innovate achieve lead

Lexical Analysis

Shashank Gupta
Assistant Professor
Department of Computer Science and Information Systems

BITS Pilani
Pilani Campus

Process of Tokenization in Transition Diagram

The matching process should always start with some transition diagram.

If failure occurs in one transition diagram.

- Retract the forward pointer to the start state.
- Activate the next transition diagram.

If failure occurs in all transition diagrams then throw the Lexical Error.

Generation of Lexical Analyzer from Transition Diagram

The more complex is your transition diagram

- More complex would be the equivalent source code.
- Since, each state will need to handle more decisions.

During implementation, Complex transition diagrams may give rise to errors.

Example

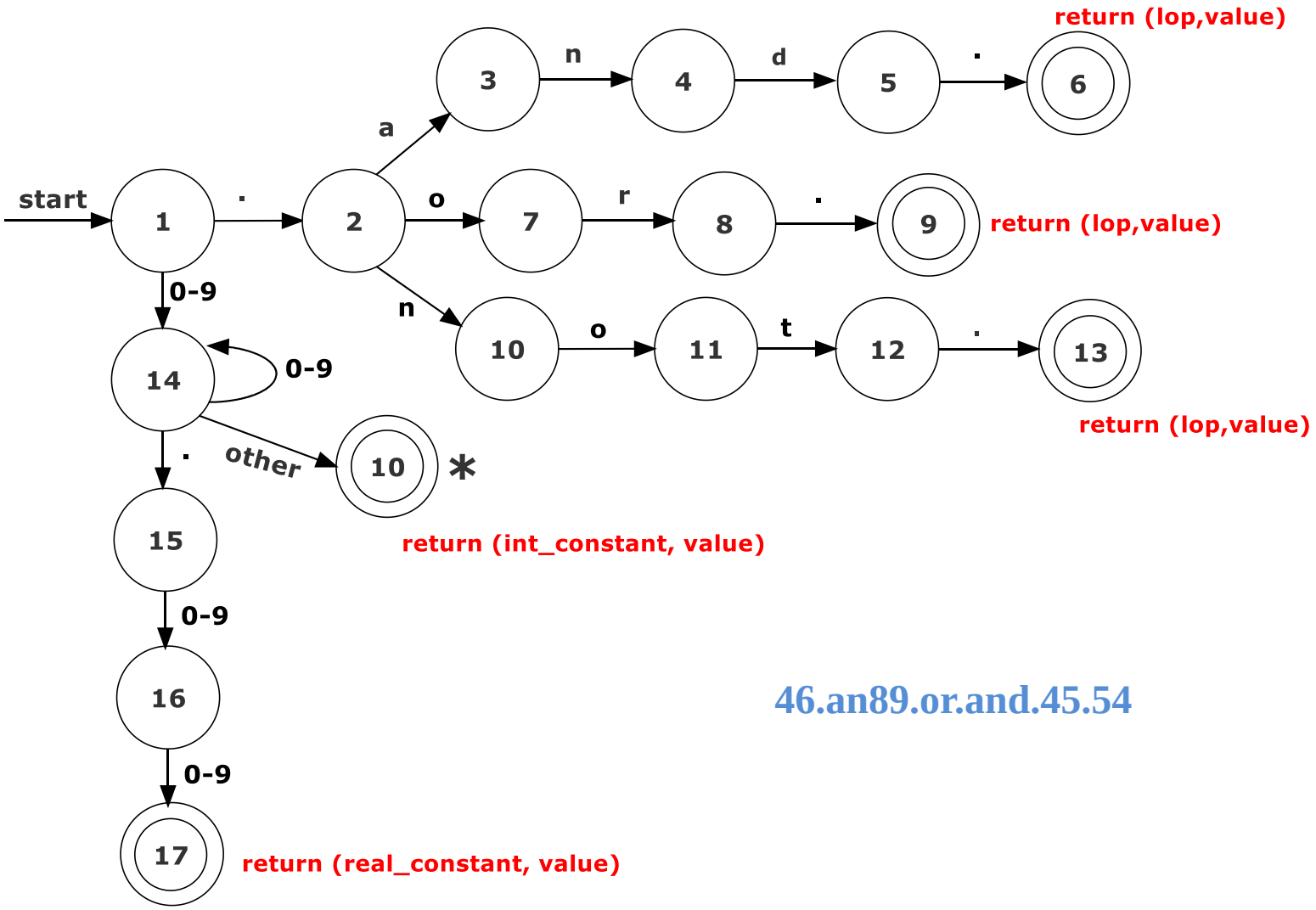
Construct a generalized transition diagram to capture the patterns of logical operators, integers and real numbers. Regular expressions [.][a][n][d][.], [.] [o][r][.], [.][n][o][t][.], [0-9][0-9]* and [0-9][0-9]*[.][0-9][0-9] respectively define patterns for logical and, or, not and integer and real numbers.

How will you tokenize the following i/p?

46.an89.or.and.45.54

Explain the step by step procedure precisely.

Transition Diagram



46.an89.or.and.45.54

Explanation for Tokenization in Generalized Transition Diagram

1. On reading 49 from state 1 to 14, analyzer will also read . and ‘a’ on state 15 and throw extra characters back to input stream and will also return <num, 49> .
2. In addition, the forward pointer will go back to state 1 and will start tracing ‘.’ ‘a’ ‘n’. As soon as, it will read ‘8 on state 4 it will simply throw lexical error as ‘.an’ and will also throw ‘8’ back into the input stream along with the forward pointer back to state 1.
3. Similarly ’89’ will also be recognized similar to step 1. In addition, .or. will be recognized as logical operator.

Explanation for Tokenization in Generalized Transition Diagram

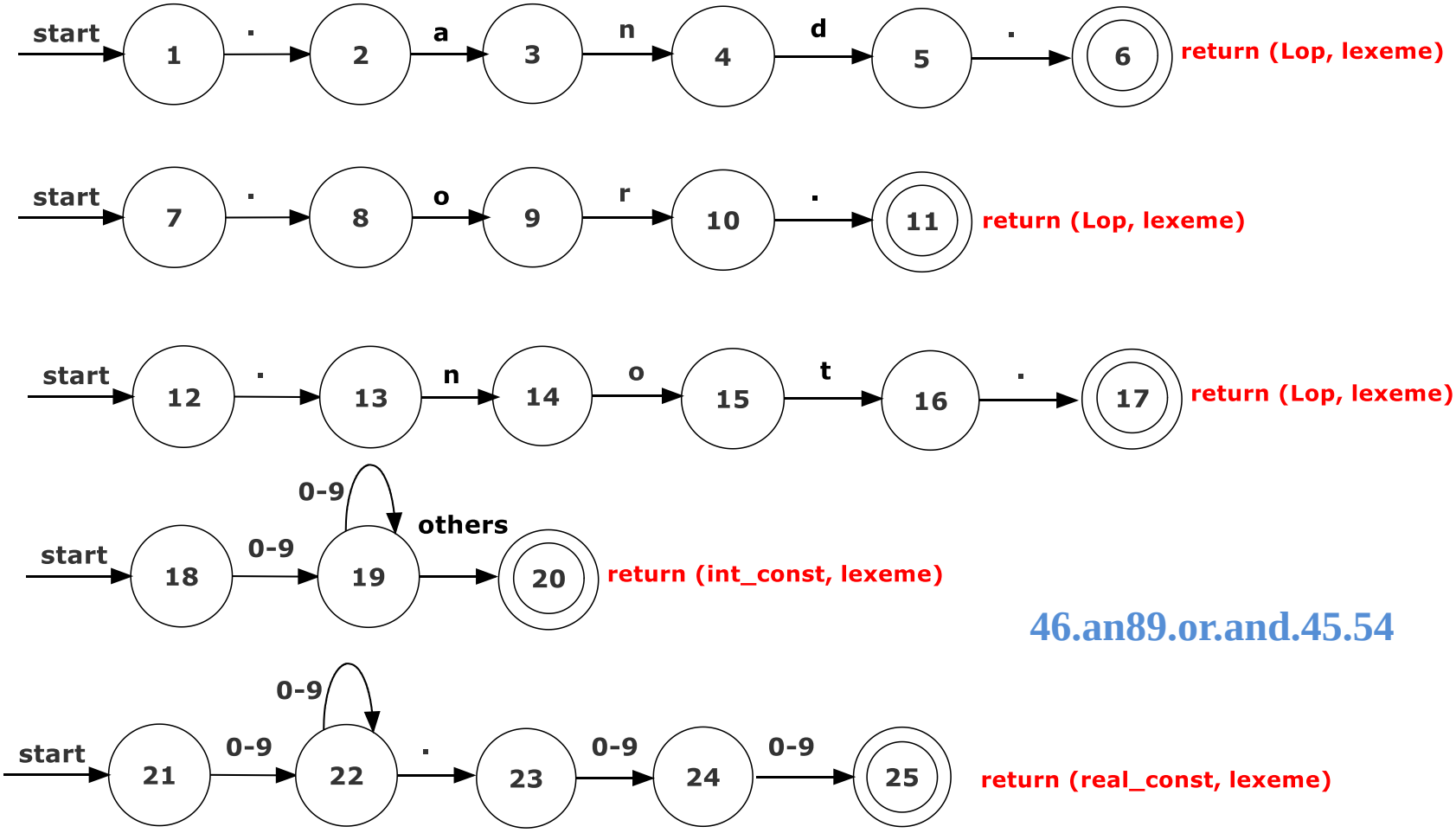
4. Once .or. will be recognized as a logical operator, the forward point will again shift back to state 1. From here it will read ‘a’ on some other state (possibly dead state but not state 2 or 14) and will recognize ‘a’ as a lexical error.

5. Similarly, it will shift forward pointer back to state 1 from dead state and will read ‘n’ on dead state and recognize ‘n’ as a lexical error. Similarly, ‘d’ will also be recognized as a lexical error.

6. Now by reading the ‘.’ analyzer will shift its forward pointer from state 1 to state 2 and again go to some dead state by reading ‘4’. Here, the analyzer will simple return ‘.’ also as lexical error and will also throw ‘4’ back into the input stream and shift its forward pointer from dead state to initial state.

7. Finally, 45.54 will simply tokenized as real number.

Multiple Transition Diagrams



46.an89.or.and.45.54

Generation of Lexical Analyzer from Transition Diagram

Different transition diagrams must be combined appropriately to generate a lexical analyzer.

- Merging different transition diagrams is not so easy.

Trace different transition diagrams one after another.

To find the longest match, all transition diagrams must be tried and the longest match must be used.

innovate achieve lead

Syntax Analysis

BITS Pilani
Pilani Campus

Dr. Shashank Gupta
Assistant Professor
Department of Computer Science and Information Systems

Agenda

- Challenges in Implementation of Syntax Analysis

Syntax Analysis

Check syntax of the extracted tokens and construct the parse tree.

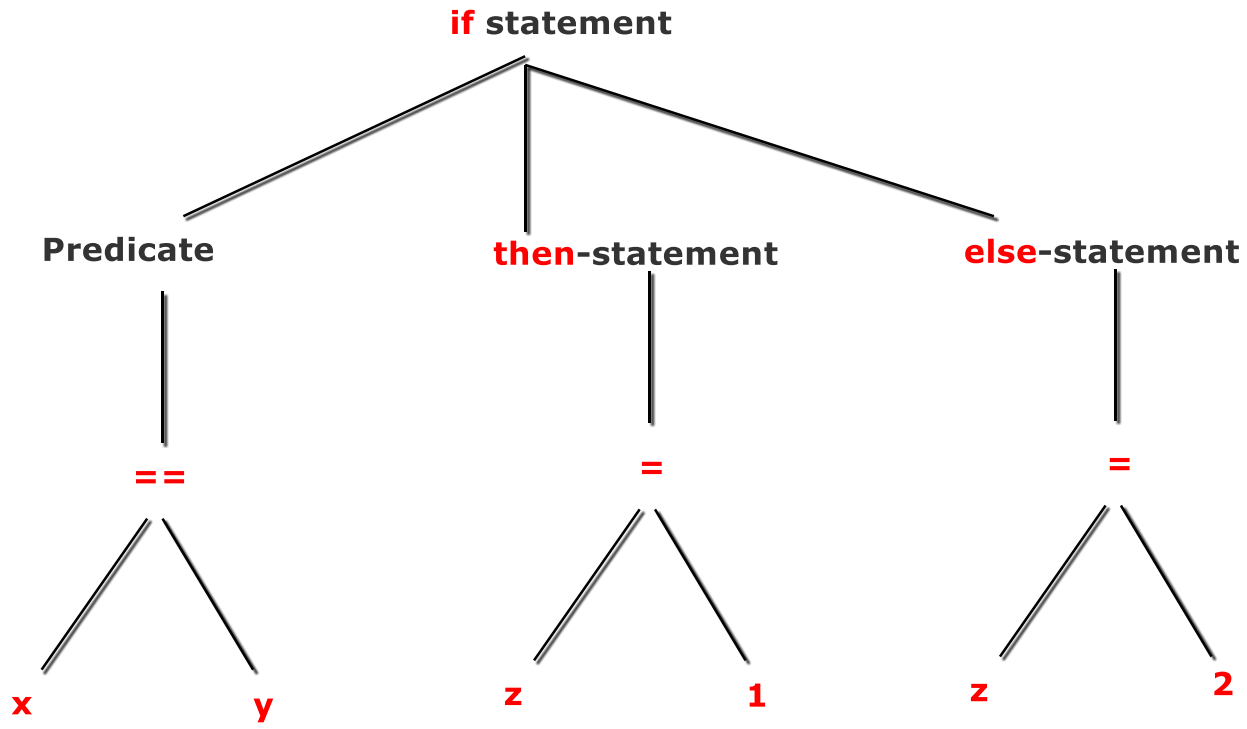
Error reporting and error recovery must be done.

Model using Context-free Grammars that will be recognized using Pushdown Automata/Table driven parsers.

Parsing

- Consider the list of tokens

  if x == y then z = 1 else z = 2



DONT'S for Syntax Analysis

Parsers cannot handle the contextual information.

- Determining the type of variables and which operations are allowed on that type.
- Initialization/Type Checking of Variables.

Hence, modelling of the parsers requires Context-Free Grammars.

Motivation for Syntax Analysis

Regular Definitions alone are not sufficient.

Many languages are not regular.

A Transition Diagram may visit states in a repetitive order, however, it cannot remember the number of times it has been to a particular state.

Hence, a more powerful language is required to recognize a valid structure of tokens.

Context-Free Grammars

Context Free Grammars **<T, N, P, S>**

- a set of tokens (terminal symbols)
- a set of non terminal symbols
- a set of productions
- a Start symbol

A grammar derives strings by beginning with a start symbol and repeatedly replacing a non terminal by the right hand side of a production for that non terminal.

The strings that can be derived from the start symbol of a grammar G form the language **L(G)** defined by the grammar.

Examples

- String of balanced parentheses

  $S \rightarrow (S) \mid \in$

- Grammar

  $List \rightarrow List + Digit$

  $\mid List - Digit$

  $\mid Digit$

  $Digit \rightarrow 0 \mid 1 \mid 2 \mid 3 - — \mid 9$

Process of Derivation

$$List \rightarrow \underline{List} + Digit$$
$$\rightarrow \underline{List} - Digit + Digit$$
$$\rightarrow \underline{Digit} - Digit + Digit$$
$$\rightarrow 9 - \underline{Digit} - Digit$$
$$\rightarrow 9 - 5 + \underline{Digit}$$
$$\rightarrow 9 - 5 + 2$$

String to be derived: 9 – 5 + 2

$$List \rightarrow List + Digit \mid List - Digit \mid Digit$$
$$Digit \rightarrow 0 \mid 1 \mid 2 \mid 3 - - - \mid 9$$

Challenges in Syntax Analyzer

Development of parser is not only restricted to whether the set of tokens are generated by Context-Free Grammar or not.

- Needs to handle the following implementation issues:

Which Production to choose ?

Which Non-terminal to be expand ?

Process of Derivation

Given a Grammar G and string w of terminals in L(G) then, start symbol derives w.

If Start Symbol of G derives a string of terminals and non-terminals at any point in derivation process

- Then that string is known as **Sentential form of Grammar**.

Sentential Form of Grammar

$$
\begin{aligned}
List &\rightarrow \underline{List} - Digit \\
&\rightarrow \underline{List} + Digit - Digit \\
&\rightarrow \underline{Digit} + Digit - Digit \\
&\rightarrow 9 + \underline{Digit} - Digit \\
&\rightarrow 9 - 5 - \underline{Digit} \\
&\rightarrow 9 + 5 - 2
\end{aligned}
$$

Process of Derivation

If the leftmost non-terminal is replaced in any sentential form of Grammar then it becomes leftmost derivation.

The same concept get applied for the rightmost derivation.

There are some form of grammars which are Ambiguous Grammar.

- It generates more than one leftmost/rightmost derivation of a string.

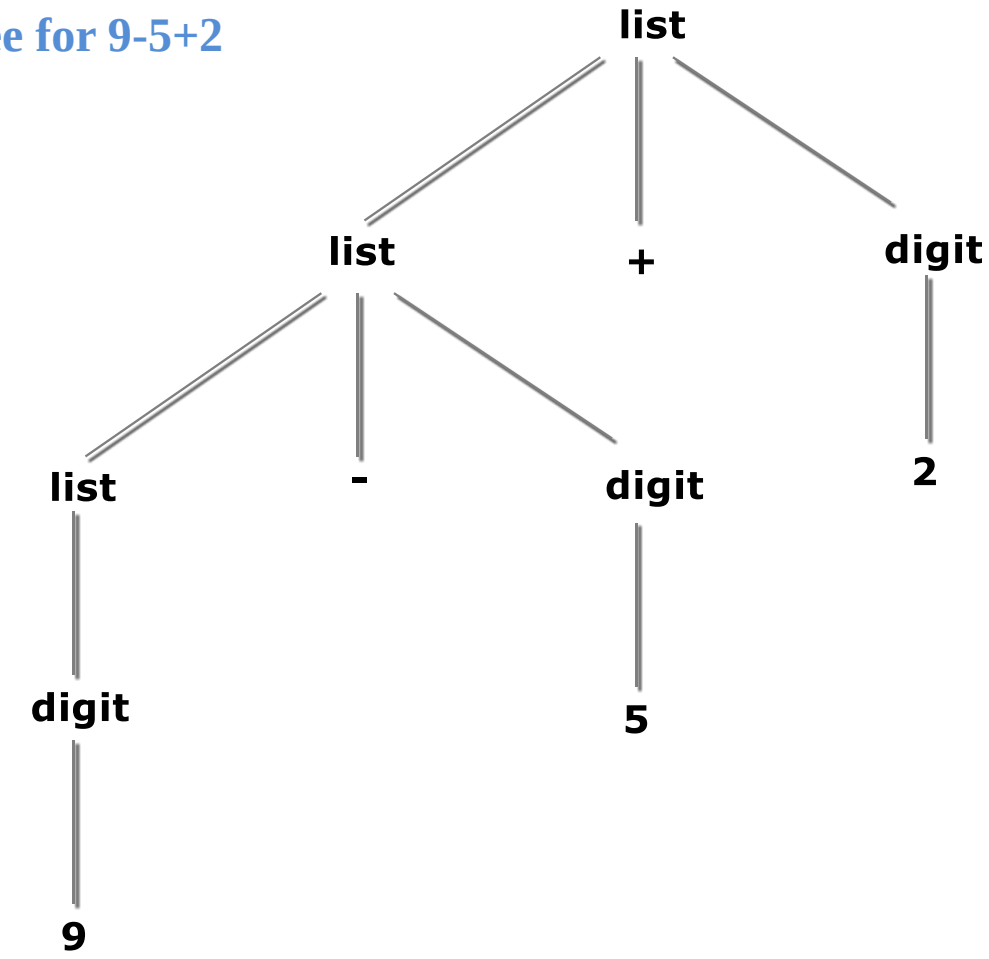
Parse Tree

It is a data structure that captures all the specifications related to the start symbol of grammar deriving a string in the language

Root of the tree will be labelled by start symbol, internal nodes are labelled by non-terminals and leaf nodes are labelled by tokens.

Example

Parse Tree for 9-5+2



Ambiguity

- A Grammar can have more than one parse tree for a string.
- Consider the Grammar

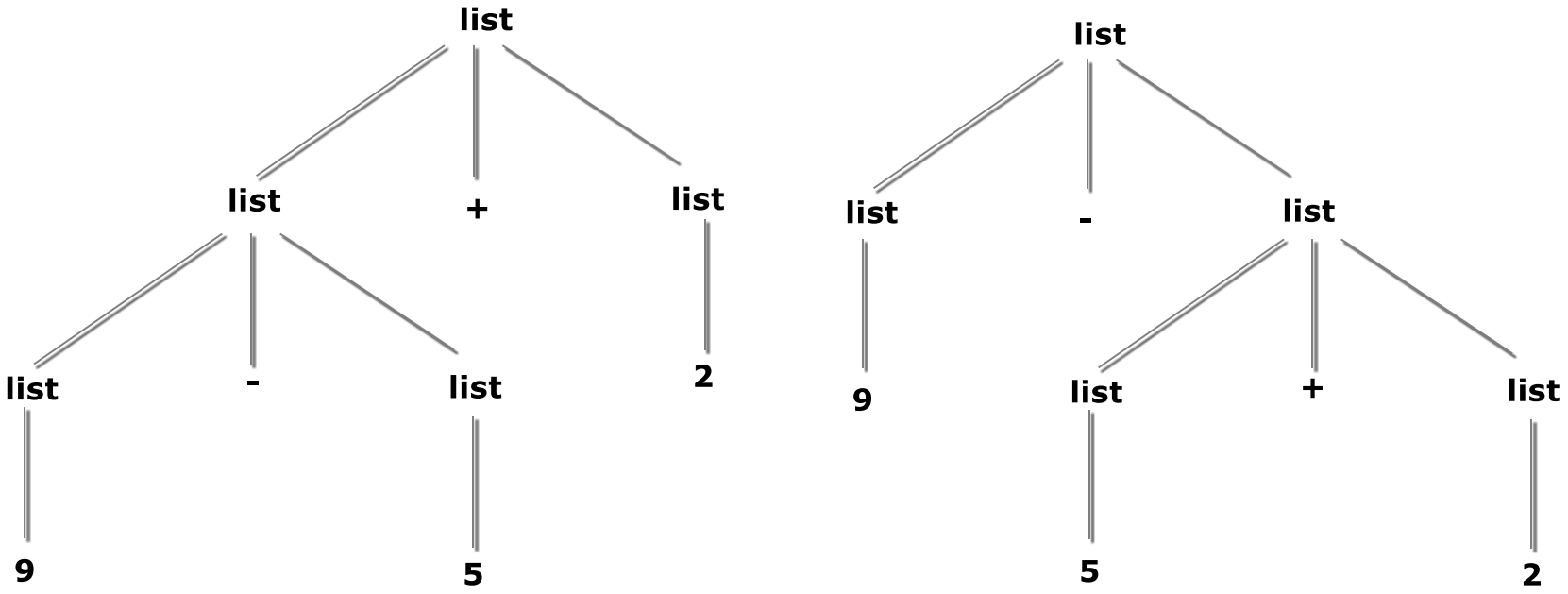
$$List \rightarrow List + List$$
$$\rightarrow List - List$$
$$Digit \rightarrow 0|1|2|---|9$$

MODIFIED GRAMMAR

$$List \rightarrow List + Digit$$
$$|List - Digit$$
$$|Digit$$
$$Digit \rightarrow 0|1|2|3---|9$$

OLD GRAMMAR

Example



String 9-5+2 has now two parse trees