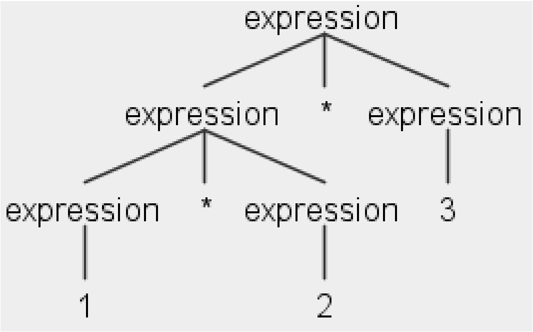
Operator association

- Consider this parser rule for arithmetic expressions:

```
expression: expression MULT expression
          | INT
          ;
```

See example34

- Question: How will this input be parsed: 1 * 2 * 3
- Answer: By default, ANTLR associates operators left to right, so the input is parsed this way: (1 * 2) * 3

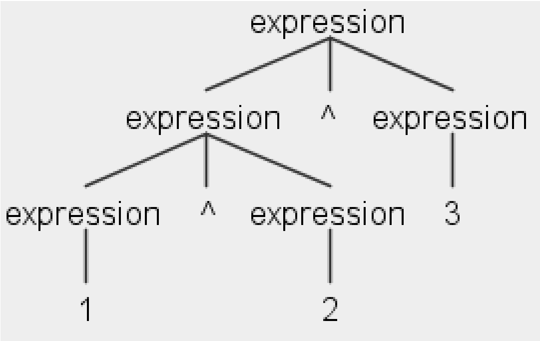


Operator association (cont.)

- Suppose the arithmetic operator is exponentiation (^):

```
expression: expression EXPON expression
          | INT
          ;
```

- Question: How will this input be parsed: 1 ^ 2 ^ 3
- Answer: Again, the default is to associate left to right, so the input is parsed this way: (1 ^ 2) ^ 3



- However, that's not right. The exponentiation operator should associate right-to-left, like this: 1 ^ (2 ^ 3)

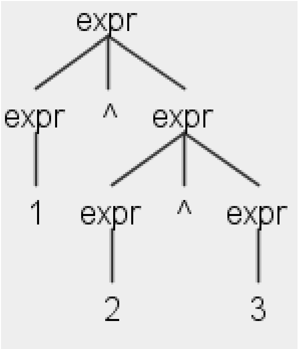
The assoc option

- We can instruct ANTLR on how we want an operator associated, using the assoc option:

```
expr:<assoc=right> expr '^' expr
    | INT
    ;
```

See example35

- Now this input 1 ^ 2 ^ 3 is parsed this way 1 ^ (2 ^ 3)



ANTLR Grammar
for
ANTLR Grammars

Grammar for grammars

- When you create a grammar, each rule is required to follow a certain form:

  rule-name colon alternatives-separated-by-vertical-bar semicolon

- Inside of ANTLR it expresses that requirement using a grammar:

```
rule: ID ':' alternative ('|' alternative)* ';' ;
```

grammar rule for grammar rules!

https://github.com/antlr/grammars-v4/blob/master/antlr4/ANTLRv4Parser.g4

https://github.com/antlr/grammars-v4/blob/master/antlr4/ANTLRv4Lexer.g4

Limitations of ANTLR

Grammar for a sequence of integers

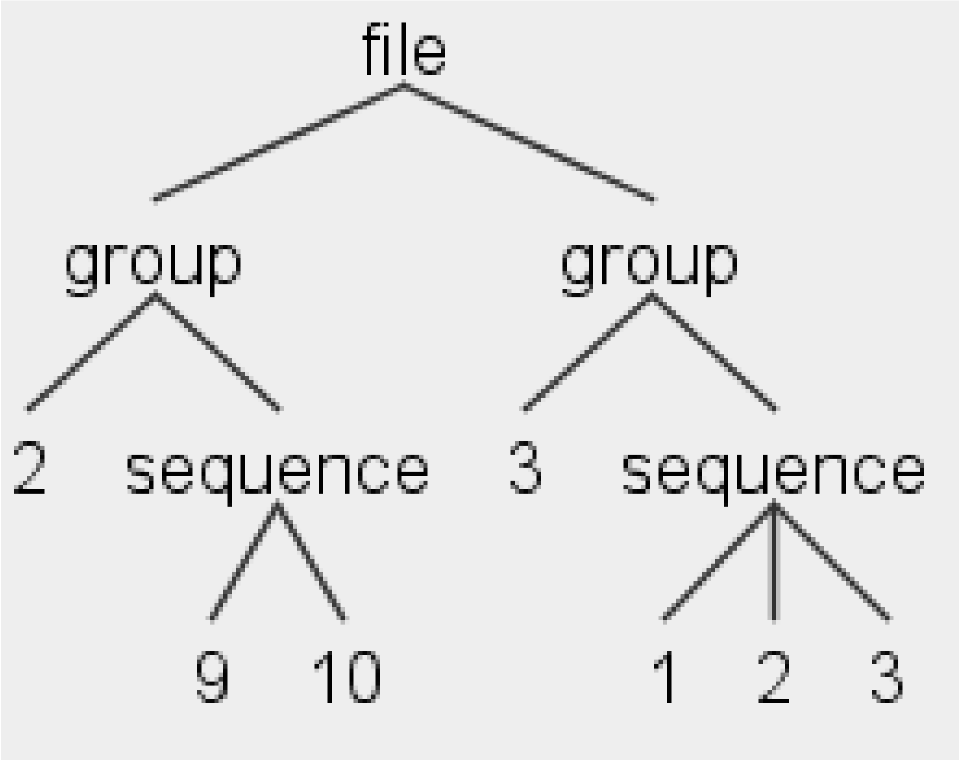
Create a grammar for this input:

2 9 10 3 1 2 3

indicates that there are 2 following integers

indicates that there are 3 following integers

Here is the parse tree we desire:



Will this grammar do the job?

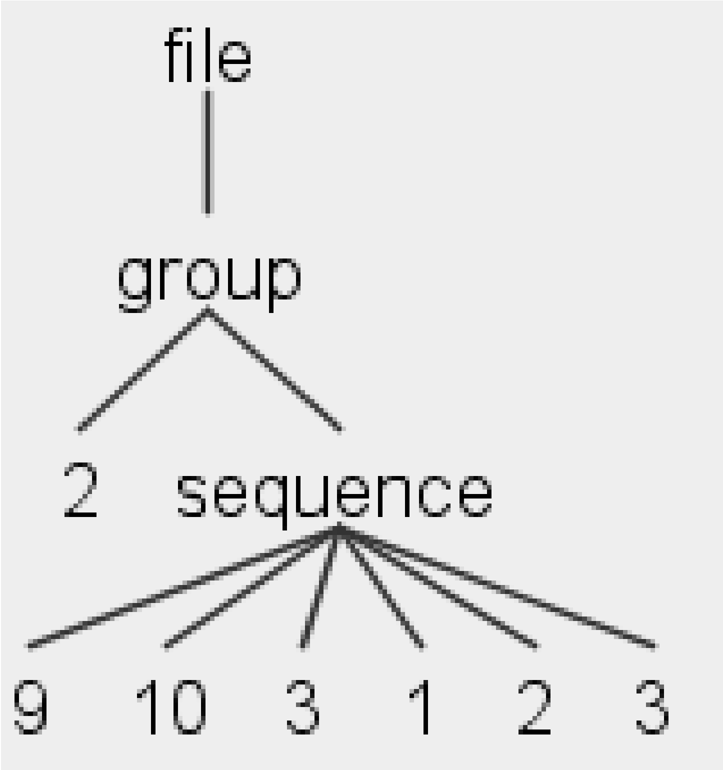
```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file: group+ ;

group: INT sequence ;

sequence: ( INT )* ;
```

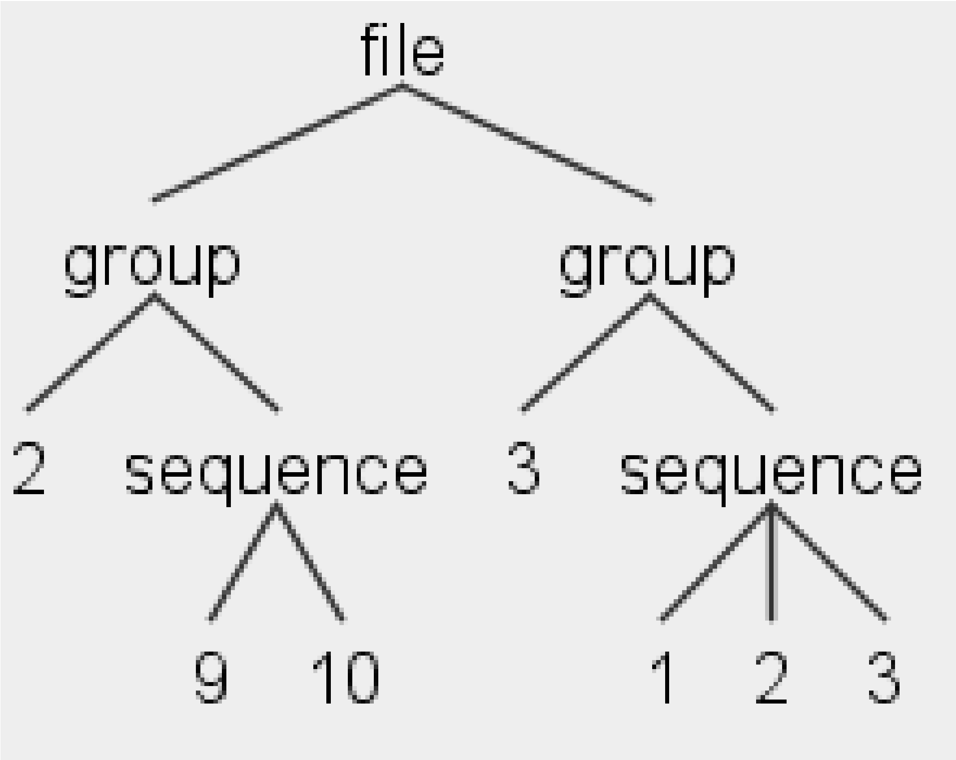
No!

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file: group+ ;

group: INT sequence ;

sequence: ( INT )* ;
```

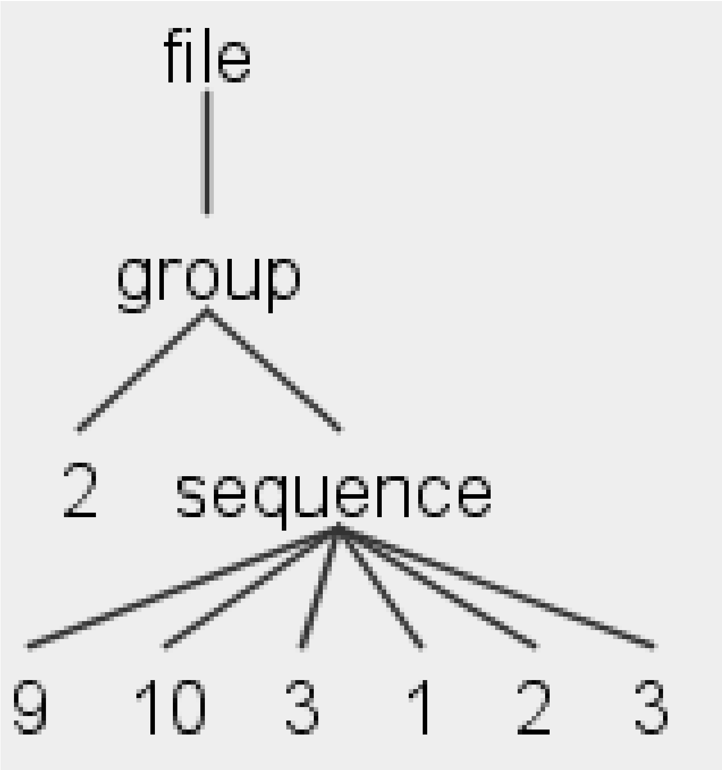
No restriction on the number of INT values within sequence.

Here's the parse tree that is generated:





want



don't want

Runtime determination of # of occurrences

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file: group+ ;

group: INT sequence ;

sequence: ( INT )* ;
```

this value needs to dictate how many of these are allowed

Bad news

- ANTLR cannot solve this problem.
- Well, it can solve this problem, but it requires inserting programming-language-specific code (e.g., Java code) into the grammar.

Error: stack overflow

Simplify

- There might come a time when you run ANTLR on your grammar and you get a "Stack Overflow" message.
- In all likelihood the reason is that you have a rule with loops within loops. The rule needs to be simplified.
- How would you simplify this lexer rule:

```
( ('a')* | ('b')* | ('c')* )* ;
```

(('a')* | ('b')* | ('c')*)* ;

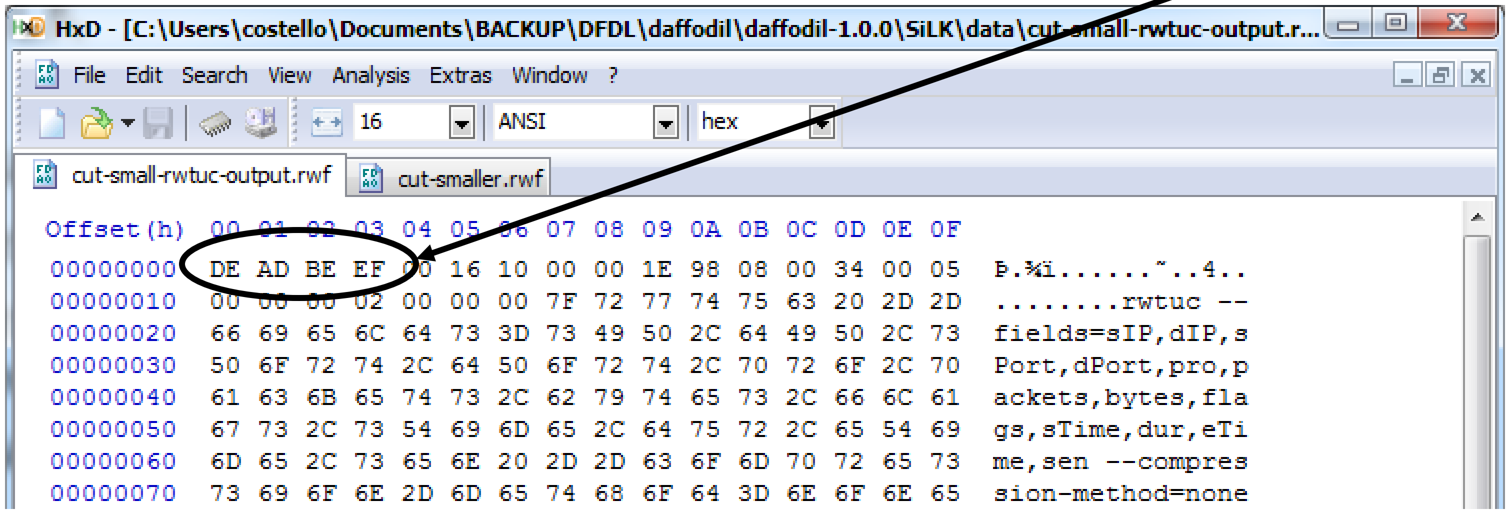
simplify

('a' | 'b' | 'c')* ;

Using ANTLR with binary files

SiLK

- SiLK = System for Internet-Level Knowledge
- It's a binary data format
- It is used to record information about data that flows through network devices (routers, gateways)
- The first 4 bytes of a SiLK file must be hex DEAD BEEF (magic bytes)



Problem: Is it a SiLK file?

- Write a lexer/parser to decide if the input is a SiLK file.
- To do this, check that the file has the correct magic bytes.
- Discard all other bytes.

The lexer grammar

```
lexer grammar MyLexer;

MagicBytes        : '\u00DE' '\u00AD' '\u00BE' '\u00EF' -> pushMode(OTHERBYTES);

mode OTHERBYTES ;
Bytes             : [\u0000-\u00FF] -> skip ;
```

Hex DEAD BEEF must be the first 4 bytes.

```
lexer grammar MyLexer;

MagicBytes        : '\u00DE' '\u00AD' '\u00BE' '\u00EF' -> pushMode(OTHERBYTES);

mode OTHERBYTES ;
Bytes             : [\u0000-\u00FF] -> skip ;
```

```
lexer grammar MyLexer;

MagicBytes        : '\u00DE' '\u00AD' '\u00BE' '\u00EF' -> pushMode(OTHERBYTES);

mode OTHERBYTES ;
Bytes             : [\u0000-\u00FF] -> skip ;
```

Discard all bytes following the magic bytes

The parser grammar

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

silk : magic_bytes EOF ;

magic_bytes : dead_beef ;

dead_beef : MagicBytes ;
```

See example36

Do Lab19

Embedding code in grammars

Problem #1

Grammar for a sequence of integers

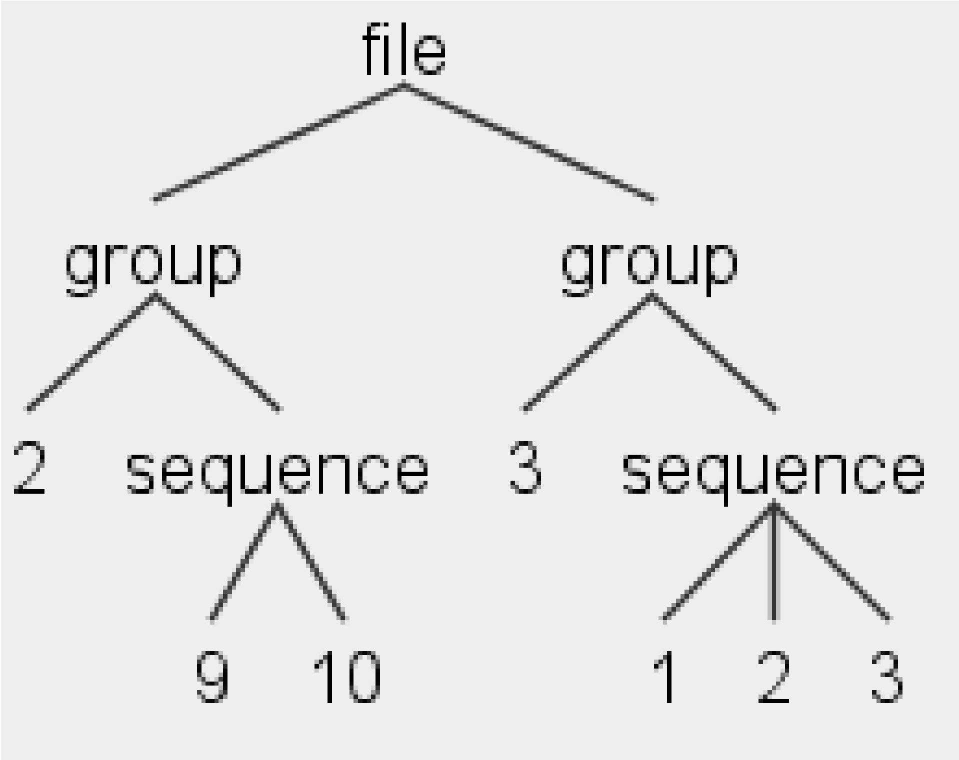
Create a grammar for this input:

2 9 10 3 1 2 3

indicates that there are 2 following integers

indicates that there are 3 following integers

Here is the parse tree we desire:



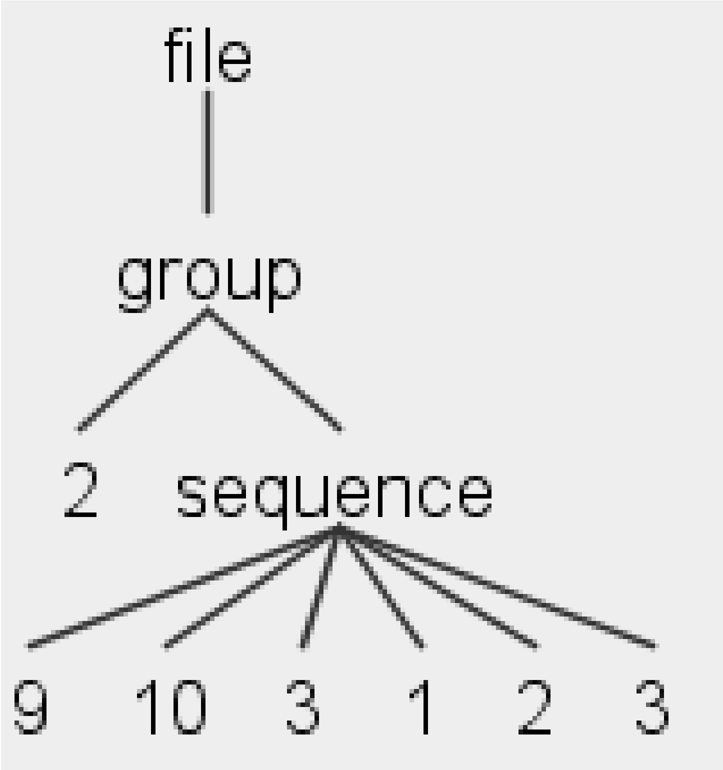
Will this do the job?

```
lexer grammar MyLexer;

INT  : [0-9]+ ;
WS  : [ \t\r\n]+ -> skip ;
```

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file: group+ ;

group: INT sequence ;

sequence: ( INT )* ;
```

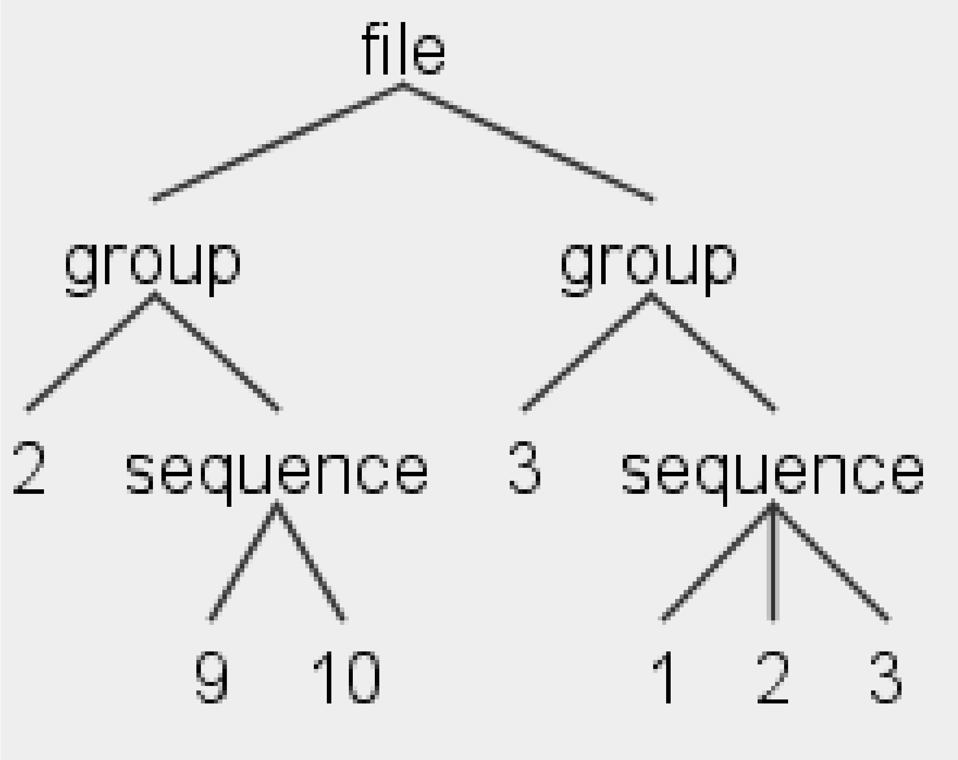
See java-examples/example38

Here's the parse tree that is generated:

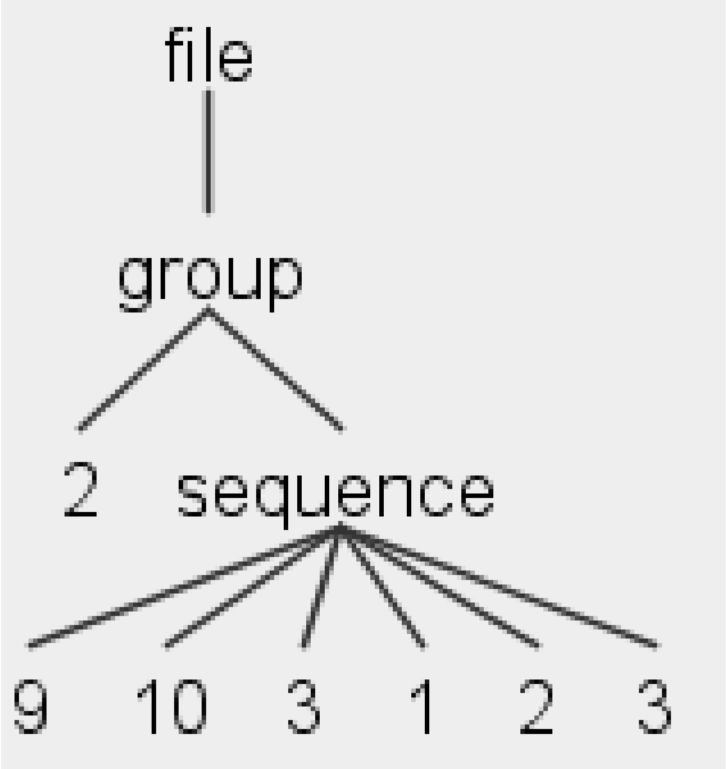


```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file: group+ ;

group: INT sequence ;

sequence: ( INT )* ;
```

No restriction on the number of INT values within sequence.



This is what we want



This is what we get

Runtime determination of # of occurrences

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file: group+ ;

group: INT sequence ;

sequence: ( INT )* ;
```

This value needs to dictate how many of these are allowed

Parser rule = function call

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file: group+ ;

group: INT sequence ;

sequence: ( INT )* ;
```

In the parser this is implemented as a function call to this

Invoking a function and passing arguments

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file: group+ ;

group: INT sequence[$INT.int] ;

sequence[int n] locals [int i = 1;] : ( {$i<=$n}? INT {$i++;} )* ;
```

Invoke the *sequence* function, pass to it the value of the INT token

Function parameter

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file: group+ ;

group: INT sequence[$INT.int] ;

sequence[int n] locals [int i = 1;] : ( {$i<=$n}? INT {$i++;} )* ;
```

The *sequence* function must be invoked with an integer, which is stored in *n*

Local variable declaration and initialization

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file: group+ ;

group: INT sequence[$INT.int] ;

sequence[int n] locals [int i = 1;] : ( {$i<=$n}? INT {$i++;} )* ;
```

Create a variable that is local to this rule and initialize its value to 1

Loop conditional

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file: group+ ;

group: INT sequence[$INT.int] ;

sequence[int n] locals [int i = 1;] : ( {$i<=$n}? INT {$i++;} )* ;
```

Repeat the loop (...)* as long as the local variable variable *i* has a value that does not exceed the value of the parameter *n*

Increment local variable

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file: group+ ;

group: INT sequence[$INT.int] ;

sequence[int n] locals [int i = 1;] : ( {$i<=$n}? INT {$i++;} )* ;
```

At the end of each iteration of the loop, increment the local variable variable *i* by 1.

Semantic predicate

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file: group+ ;

group: INT sequence[$INT.int] ;

sequence[int n] locals [int i = 1;] : ( {$i<=$n}? INT {$i++;} )* ;
```

This is a semantic predicate

Syntax of a semantic predicate

- A semantic predicate has this form:
  { *boolean expr* }?
- That is, a boolean expression within curly braces, followed by a question mark symbol
- A semantic predicate precedes an alternative and determines if the alternative should be taken: if the predicate evaluates to true, do the alternative; otherwise, don't do the alternative

Action

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file: group+ ;

group: INT sequence[$INT.int] ;

sequence[int n] locals [int i = 1;] : ( {$i<=$n}? INT {$i++;} )* ;
```

This is an action

Syntax of an action

- An action has this form:
  { *statements* }
- That is, one or more statements within curly braces
- An action follows an alternative

Match on *n* integers

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file: group+ ;

group: INT sequence[$INT.int] ;

sequence[int n] locals [int i = 1;] : ( {$i<=$n}? INT {$i++;} )* ;
```

This loop will match on *n* integers

See java-examples/example39

Doesn't work when the target is Python

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file: group+ ;

group: INT sequence[$INT.int] ;

sequence[int n] locals [int i = 1;] : ( {$i<=$n}? INT {$i++;} )* ;
```

This error is generated:

error(134): MyParser.g4:5:0: symbol file conflicts with generated code in target language or runtime

See: python-examples/example03

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file: group+ ;

group: INT sequence[$INT.int] ;

sequence[int n] locals [int i = 1;] : ( {$i<=$n}? INT {$i++;} )* ;
```

error(134): MyParser.g4:5:0: symbol file conflicts with generated code in target language or runtime

Change rule name

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

intfile: group+ ;

group: INT sequence[$INT.int] ;

sequence[int n] locals [int i = 1;] : ( {$i<=$n}? INT {$i++;} )* ;
```

See: python-examples/example04

Remove semi-colons

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

intfile: group+ ;

group: INT sequence[$INT.int] ;

sequence[int n] locals [int i = 1;] : ( {$i<=$n}? INT {$i++;} )* ;
```

Python does not end statements with a semi-colon, so remove the semi-colons.

Don't use ++ operator

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

intfile: group+ ;

group: INT sequence[$INT.int] ;

sequence[int n] locals [int i = 1] : ( {$i<=$n}? INT {$i++} )* ;
```

Python does not have a ++ operator.
So change to: $i = $i + 1

This works

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

intfile: group+ ;

group: INT sequence[$INT.int] ;

sequence[int n] locals [int i = 1] : ( {$i<=$n}? INT {$i = $i + 1} )* ;
```

See python-examples/example04

Actions and semantic predicates are Java/Python code

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

intfile: group+ ;

group: INT sequence[$INT.int] ;

sequence[int n] locals [int i = 1] : ( {$i<=$n}? INT {$i = $i + 1} )* ;
```

Caution: by using actions and/or sematic predicates, you are tying your grammar to a specific target language.

Acknowlegement

The following examples were created by James Garriss – thanks James!

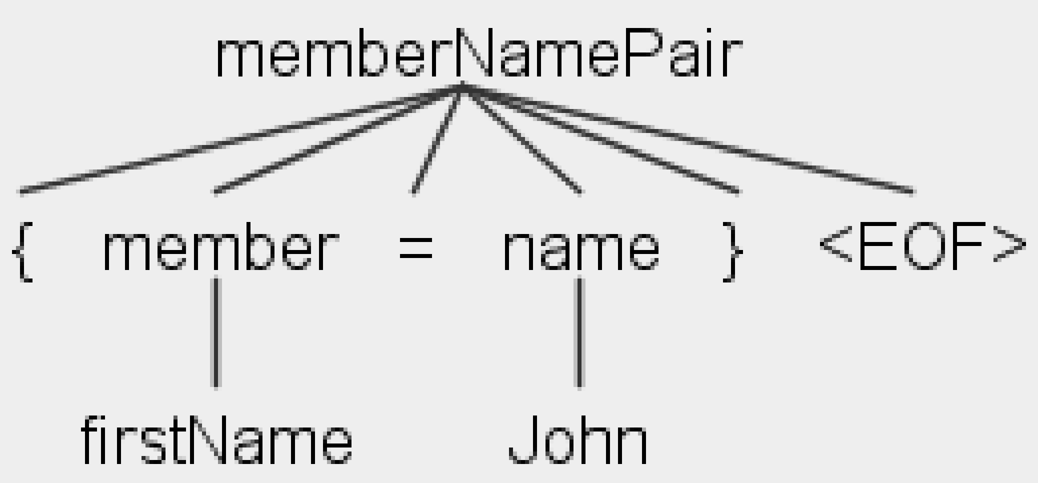
Problem #2

Lexer processes token

- Problem: design a token rule which removes quotes that wrap strings.
- Example: if the input is "John" (with quotes) then the token rule should send to the parser the token value John. If the input is John (no quotes) then the token rule should send to the parser the token value John.
- The solution is to remove the quotes (if present) while lexing the input using the getText/setText methods (Java) or self.text (Python).

Here's the parse tree we want

Input:
{ firstName = "John" }



```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

memberNamePair : LCurly member EQ name RCurly EOF ;

member : UnquotedString ;

name : (UnquotedString | QuotedString) ;
```

See java-examples/example40

```
lexer grammar MyLexer;

LCurly : '{' ;
RCurly : '}' ;
EQ : '=' ;

UnquotedString : [a-zA-Z]+ ;

QuotedString : '"' UnquotedString '"'
      {
       String theString = getText();
       int firstLetter = 1;
       int lastLetter = theString.length() - 1;
       String theNewString = theString.substring(firstLetter, lastLetter);
       setText(theNewString);
      } ;

WS  : [ \t\r\n]+ -> skip ;
```

```
lexer grammar MyLexer;

LCurly : '{' ;
RCurly : '}' ;
EQ : '=' ;

UnquotedString : [a-zA-Z]+ ;

QuotedString : '"' UnquotedString '"'
      {
       String theString = getText();
       int firstLetter = 1;
       int lastLetter = theString.length() - 1;
       String theNewString = theString.substring(firstLetter, lastLetter);
       setText(theNewString);
      } ;

WS  : [ \t\r\n]+ -> skip ;
```

An action in the token rule (Java code)

```
lexer grammar MyLexer;

LCurly : '{' ;
RCurly : '}' ;
EQ : '=' ;

UnquotedString : [a-zA-Z]+ ;

QuotedString : '"' UnquotedString '"'
      {
       String theString = getText();
       int firstLetter = 1;
       int lastLetter = theString.length() - 1;
       String theNewString = theString.substring(firstLetter, lastLetter);
       setText(theNewString);
      } ;

WS  : [ \t\r\n]+ -> skip ;
```

This gets "John"

```
lexer grammar MyLexer;

LCurly : '{' ;
RCurly : '}' ;
EQ : '=' ;

UnquotedString : [a-zA-Z]+ ;

QuotedString : '"' UnquotedString '"'
      {
       String theString = getText();
       int firstLetter = 1;
       int lastLetter = theString.length() - 1;
       String theNewString = theString.substring(firstLetter, lastLetter);
       setText(theNewString);
      } ;

WS  : [ \t\r\n]+ -> skip ;
```

Extract the text within the quotes

```
lexer grammar MyLexer;

LCurly : '{' ;
RCurly : '}' ;
EQ : '=' ;

UnquotedString : [a-zA-Z]+ ;

QuotedString : '"' UnquotedString '"'
       {
        String theString = getText();
        int firstLetter = 1;
        int lastLetter = theString.length() - 1;
        String theNewString = theString.substring(firstLetter, lastLetter);
        setText(theNewString);
       } ;

WS  : [ \t\r\n]+ -> skip ;
```

Set the token value to be sent to the parser

Python version

```
lexer grammar MyLexer;

LCurly : '{' ;
RCurly : '}' ;
EQ : '=' ;

UnquotedString : [a-zA-Z]+ ;

QuotedString : '"' UnquotedString '"'
       {
        String theString = getText();
        int firstLetter = 1;
        int lastLetter = theString.length() - 1;
        String theNewString = theString.substring(firstLetter, lastLetter);
        setText(theNewString);
       } ;

WS  : [ \t\r\n]+ -> skip ;
```

delete types

replace

delete semi-colons

replace

use Python splice

Here's the Python lexer

```
lexer grammar MyLexer;

LCurly : '{' ;
RCurly : '}' ;
EQ : '=' ;

UnquotedString : [a-zA-Z]+ ;

QuotedString : '"' UnquotedString '"'
      {
       theString = self.text
       firstLetter = 1
       lastLetter = len(theString) - 1
       theNewString = theString[firstLetter:lastLetter]
       self.text = theNewString
      } ;

WS  : [ \t\r\n]+ -> skip ;
```

See python-examples/example05 (parser is the same as with the Java version)

Do Lab20, 21, 22

Problem #3

Parser rule checks line length

- Problem: design a parser for a line of text; the parser outputs an error if the line length exceeds 20 characters.
- Example: if the input is this:

  It will snow for hours and hours and hours.

  then there should be an output saying it is too long.

This is insufficient

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file : (line)+ EOF ;

line : STRING NL ;
```

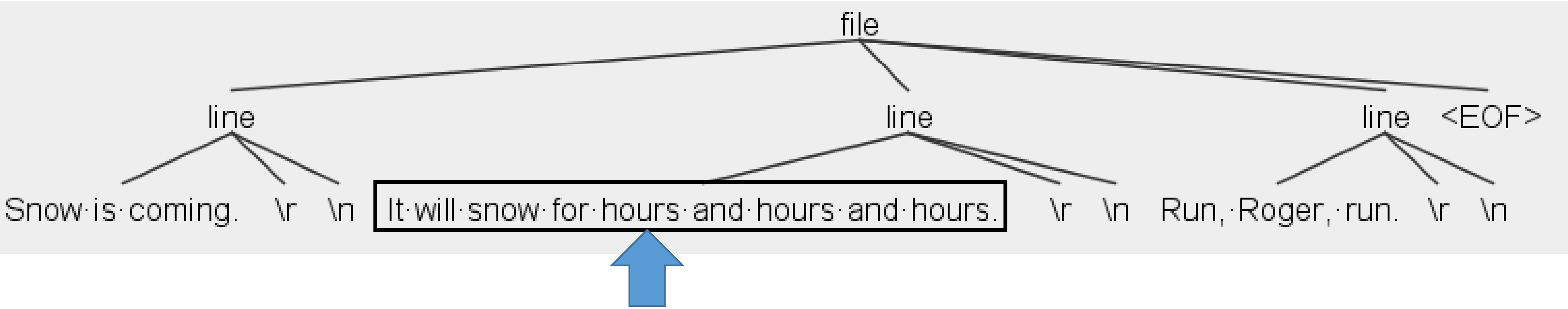
```
lexer grammar MyLexer;

STRING : ('A'..'Z' | 'a'..'z' | ' ' | '.' | ',')+ ;

NL  : '\r' '\n' ;
```

See java-examples/example41

No errors

<u>Input</u>:

Snow is coming.
It will snow for hours and hours and hours.
Run, Roger, run.



want an error with this

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file : (line
          {
              if ($line.lineLength > 20)
              {
                  System.out.println("Line is too long: " + $line.lineLength + " characters!");
              }
          }
         )+ EOF
    ;

line returns [int lineLength]
    : s=A_STRING
      {
        String line = $s.text;
        int lineLength = line.length();
        $lineLength = lineLength;
      }
      NL
    ;
```

See java-examples/example42

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file : (line
         {
             if ($line.lineLength > 20)
             {
                 System.out.println("Line is too long: " + $line.lineLength + " characters!");
             }
         }
        )+ EOF
    ;

line returns [int lineLength]
    : s=A_STRING
      {
        String line = $s.text;
        int lineLength = line.length();
        $lineLength = lineLength;
      }
      NL
    ;
```

This rule will return an int. The value returned is named lineLength.

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file : (line
          {
              if ($line.lineLength > 20)
              {
                  System.out.println("Line is too long: " + $line.lineLength + " characters!");
              }
          }
         )+ EOF
     ;

line returns [int lineLength]
     : s=A_STRING
       {
         String line = $s.text;
         int lineLength = line.length();
         $lineLength = lineLength;
       }
       NL
     ;
```

s is a token label (i.e., an alias for the `A_STRING` token type)

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file : (line
          {
              if ($line.lineLength > 20)
              {
                  System.out.println("Line is too long: " + $line.lineLength + " characters!");
              }
          }
         )+ EOF
     ;

line returns [int lineLength]
     : s=A_STRING
       {
        String line = $s.text;
        int lineLength = line.length();
        $lineLength = lineLength;
       }
       NL
     ;
```

An action (Java code)

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file : (line
          {
              if ($line.lineLength > 20)
              {
                  System.out.println("Line is too long: " + $line.lineLength + " characters!");
              }
          }
         )+ EOF
     ;

line returns [int lineLength]
     : s=A_STRING
       {
         String line = $s.text;
         int lineLength = line.length();
         $lineLength = lineLength;
       }
       NL
     ;
```

Get the value (text) of the token.

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file : (line
          {
              if ($line.lineLength > 20)
              {
                  System.out.println("Line is too long: " + $line.lineLength + " characters!");
              }
          }
         )+ EOF
    ;

line returns [int lineLength]
    : s=A_STRING
      {
        String line = $s.text;
        int lineLength = line.length();
        $lineLength = lineLength;
      }
      NL
    ;
```

Get the length of line

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file : (line
          {
              if ($line.lineLength > 20)
              {
                  System.out.println("Line is too long: " + $line.lineLength + " characters!");
              }
          }
         )+ EOF
    ;

line returns [int lineLength]
    : s=A_STRING
      {
        String line = $s.text;
        int lineLength = line.length();
        $lineLength = lineLength;
      }
      NL
    ;
```

Set the return value to the line length

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file : (line
          {
              if ($line.lineLength > 20)
              {
                  System.out.println("Line is too long: " + $line.lineLength + " characters!");
              }
          }
         )+ EOF
     ;

line returns [int lineLength]
     : s=A_STRING
       {
         String line = $s.text;
         int lineLength = line.length();
         $lineLength = lineLength;
       }
       NL
     ;
```

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file : (line
          {
              if ($line.lineLength > 20)
              {
                  System.out.println("Line is too long: " + $line.lineLength + " characters!");
              }
          }
         )+ EOF
     ;

line returns [int lineLength]
     : s=A_STRING
       {
         String line = $s.text;
         int lineLength = line.length();
         $lineLength = lineLength;
       }
       NL
     ;
```

An action (Java code).

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file : (line
          {
              if ($line.lineLength > 20)
              {
                  System.out.println("Line is too long: " + $line.lineLength + " characters!");
              }
          }
         )+ EOF
     ;

line returns [int lineLength]
     : s=A_STRING
       {
         String line = $s.text;
         int lineLength = line.length();
         $lineLength = lineLength;
       }
       NL
     ;
```

This is the variable returned from the line rule.

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file : (line
         {
            if ($line.lineLength > 20)
            {
                System.out.println("Line is too long: " + $line.lineLength + " characters!");
            }
         }
        )+ EOF
     ;

line returns [int lineLength]
     : s=A_STRING
       {
         String line = $s.text;
         int lineLength = line.length();
         $lineLength = lineLength;
       }
       NL
     ;
```

Is the value of lineLength, returned from the line rule greater than 20?

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file : (line
         {
             if ($line.lineLength > 20)
             {
                 System.out.println("Line is too long: " + $line.lineLength + " characters!");
             }
         }
        )+ EOF
    ;

line returns [int lineLength]
    : s=A_STRING
      {
        String line = $s.text;
        int lineLength = line.length();
        $lineLength = lineLength;
      }
      NL
    ;
```

Output a message that the line is too long.

Errors!

<u>Input</u>:

Snow is coming.
It will snow for hours and hours and hours.
Run, Roger, run.

Line is too long: 43 characters!

Python version

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file : (line
          {
          if ($line.lineLength > 20)
          {
             System.out.println("Line is too long: " + $line.lineLength + " characters!");
          }
          }
        )+ EOF
    ;

line returns [int lineLength]
    : s=A_STRING
      {
        lineLength = len($s.text);
        $lineLength = lineLength;
      }
      NL
    ;
```

change rule name

add a colon here

delete curly braces

replace

delete semi-colons

Changes (cont.)

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file : (line
          {
if ($line.lineLength > 20)
{
System.out.println("Line is too long: " + $line.lineLength + " characters!");
}
          }
         )+ EOF
    ;

line returns [int lineLength]
    : s=A_STRING
      {
lineLength = len($s.text);
$lineLength = lineLength;
      }
      NL
    ;
```

Indentation matters in Python. If you leave this indentation, you will get an "unexpected indentation" error message. Remove the indentation.

Here's the Python version

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

afile : (line
         {
if len($line.text) > 20:
    print("Line is too long: " + str(len($line.text)) + " characters!")
         }
        )+ EOF
    ;

line returns [lineLength]
    : s=A_STRING
      {
lineLength = len($s.text)
$lineLength = lineLength
      }
      NL
    ;
```

See python-examples/example06 (lexer is the same as with the Java version)

Note: ANTLR curly braces indicate "action"

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

afile : (line
          {
if len($line.text) > 20:
    print("Line is too long: " + str(len($line.text)) + " characters!")
          }
         )+ EOF
    ;

line returns [lineLength]
    : s=A_STRING
      {
lineLength = len($s.text)
$lineLength = lineLength
      }
      NL
    ;
```

Problem #4

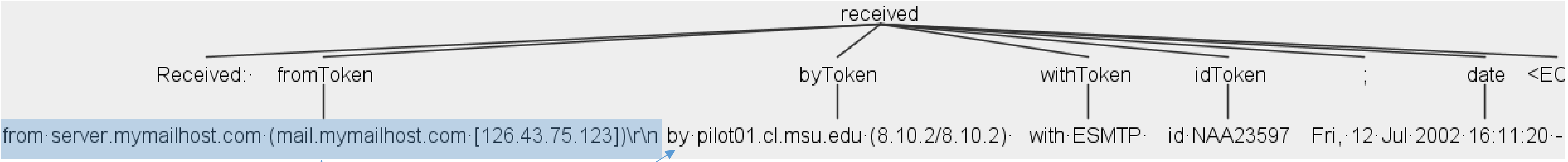
Gobble up characters till next reserved word

Problem: design a parser for email headers; gobble up all the characters between the keywords.

gobble up the characters between the **Received:** and **by** keywords

Received: from server.mymailhost.com (mail.mymailhost.com [126.43.75.123])
by pilot01.cl.msu.edu (8.10.2/8.10.2) **with** ESMTP id NAA23597;
Fri, 12 Jul 2002 16:11:20 -0400 (EDT)

Desired parse tree



gobbled up all the text prior to the **by** keyword

```
lexer grammar MyLexer;

Received : 'Received: ' ;
SemiColon : ';' ;

FromText
    : 'from '
      .+?
      {
       (_input.LA(1) == 'b') &&
       (_input.LA(2) == 'y')
      }?
    ;

ByText
    : 'by '
      .+?
      {
       (_input.LA(1) == 'w') &&
       (_input.LA(2) == 'i') &&
       (_input.LA(3) == 't') &&
       (_input.LA(4) == 'h')
      }?
    ;
```

continued →

```
WithText
    : 'with '
      .+?
      {
       (_input.LA(1) == 'i') &&
       (_input.LA(2) == 'd')
      }?
    ;

IdText
    : 'id '
      .+?
      {
       (_input.LA(1) == ';')
      }?
    ;

DateContents : ('Mon' | 'Tue' | 'Wed' | 'Thu' | 'Fri' | 'Sat' | 'Sun') (Letter | Number | Special)+ ;

fragment Letter :  'A'..'Z' | 'a'..'z' ;

fragment Number : '0'..'9' ;

fragment Special : ' ' | '_' | '-' | '.' | ',' | '~' | ':' | '+' | '$' | '=' | '(' | ')' | '[' | ']' |
'/' ;

Whitespace : [\t\r\n]+ -> skip ;
```

```
lexer grammar MyLexer;


Received : 'Received: ' ;
SemiColon : ';' ;

FromText
    : 'from '
      .+?
      {
         (_input.LA(1) == 'b') &&
         (_input.LA(2) == 'y')
      }?
    ;

ByText
    : 'by '
      .+?
      {
       (_input.LA(1) == 'w') &&
       (_input.LA(2) == 'i') &&
       (_input.LA(3) == 't') &&
       (_input.LA(4) == 'h')
      }?
    ;
...
```

semantic predicate (as denoted by the question mark at the end)

```
lexer grammar MyLexer;


Received : 'Received: ' ;
SemiColon : ';' ;

FromText
    : 'from '
      .+?
      {
         (_input.LA(1) == 'b') &&
         (_input.LA(2) == 'y')
      }?
    ;

ByText
    : 'by '
      .+?
      {
       (_input.LA(1) == 'w') &&
       (_input.LA(2) == 'i') &&
       (_input.LA(3) == 't') &&
       (_input.LA(4) == 'h')
      }?
    ;
...
```

The lexer keeps failing this token rule until it gets to a character whose next character is 'b' and after that is 'y'

```
lexer grammar MyLexer;


Received : 'Received: ' ;
SemiColon : ';' ;

FromText
    : 'from '
      .+?
      {
         (_input.LA(1) == 'b') &&
         (_input.LA(2) == 'y')
      }?
    ;

ByText
    : 'by '
      .+?
      {
       (_input.LA(1) == 'w') &&
       (_input.LA(2) == 'i') &&
       (_input.LA(3) == 't') &&
       (_input.LA(4) == 'h')
      }?
    ;
...
```

LA = Look Ahead
LA(1) means look ahead 1 character
LA(2) means look ahead 2 characters
...
LA(k) means look ahead k characters

Parser grammar

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

received : Received fromToken byToken withToken idToken SemiColon date EOF ;

fromToken : FromText ;

byToken: ByText ;

withToken : WithText ;

idToken : IdText ;

date : DateContents+ ;
```

See java-examples/example43

Info on Look Ahead (LA)

http://www.antlr.org/api/Java/index.html?org/antlr/v4/runtime/IntStream.html

Python version

```
lexer grammar MyLexer;

Received : 'Received: ' ;
SemiColon : ';' ;

FromText : 'from ' .+?
         {(self._input.LA(1) == ord('b')) and (self._input.LA(2) == ord('y'))}?
         ;
...
```

See python-examples/example7

This is quite different from the Java version:

```
(_input.LA(1) == 'b') &&(_input.LA(2) == 'y')
```

Writing applications to process parse trees

Language recognition

- Sometimes we simply want to know: Is the input a member of the language?
- That is, we are just interested in language recognition.
- In such cases the following slides are not relevant.

When we do regex matching we are doing language recognition.

Language processing

- Other times we want to build a parse tree and then have applications perform processing on the parse tree.
- The following slides show how to write an application to process the parser generated by ANTLR.

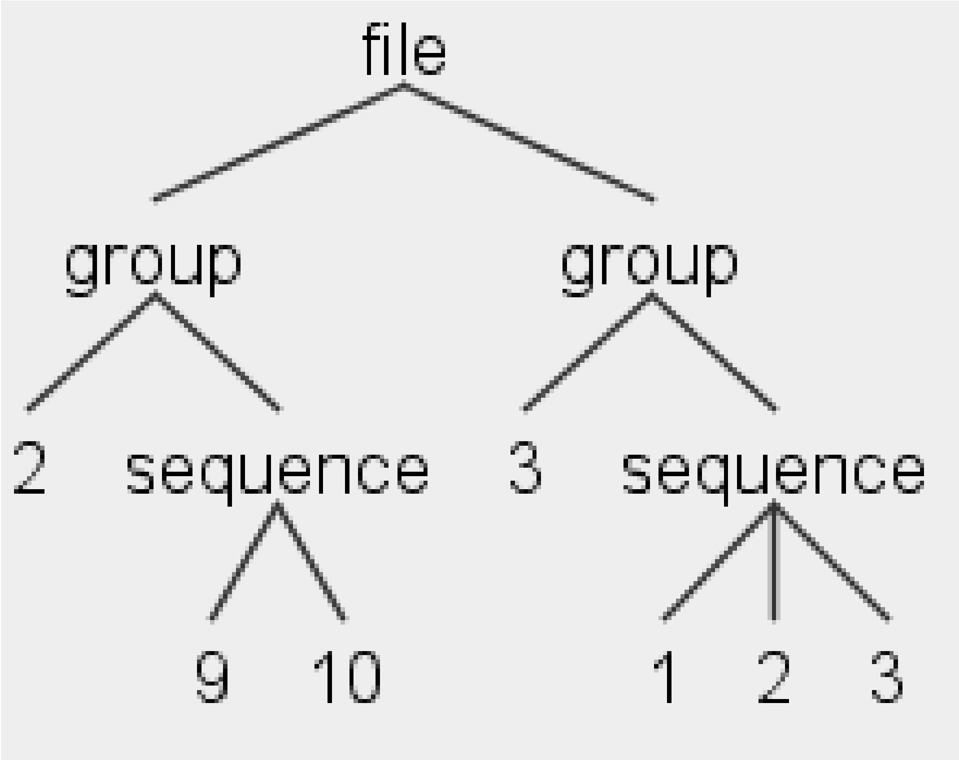
Recall this problem

Create a grammar for this input:

2 9 10 3 1 2 3

indicates that there are 2 following integers	indicates that there are 3 following integers

Here is the parse tree we desired:



Here is the parser grammar we created

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file: group+ ;

group: INT sequence[$INT.int] ;

sequence[int n] locals [int i = 1;] : ( {$i<=$n}? INT {$i++;} )* ;
```

Java code embedded in the grammar

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file: group+ ;

group: INT sequence[$INT.int] ;

sequence[int n] locals [int i = 1;] : ( {$i<=$n}? INT {$i++;} )* ;
```

This is Java code. It destroys the clarity and simplicity of the grammar. Plus the resulting parser can only be used by Java applications. These are good reasons why you should <u>not</u> do this.

2 9 10 3 1 2 3

indicates that there are 2 following integers

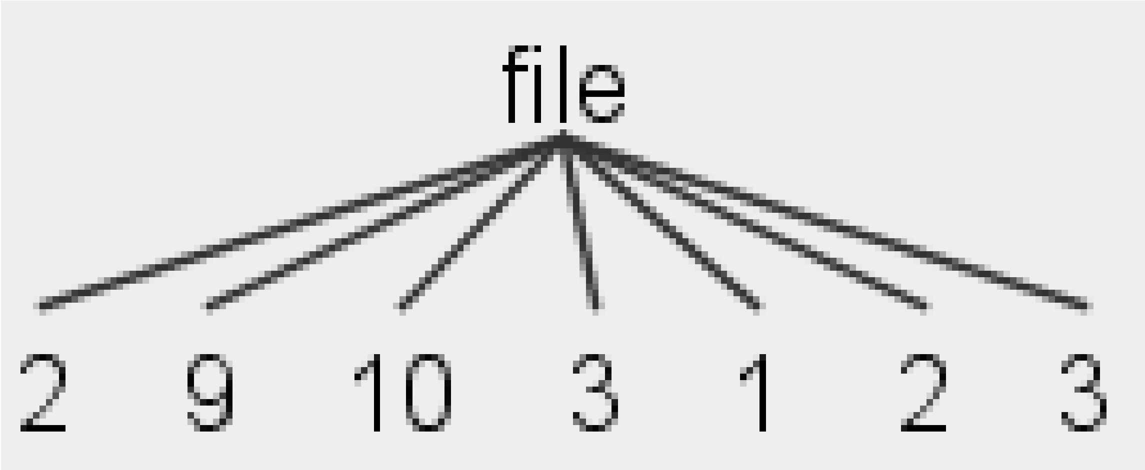
indicates that there are 3 following integers

Let application code do these checks.

2 9 10 3 1 2 3

The grammar just checks for a series of integers.

Create a grammar to produce this parse tree



Here are the lexer and parser grammars

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

file: (INT)+ ;
```

```
lexer grammar MyLexer;

INT  : [0-9]+ ;
WS   : [ \t\r\n]+ -> skip ;
```

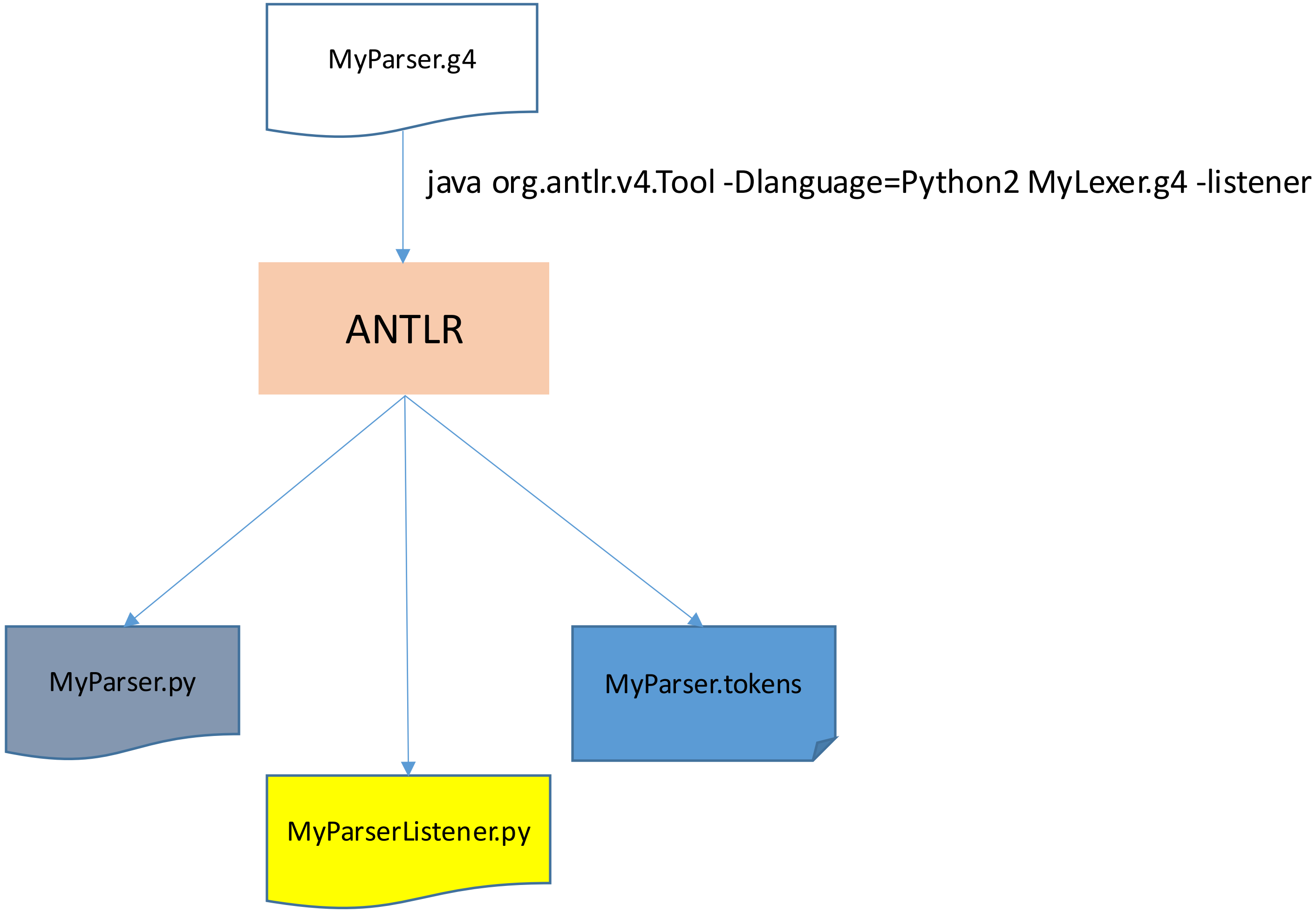
See java-examples/example44 and python-examples/example08

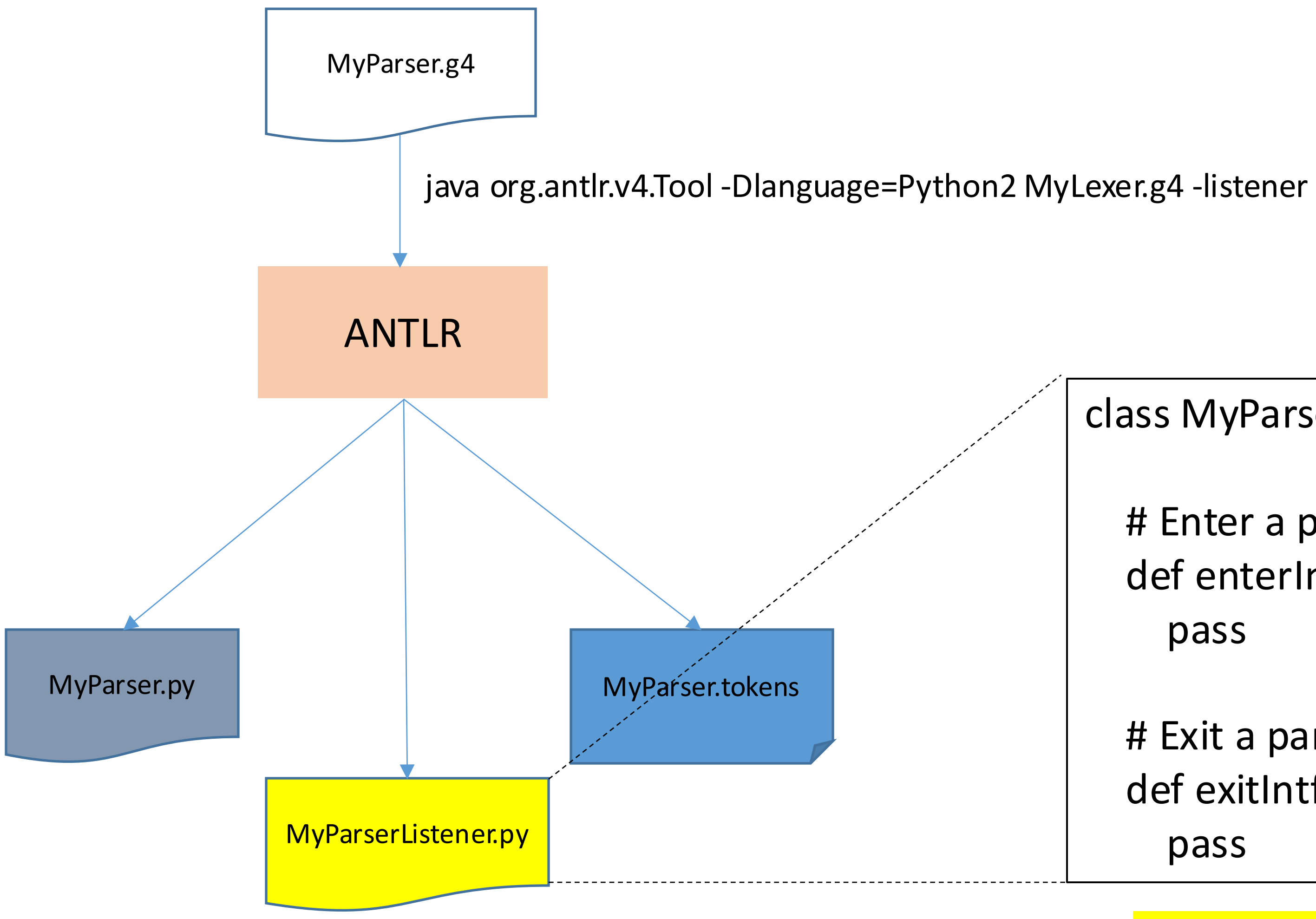
Generate a Listener

When you run ANTLR on your grammar, you can tell ANTLR: "*Please generate a listener*."

```
java org.antlr.v4.Tool -Dlanguage=Python2 MyLexer.g4 -listener
```

flag tells ANTLR that you want it to generate a listener


MyParser.g4
java org.antlr.v4.Tool -Dlanguage=Python2 MyLexer.g4 -listener
ANTLR
MyParser.py
MyParserListener.py
MyParser.tokens




```
class MyParserListener(ParseTreeListener):

    # Enter a parse tree produced by MyParser#intfile.
    def enterIntfile(self, ctx):
        pass

    # Exit a parse tree produced by MyParser#intfile.
    def exitIntfile(self, ctx):
        pass
```

Actually, ANTLR generates a skeleton listener.

MyParser.g4

```
parser grammar MyParser;

options { tokenVocab=MyLexer; }

intfile: (INT)+ ;
```

MyParserListener.py

```
class MyParserListener(ParseTreeListener):

    # Enter a parse tree produced by MyParser#intfile.
    def enterIntfile(self, ctx):
        pass

    # Exit a parse tree produced by MyParser#intfile.
    def exitIntfile(self, ctx):
        pass
```

This function gets called when the intfile rule is entered

This function gets called when the intfile rule is exited

Create a file to replace the skeletal listener

MyParserListener.py

```
class MyParserListener(ParseTreeListener):

    # Enter a parse tree produced by MyParser#intfile.
    def enterIntfile(self, ctx):
        pass

    # Exit a parse tree produced by MyParser#intfile.
    def exitIntfile(self, ctx):
        pass
```

rewriter.py

```
class RewriteListener(MyParserListener):
    # Enter a parse tree produced by MyParserParser#intfile.
    def enterIntfile(self, ctx):
        print("Entering intfile")

    # Exit a parse tree produced by MyParserParser#intfile.
    def exitIntfile(self, ctx):
        print("Exiting intfile")
```

main.py

```
import sys
from antlr4 import *
from MyLexer import MyLexer
from MyParser import MyParser
from rewriter import RewriteListener
import stdio

def main(argv):
    istream = FileStream(argv[1])
    lexer = MyLexer(istream)
    stream = CommonTokenStream(lexer)
    parser = MyParser(stream)
    tree = parser.intfile()
    print(tree.toStringTree(recog=parser))

    walker = ParseTreeWalker()
    walker.walk(RewriteListener(), tree)
    print("Done")

if __name__ == '__main__':
    main(sys.argv)
```

main.py

```
import sys
from antlr4 import *
from MyLexer import MyLexer
from MyParser import MyParser
from rewriter import RewriteListener
import stdio

def main(argv):
    istream = FileStream(argv[1])
    lexer = MyLexer(istream)
    stream = CommonTokenStream(lexer)
    parser = MyParser(stream)
    tree = parser.intfile()
    print(tree.toStringTree(recog=parser))

    walker = ParseTreeWalker()
    walker.walk(RewriteListener(), tree)
    print("Done")

if __name__ == '__main__':
    main(sys.argv)
```

name of the start rule

main.py

```
import sys
from antlr4 import *
from MyLexer import MyLexer
from MyParser import MyParser
from rewriter import RewriteListener
import stdio

def main(argv):
    istream = FileStream(argv[1])
    lexer = MyLexer(istream)
    stream = CommonTokenStream(lexer)
    parser = MyParser(stream)
    tree = parser.intfile()
    print(tree.toStringTree(recog=parser))

    walker = ParseTreeWalker()
    walker.walk(RewriteListener(), tree)
    print("Done")

if __name__ == '__main__':
    main(sys.argv)
```

python

```
(intfile 2 9 10 3 1 2 3)
Entering intfile
Exiting intfile
Done
```

The End